POLITECNICO DI TORINO

MASTER's Degree in Computer Engineering

MASTER's Degree Thesis

Toward Efficient DDoS Detection with eBPF

Supervisors

Prof. Fulvio RISSO

Ing. Federico PAROLA

Ing. Simone MAGNANI

Candidate

Giuseppe OGNIBENE

April 2021

Abstract

In today's Internet, IT security is a key component that faces new challenges every day to offer services in a secure way. In this context, network monitoring represents the main point to be able to detect cyber attacks, and, in today's network infrastructure, it is increasingly implemented using NFV (Network Function Virtualization) technology where network services are implemented in pure software. This brings several advantages such as flexibility and cost reduction as these functions can be performed on general purpose hardware. In this context, eBPF (Extended Berkeley Packet Filter) is an excellent technology, suitable for creating network functions for fast packet processing in the Linux kernel. This thesis work was born with the intention of analysing the advantages, disadvantages and limitations of having a network monitoring using eBPF when it is used to provide the necessary information to a detection algorithm of DDoS attacks, called LUCID. LUCID is able to detect DDoS attacks through Deep Learning techniques, adapted for limited resources environments. Network monitoring was carried out using two frameworks that seamlessly integrate with LUCID. The first one is Polycube, an open-source project developed at the Politecnico di Torino, which allows the creation of extremely fast network monitoring programs. The second one is DeChainy, an open-source framework for creating and distributing network monitoring probes in eBPF.

During the thesis work, a number of tests were carried out aimed at obtaining all the information necessary to understand the limits in having a detection algorithm that runs on a single machine, responsible for both the security and forwarding traffic to end users. In addition, tests were also carried out aimed at analysing only the monitoring and data extraction parts. In the final part, we also thought about a distributed version of the attack detection system, where the single instance of detection runs on multiple machines simultaneously, in a parallel way, potentially allowing for resource savings and attack detection speed increase.

Table of Contents

List of Figures

Acronyms

ML
Machine Learning

DoS
Denial of Service

DDoS
Distributed Denial of Service

IP
Internet Protocol

TCP
Transport Control Protocol

DL
Deep Learning

NFV
Network Function Virtualization

eBPF
extended Berkeley Packet Filter

LUCID
Lightweight, Usable CNN in DDoS Detection

Chapter 1

Introduction

In the context of network monitoring, of particular interest is the monitoring aimed at coping with possible cyber attacks, which in recent years are becoming more and more frequent and for this reason, properly monitoring the network is a fundamental point to ensure a safe use of internet services.

Although enormous progress has been made in recent years in the field of cyber security, an ideal solution, 100% effective, has not yet been found, that is, one that can successfully block all malicious sources and at the same time ensure a service to good sources. Indeed, DDoS attacks remain one of the biggest threats on the Internet and IT environments.

1.1 Goal of the thesis

The goal of this thesis work is to collect as much data as possible about the network monitoring carried out with eBPF, coupled to a DDoS attack detection algorithm (LUCID). From testing results, an attempt was therefore made to understand the trend related to appropriate parameters vary, to better understand which components can be improved to provide a more efficient detection of DDoS attacks. Tests have been carried out to verify under what conditions the use of a single machine on which both the detection algorithm and the traffic forwarding are run, start to give performance problems, high resource consumption, increase in detection time, decrease in the number of attackers found in the unit of time and so on. In particular, we tried to divide the various components that form an IDS, in order to concentrate which component can represent a strength and which a weakness. The main reason that gave rise to this thesis was the need to find a way to overcome some problems that could arise in the utilization of detection algorithms that are used with the so-called *SPF* or *Single Point of Failure* and so, many (really many) tests of various types have been conducted, under different

conditions and a lot of interesting data has been collected.

As mentioned earlier, the detection algorithm used is called LUCID. It exploits the properties of Convolutional Neural Networks (CNN), a Deep Learning technical specification that advanced the state-of-the -art in some specific scenarios such as malware detection, network traffic analysis and intrusion detection. Furthermore LUCID was chosen because it integrates perfectly with Polycube and DeChainy which play the role of data provider on which LUCID performs its analysis.

1.2 Hypothesis

The world of detection is very wide, and, for this reason, in this thesis work we have focused on the detection of attacks on a data center. As a matter of fact in a data center, you usually update your hardware practically every year in order to save costs, especially those of electricity, and data centers potentially have a lot of unused hardware that can be used to detect DDoS attacks. Our hypotheses are the following:

- The data center has one or more entry points: we only focus on one point;
- Attacking the data center means: attacking the entry nodes as the internal servers are hidden from the outside world;
- In the data center we have N generic servers that offer a certain number of services;
- There is a load balancer with basic functionality that sorts the traffic on the various servers with a certain policy;
- On this data center there is dedicated hardware for detection, which is placed between the data center and the outside world and which must also take care of the forwarding of traffic. This could therefore represent a *Single Point of Failure* as shown in Figure [1.1].

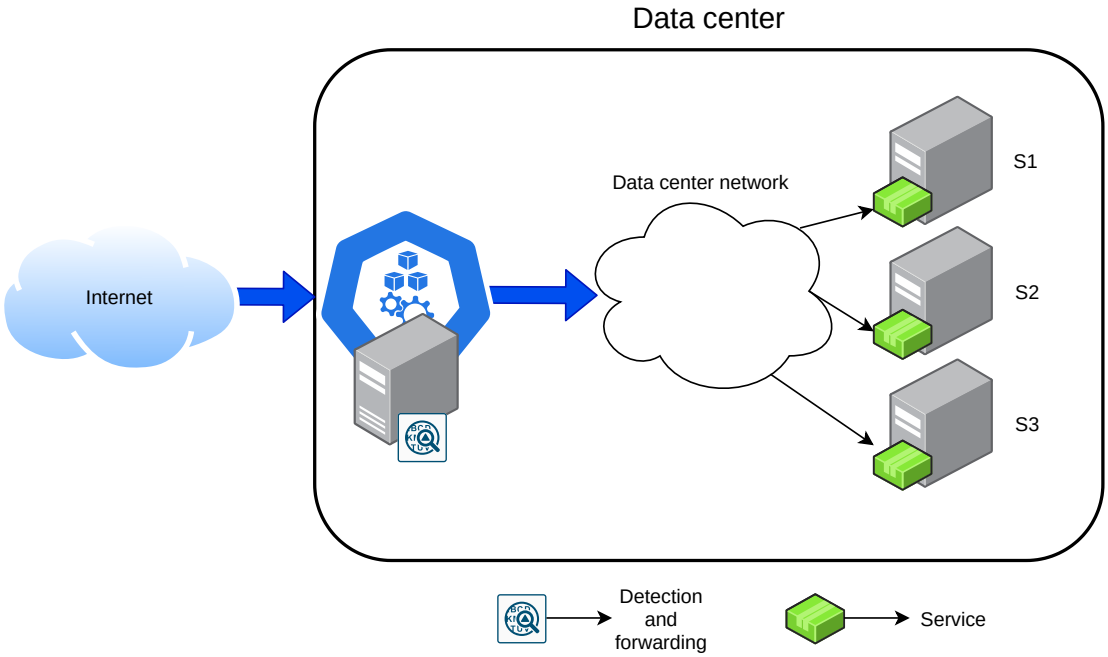

Figure 1.1: Possible representation of a data center.

1.3 Thesis structure

The thesis is organized as follows. Chapter 2 presents an overview of existing cyber attacks and describes the main detection algorithms and how they are applied to defend a network or data center. Chapter 3 explains the technologies and frameworks used for the tests carried out. Chapter 4 presents in detail the Polycube services, the relationship with LUCID, the most important parts of the eBPF probe code, and, finally the tools, architecture and tests configuration performed. In chapter 5 the tests carried out with LUCID are presented and the results obtained are explained. Chapter 6 explains the results obtained by DeChainy used only as a probe and extractor. Lastly Chapter 7 contains conclusions, with reference to possible solutions of some limitations encountered and also regarding possible future works.

Chapter 2

Background

This chapter presents what is meant by *Network Monitoring*, *Denial of Service* and what types of those exist, with particular focus on the *SYN Flood* attack, while, in the second part we provide an overview of detection.

2.1 Network Monitoring

Network Monitoring refers to the component of a system that constantly monitors the elements of a network and alerts the administrator, or another part of the network in case of failures, errors or otherwise.

Network Monitoring is done using appropriate diagnostic software tools, but also through specific hardware devices that are connected to the network and which analyse network traffic and the operation of network devices. The two main characteristics of having a constant monitoring of the network are:

- The ability to generate alarms that can be used by those who manage the network to perform the appropriate checks;
- The possibility to generate a report where all the problems found during monitoring are saved.

As mentioned above, these features are really important in the field of cyber security because it allows you to find out if a cyber attack is in taking place.

2.2 Cyber attack taxonomy

With *DoS* or *Denial of Service* attack we mean a type of cyber attack, aimed at exhausting the victim's resources, which can be, for example, a website or a web server, in such a way as to slow down or permanently block the provision

of a service. Antoher type of attack is the *DDoS* or *Distributed DoS* where the malicious traffic that reaches the victim comes from different sources.

Concerning *DDoS* attacks[1], they are usually distinguished them as:

- *Application layer attacks*: the goal of this type of attack is to exhaust the target resources to create a *DoS*. The attack target the level at which web pages are generated on the server and served in response to HTTP requests. This because a single HTTP request is computationally cheap to perform on the client side, but it can be expensive for the target server to respond, as the server often loads multiple files and executes database queries to create a web page. An example of this type of attack is the *HTTP Flood*;

- *Protocol attacks*: also known as state exhaustion attacks. Attacks of this type cause a disruption of service due to excessive consumption of server resources and/or the resources of network devices such as firewalls and load balancers. Weaknesses in layer 3 and layer 4 of the network protocol stack are exploited to make the victim server inaccessible. An example of this type of attack is the SYN Flood which will be explained in more detail later;

- *Volumetric attacks*: in this type of attack, an attempt is made to create congestion by consuming all available bandwidth between the target and the larger Internet. Basically, large amounts of data are sent to a destination using some form of amplification or other means to create massive traffic, such as requests from a botnet (computer network or other IoT device that has been infected with malware and commanded by a botmaster).

Here, a quick overview of the main flood cyber attacks [1] is provided with a focus on SYN Flood:

- *ICMP attack*: typically, *Internet Control Message Protocol (ICMP)* echo request and echo response messages are used to ping a network device for the purpose of diagnosing the health and connectivity of the device and the connection between the sender and the device. A *Ping Flood* is a type of DoS or DDoS attack where the attacker sends a large number of ICMP echo request packets. The *Ping Flood* attack aims to overwhelm the targeted device ability to respond to an high number of requests and/or overload the network connection with bogus traffic;

- *UDP Flood Attack*: a UDP flood is a type of DoS or DDoS attack where a large number of *User Datagram Protocol (UDP)* packets are sent in such a way that they overwhelm the device ability to process and respond. It mainly takes advantage of the steps a server takes when it responds to a UDP packet sent to one of its ports. Under normal conditions, the server that receives

a UDP packet on a particular port, performs two steps in response: it first checks if any programs are running which are currently listening for requests at the specified port, and if no programs are receiving packets on that particular port, the server replies with an ICMP packet to inform the sender that the destination was unreachable;

- *SYN flooding attack*: a *SYN Flood* (Figure [2.1] from [2]) is a type of Denial-of-Service (DDoS) attack that aims to make a server unavailable to legitimate traffic or otherwise slow it down by consuming all available resources. In this attack, a large number of initial connection request (SYN) packets are sent, thereby attempting to overwhelm all available ports on a target server machine. This type of attack works by exploiting the three way handshaking process of a TCP connection. Under normal conditions, the TCP connection is established through three different steps:

  1. The client sends a SYN packet to the server to initiate the connection;
  2. The server then responds to that initial packet with a SYN/ACK packet in order to acknowledge the communication;
  3. Finally, the client responds with an ACK packet to acknowledge receipt of the packet from the server. After completing this handshake, the TCP connection is open and it is capable of sending and receiving data.

  If the client does not respond with a final ACK, the connection is considered *half open*. In this type of DDoS attack, the victim server continually leaves open connections and it waits for each connection to time out before the ports become available again.

Protocols listed above, i.e. the ones exploited by attacks, are the same ones that are checked in the eBPF probe used for the tests.

Finally, in Figure [2.2], from [3], it is possible to see the percentage of attacks carried out in Q3 of 2020 divided by "attack vector", i.e. by the method used in the attack. As you can see, the SYN Flood attack is the most used one with a percentage of 42%.For this reason, in this thesis we paid more attention on SYN Flood ones.

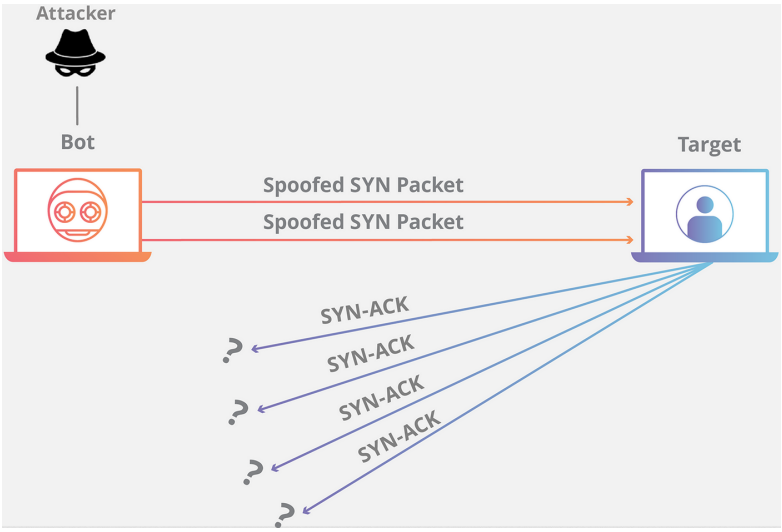

Figure 2.1: SYN Flood attack

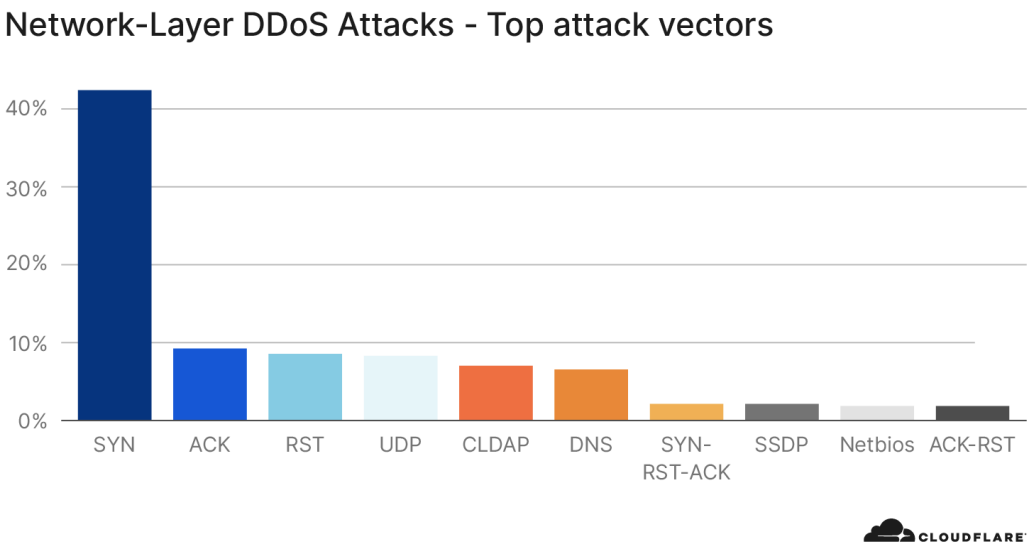

Figure 2.2: Top attack vectors.

2.3 Intrusion Detection Systems

In this section, we will first discuss the distinction between IDS and IPS, we will then describe the analysis methodologies and finally the technologies with which the IDS are implemented.

With the term "intrusion", we mean the attempt to compromise the so-called

CIA (Confidentiality, Integrity, Availability). Intrusion detection is then the process of monitoring the network and/or an IT system in order to find a possible intrusion. In this context, the Intrusion Detection System or IDS plays the role of a software or hardware component that is able to automate the detection of an intrusion, while, for Intrusion Prevention System or IPS we mean an IDS with the additional ability to stop an intrusion. It should be noted that an IDS differs from other defense components such as the firewall, as an IDS monitors the system while a firewall tries to prevent certain elements from entering. Figure [2.3], presented in Liu et al. [4] is a great overview of the IDSs.

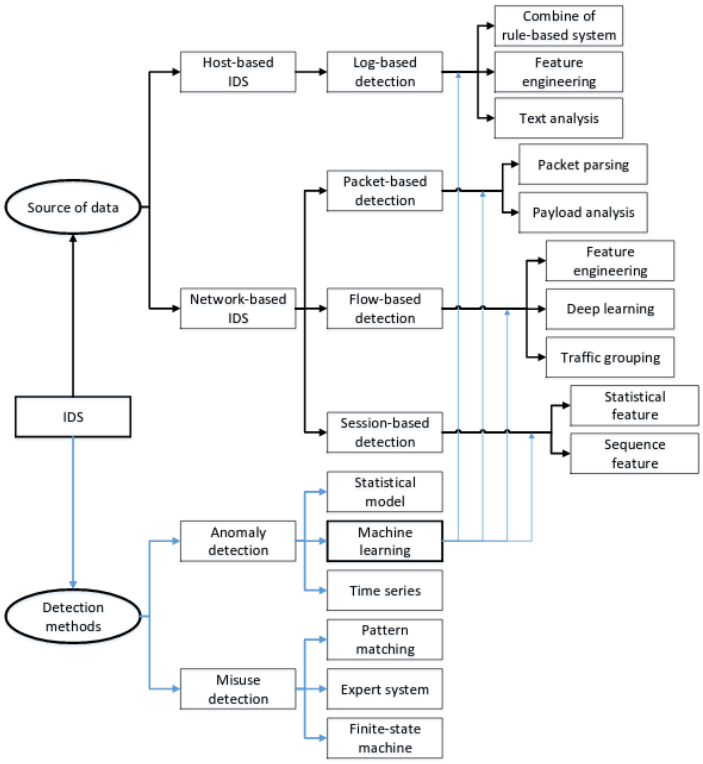

Figure 2.3: Taxonomy system of IDS

2.3.1 Methodologies for IDS

As described in Liato et al. [5], Intrusion Detection Systems methodologies are classified into three main categories: Signature-based Detection (SD), Anomaly-based Detection (AD) and Stateful Protocol Analysis (SPA).

Signature-based Detection

A signature is a pattern or string that matches a known attack, intrusion, or threat. Signature Detection is the process of comparing patterns with captured events to recognize possible intrusions. Signature Detection is also known as Knowledge-based Detection or Misuse Detection as it uses accumulated knowledge from specific attacks and system vulnerabilities to identify an attack. The advantages of this methodology are the simplicity of implementation, the low false positive rate and, at the same time, the high true positive rate for known attacks. The main disadvantages is that it cannot react to unknown attacks also known as "zero day" attacks or in any case to known attacks that are modified. It also requires a continuous signatures update (which can be both complicated and even expensive).

Anomaly-based Detection

An anomaly is defined as a deviation from known behavior, and the profiles represent normal or expected behaviors derived from monitoring regular activity, network connections, hosts, or users over a period of time. These profiles can be *static* or *dynamic* and can be generated for various attributes such as failed login attempts, processor usage, count of emails sent, etc. This methodology compares normal profiles with observed events to recognize significant attacks and for this reason it is also called Behavior-based detection since it is based precisely on behavioral analysis. The advantages are that it can also detect new or unexpected vulnerabilities and is not strictly dependent on the devices operating system. The disadvantages are that it is risk prone to build “weak” profiles due to the variations of the observed events (it is necessary to choose the right threshold), with a consequent high rate of false positives, in addition there is a phase of vulnerability during the learning phase.

Based on what is meant by "behavior" of the target system, AD can be divided into three main categories which are: Statistical-based, Knowledge-based, and Machine learning-based. This last category contains in turn other sub-categories, one of these includes the detection algorithm used in this thesis work.

Stateful Protocol Analysis

The stateful in SPA indicates that the IDS may know and track the protocol states, (such as matching requests and responses). The SPA methodology seems to resemble Anomaly-based but in reality they are essentially different. AD uses preloaded profiles specific to the network or host, while the SPA depends on generic profiles that are developed by the equipment supplier for a specific protocol. In general, the network protocol models in SPA are originally based on the original protocol standards of international organizations such as the IETF and IEEE. For this reason SPA is also known as Specification-based detection. The advantages of

this methodology are that it is able to recognize unexpected sequences of commands. The disadvantages are that it requires a high consumption of resources, fails to detect attacks that result in a benign behavior of a given protocol and finally there could be problems between the protocol model used and how it is implemented.

2.3.2 Types of technology for IDS

There are four main categories, divided according to where they are implemented to monitor suspicious activity and what kind of events they can recognize:

- Host-based IDS (HIDS);
- Network-based IDS (NIDS);
- Wireless-based IDS (WIDS);
- Network Behavior Analisys (NBA);
- Mixed IDS (MIDS), which is a hybrid technology of the first 4 categories.

A lack present in all IDS technologies is the presence of an accuracy that is never 100%. There are various indicators of the degree of accuracy, the most important are:

- True Positive: the IDS correctly identifies a malicious activity as such;
- False Positive: the IDS misidentifies a benign activity as malicious;
- True Negative: the IDS correctly identifies a benign activity as such;
- False Negative: the IDS identifies a malicious activity as benign.

Thanks to these four indicators it is then possible to trace other values that are useful for evaluating the goodness of an IDS or in any case of a detection algorithm such as: False Positive Rate, Accuracy, Precision, F1 score and so on.

Host-based IDS

A HIDS monitors and collects the characteristics of hosts that contain sensitive information, servers that provide public services, suspicious activities. This information can be taken from the kernel, the system, the network, and so on. A HIDS is based on a software agent that is installed on a single host and, for this reason, this is the only IDS technology capable of analysing communication activities with end-to-end encryption. The advantages are that it is possible to use a combination of various techniques so as to provide more robust protection and to collect as much

information as possible. The disadvantages are that due to the lack of information on the context there is a higher rate of false positives and false negatives. This is also due to the fact that the sharing of computational resources with the host can cause delays in the generation of alarms and report, and coexistence in the same OS can create conflicts with pre-existing security systems.

Network-based IDS

A NIDS monitors traffic on one or more specific network segments using appropriate sensors, it then analyses the activity of applications and also protocols to recognize suspicious activity. A NIDS is based on sensors that can be active or passive, installed in the various network segments and also on the hosts; in particular they can be loaded on the network cards configured in a promiscuous way, in order to accept all the incoming traffic. The advantages are that the sensors are able to analyse the application protocols with maximum visibility on the context and also the possibility of using a combination of various detection methodologies. The disadvantages are the high rate of false negatives and false positives, the inability to detect attacks using encrypted traffic, and the lack of support in handling high traffic loads.

Wireless-based IDS

A WIDS is similar to a NIDS, but it captures wireless traffic in ad-hoc networks, sensor networks, and mesh networks. A WIDS consists of various passive sensors that are installed on the WLAN nodes and also on the wireless clients, making the system very accurate, thanks to its selective focus on the activity of wireless protocols. A WIDS collects various data from the WLAN, from connected devices such as access points, controllers, and so on, but also from terminal clients. The disadvantages are that it is not suitable for monitoring application, transport and network layers, the sensors are susceptible to physical interference attacks, and cannot compensate for the use of unsecured wireless protocols.

Network Behavior Analisys

An NBA system inspects network traffic to recognize attacks with unexpected traffic flows. It is based on mainly passive sensors that are installed in the network, it is able to reconstruct the dynamics of malware infections and DDoS attacks. In practice, it goes to see what happens inside the network, aggregating information from various points and, then, making offline analysis. The main limitation is the delay in detecting attacks caused by the processing of data streams in the form of blocks and not in real time. The various data sources can be hosts, OSes and even the services of various protocols.

2.3.3 Architecture of an IDS

Let's now briefly see the possible architectures of an IDS:

- Centralized: it collects and analyse data from a single monitored system;
- Distributed: it collects data from multiple monitored systems to detect single, distributed and cooperative attacks;
- Hybrid: it represents a mix of the two previous architectures.

Distributed detection

In addition to collecting data to try to understand how to improve the detection of DDoS attacks, using eBPF, we also looked at a possible implementation of a distributed version of an IDS. This is because the distributed world can be an interesting solution in detecting DDoS attacks.

We found a lot of documentation that talks about distributed algorithms for detecting attacks on a network and in more networks. This documentation was useful to understand the logic behind distributed algorithms, both for the exchange of information between nodes but also to understand how detection is accomplished (in some cases, since some of them only offer the possibility of distributing the algorithm leaving the choice of how to detect to the end user).

In Hindy, Hanan, et al. [6], an overview about the distributed IDS is shown, taking into account the decision phase, the infrastructure used and the calculation position of the detection.

For example, in [7], they propose a collaborative DDoS detection method, which deploys network monitors on edges of networks. Each monitor adopts a centralized single point detection algorithm. All monitors are organized into AutoTree through a Collaborative Platform (CP). Results of partial detections are integrated step by step and ultimately gathered at the root node of AutoTree, which makes the final judgment. All collaborative nodes work together by sharing results of local detection, breaking limitation of centralized single point detection, as well as making it possible to detect low-profile DDoS attacks hiding in high-volume normal traffic at early stage of the attack. According to them, their approach can effectively reduce false positive and false negative rate of DDoS detection. Nodes keep summarize local traffic into Sketch matrix, and store a time series of Sketch matrixes. Through reverse sketch, one can obtain corresponding anomalous IP address, thus saving memory and computational resources of collaborative nodes.

Another good example of work in the distributed and collaborative world is [8]. This paper presents an innovative approach that coordinates distributed network traffic Monitors and attack Correlators supported by Open Virtual Switches (OVS). The Monitors conduct anomaly detection and the Correlators perform deep packet

inspection for attack signature recognition. They collaboratively look for network flooding attack signature constituents that possess different characteristics in the level of information abstraction. Therefore, this approach is able to not only quickly raise an alert against potential threats, but also to follow it up with careful verification in order to reduce false alarms.

Chapter 3

Used technologies

This chapter provides a description of the key technologies on which this thesis is based and the frameworks used.

3.1 eBPF (Extended Berkeley Packet Filter)

Extended Berkeley Packet Filter (eBPF) [9] represents the enchanced version of BPF. It was proposed by Alexei Staravoitov in 2013 and it has first appeared in Kernel 3.18 and renders the original version, which is being referred to as "classic" BPF (cBPF) these days mostly obsolete. eBPF is a recent technology that enables flexible data processing thanks to the capability to inject new code in the Linux kernel at run-time, which is triggered each time a given event occurs. cBPF was born in 1992 and was a very simple VM used to perform in-kernel packet filtering (a famous example is tcpdump).

The important thing is that (BPF) [10] was mainly used to create packet filtering programs, mainly used in monitoring activities. eBPF is very promising due to some characteristics that can hardly be found all together, such as the capability to execute code directly in the vanilla Linux kernel, hence without the necessity to install any additional kernel module; the possibility to compile and inject dynamically the code; the capability to support arbitrary service chains; the integration with the Linux eXpress Data Path (XDP) for early (and efficient) access to incoming network packets. At the same time, eBPF is known for some limitations such as limited program size, limited support for loops, and more, which may impair its capacity to create powerful networking programs.

In the following pages we see some of the most important features of eBPF [9].

3.1.1 Verifier

A very important concept of eBPF is safety. Since an eBPF program can be loaded at runtime in the kernel, it is checked by the verifier, which ensures that the given program cannot harm the system. The verifier checks various aspects of an eBPF program such as:

- The program must not have infinite loops;
- The program must not use uninitialized variables or access out-of-bounds memory;
- The program must be of a certain size that meets the system requirements;
- The program must have a finite complexity. The verifier will evaluate all possible execution paths and must be able to complete the analysis within appropriate limits.

3.1.2 Helper Functions

Technically creating code that does complex operations with eBPF could be critical because the C of the eBPF is limited. The solution that comes to the aid of developers are helpers. Helpers are native software functions in the Linux kernel that can be called within an eBPF program. A simple example of a helper is bpf_trace_printk() [11], which is a "printk()-like" facility for debugging. An interesting thing about helpers is that it is possible to put all the loops we want unlike the eBPF where they are forbidden. However, while the eBPF code can be injected on demand, the helpers must be compiled a priori into the Linux kernel. So it is possible to use the helpers already listed in the Linux kernel or, in case there is no helper that suits us, we can create a new one, however, to use it it must be added to the Linux kernel In brief, helper functions allow eBPF programs to consult a kernel-defined set of function calls to retrieve and send data to and from the kernel. The support functions available may differ for each type of BPF program.

3.1.3 Maps

One of the characteristics of traditional BPF was that it was stateless, this has been changed in eBPF, indeed here you have a memory where to save the state. The main purpose is to export data from the kernel to userspace, push data from userspace to the kernel or share data between different eBPF programs. The problem with having shared memory is concurrency. The solution that has been defined in eBPF is to define non-generic but typed memory, that is a data structure

called map [Figure 3.1], which has some particular form such as vector, hashmap, table and so on and of which you can also have more than one simultaneously. If maps are used, the concurrency does not have to be managed by the programmer but the system takes care of it. The maps can also be nested, i.e. if it is possible to have maps of various types of maps. In brief, maps are efficient key/value stores that reside in kernel space.

They can be accessed from a BPF program in order to keep state among multiple BPF program invocations. They can also be accessed through file descriptors from user space and can be arbitrarily shared with other BPF programs or user space applications or between user and kernel space. BPF programs which share maps with each other are not required to be of the same program type.

There are different types of maps, which have behaviors and structures that distinguish them. There are both generic maps, some of which are:

- BPF_MAP_TYPE_HASH;
- BPF_MAP_TYPE_ARRAY;
- BPF_MAP_TYPE_PERCPU_HASH;
- BPF_MAP_TYPE_PERCPU_ARRAY.

There are also non-generic maps, some of which are:

- BPF_MAP_TYPE_PROG_ARRAY;
- BPF_MAP_TYPE_PERF_EVENT_ARRAY;
- BPF_MAP_TYPE_PERF_EVENT_ARRAY;
- BPF_MAP_TYPE_STACK_TRACE.

Some maps also have a PERCPU version that allows you to have different instances of the same table for each CPU core, which leads to an improvement in performance. No synchronization mechanism is needed and maps can also be cached for a further increase in access speed.

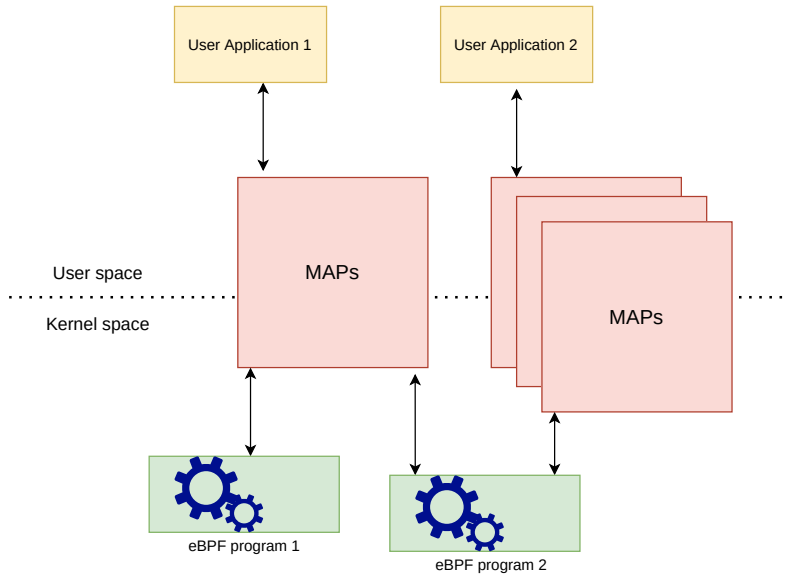

Figure 3.1: Shared memory: architecture.

3.1.4 Tail calls

The concept of tail calls can be used to overcome the size limit of an eBPF program. In practice, thanks to tail calls, an eBPF program can call another without going back to the old program. Indeed, tail calls are different from the concept of function calls. A tail call has minimal overhead and it is implemented as a long jump, reusing the same stack frame.

3.1.5 Program Types

There are different types of programs, this is because the execution of an eBPF program is triggered by a specific event in the kernel, called Hook Point. With eBPF there are a number of hook points where you can listen for any events. These hook points are located at different levels in the Linux networking stack. An event can be captured as soon as it exits the card network, as soon as the Operating System comes into play, in sockets or at the RAW level (traditional BPF). The metadata associated with packets and allowed actions change according to the hook used. For networking purposes, program execution starts when a packet arrives. Two of the main types for networking are XDP (eXpress Data Path) and tc (traffic control).

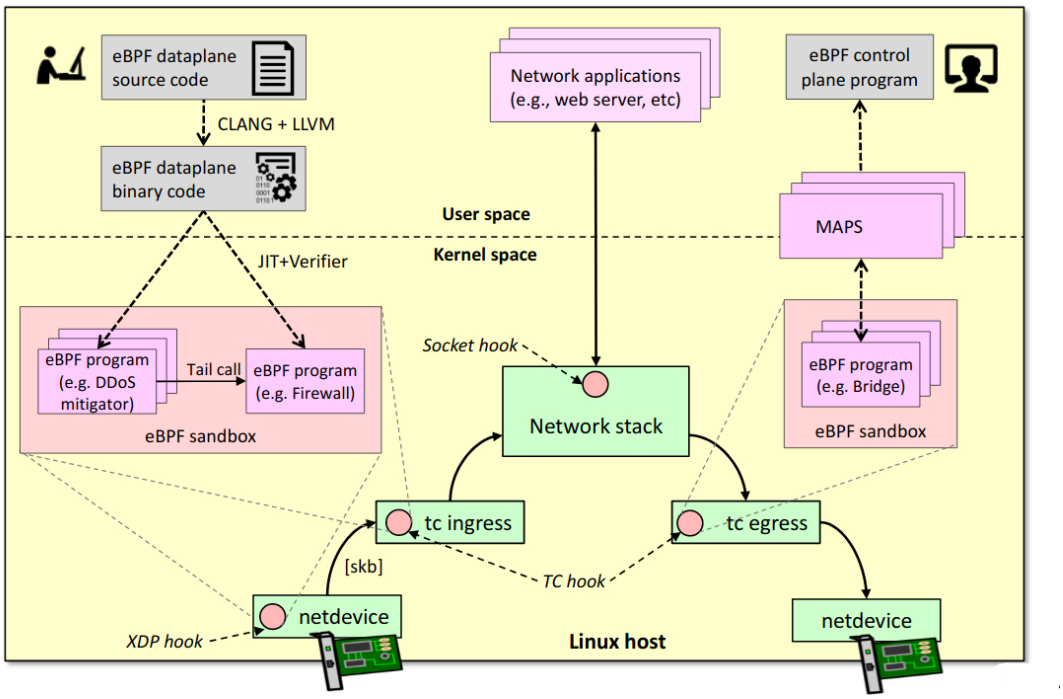

Figure 3.2: Graphical representation of XDP and TC hook points.

XDP: eXpress Data Path

XDP [10] provides a mechanism to run an eBPF program at the lowest possible level in the Linux networking stack, basically immediately after receiving a packet. No expensive operations are performed at this level of the network stack, indeed here we are before any allocation of kernel metadata structures such as skb, spending fewer CPU cycles for packet processing than conventional stack delivery, but from on the other hand, the information provided to the program is poor. XDP has three operating modes:

- Driver (or Native) mode: the network card driver must support this model. Note that, a device that runs in XDP Driver can redirect a packet only to another device running XDP Driver;
- Offloaded mode: the BPF program is offloaded directly into the NIC instead of running on the host CPU. This is usually implemented in so-called SmartNICs;
- Generic (or SKB) mode: in this case, XDP can also be used with drivers that do not offer native support. Note that, a device that runs in XDP Generic

can redirect a packet to all the devices that run in XDP Generic.

Possible use cases are as varied as: early packet discard (for example DDoS mitigation), firewalling, load balancing, forwarding. After the XDP program has been executed, an appropriate value is returned [Figure 3.3]:

- XDP_DROP: the packet is dropped directly at the driver level, without wasting any other resources;
- XDP_PASS: the packet can keep going up the network stack. In this case, the CPU that is processing the packet allocates an skb, populates it and passes it forward to the GRO (Generic Receive Offload) engine;
- XDP_TX: the packet just received is modified and/or checked and then is sent back to the same NIC it came from. This type of action is used for example to make load balancer;
- XDP_REDIRECT: the packet is redirected to another NIC;
- XDP_ABORTED: indicates an exception; in this case the behavior is the same as XDP_DROP except that XDP_ABORTED passes trace_xdp_exception tracepoint which can be further monitored to detect incorrect behavior.

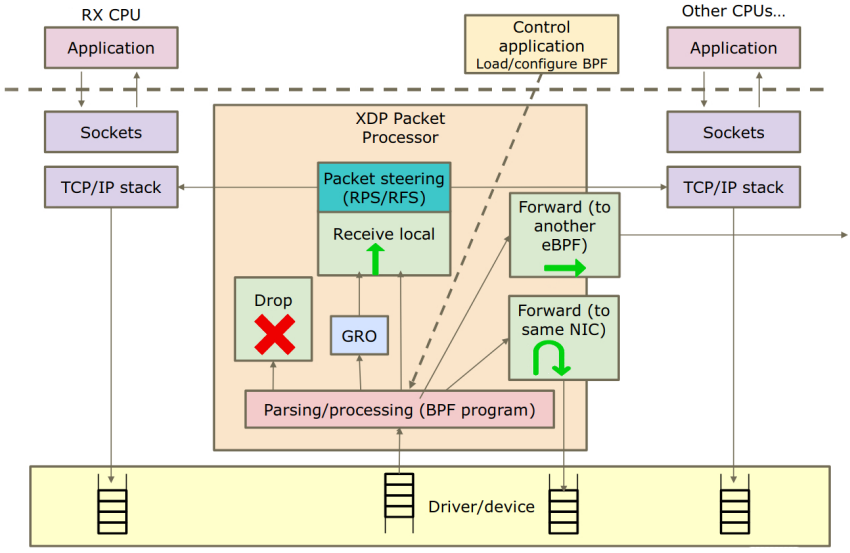

Figure 3.3: Graphic representation of the possible actions of an XDP program.

Obviously, like all things, XDP has not only advantages, indeed:

- With XDP it is possible to use only a limited number of helpers (compared for example to those that can be called with TC;
- XDP Driver mode limitations: most XDP-enabled drivers today use a specific memory model (e.g., one packet per page) to support XDP on their devices. Among the different actions allowed in XDP, there is the possibility to redirect the packet to another physical interface (XDP_REDIRECT). While this action is currently possible within the same driver, in our understanding, it is not possible between interfaces of different drivers;
- Generic XDP limitations: in case XDP is used without driver support, the XDP program is executed immediately after the skb allocation and therefore in practice loses the advantages that come from the Driver mode. Although it must be said that it continues to perform better than TC;

tc (traffic control)

tc programs intercept data when it reaches the traffic control function of the kernel, in RX or TX mode. Compared to XDP, a TC program [9]:

- The BPF input context is a sk_buff not a xdp_buff. When the kernel's networking stack receives a packet, after the XDP layer, it allocates a buffer and parses the packet to store metadata about the packet. This representation is known as the sk_buff;
- Compared to XDP, tc BPF programs can be triggered out of ingress and also egress points in the networking data path as opposed to ingress only in the case of XDP;
- The tc BPF programs do not require any driver changes since they are run at hook points in generic layers in the networking stack. Therefore, they can be attached to any type of networking device.

3.2 BCC

BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. Thanks to BCC it is possible to write eBPF programs much easier, this is made possible thanks to the kernel instrumentation in C (which includes a C wrapper around LLVM) and front-end in Python and lua. It is suitable for many tasks, including performance analysis and network traffic control.

3.3 Polycube

Polycube [12] is an open-source software framework based on eBPF, that enables the creation of arbitrary and complex network function chains. A powerful in-kernel data plane and a versatile user-space control plane with strong isolation, persistence and composability characteristics can be used in every function. In addition, a common model for each network function's control and management strategy simplifies the manageability and speeds up the implementation of new network services.

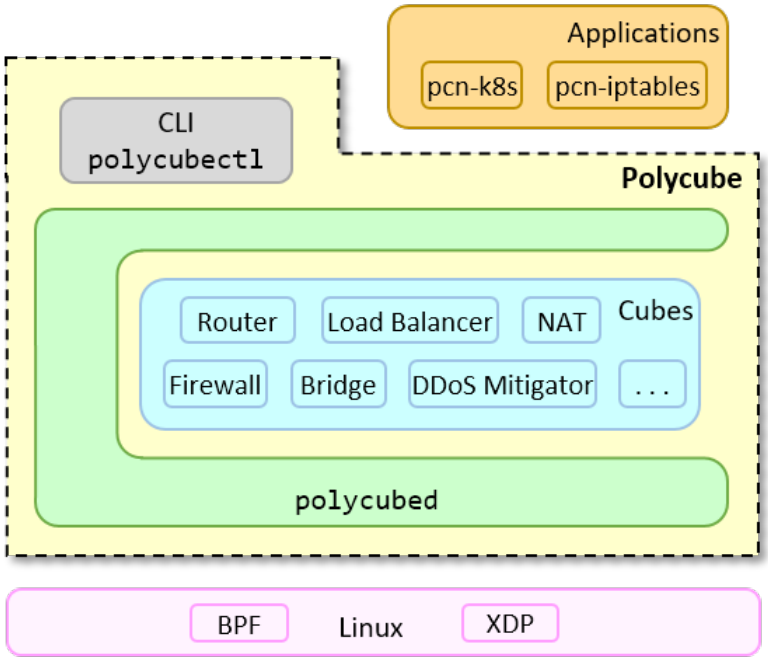

Figure 3.4: Polycube architecture

Polycube architecture main points are:

- Polycubed is a service independent daemon that allows you to control the entire polycube service, starting from startup, through configuration and stopping all available network functions, i.e. services. This module mainly acts as a proxy, that is, it receives a request from its REST interface, forwards it to the appropriate service instance and responds to the user. Polycube supports both local services which are implemented as shared libraries, which are installed on the same server as polycubed and whose interaction occurs through direct calls, but also remote services which are implemented as remote daemons even running on a different machine, which communicate with polycubed via gRPC;

- A Polycube service can be seen as a plug-in that can be installed and started at runtime. Each service must implement a specific interface to be recognized and controlled by the polycubed daemon. Each network function is implemented as a separate module and multiple versions of the same function can coexist, this is because there may be cases where a simple and fast version is needed and therefore a reduced set of functions may suffice, but, there may be cases where the full but slower version is needed. Each implementation of the service includes the datapath (Data Plane), i.e. the eBPF code to be injected into the kernel, the control/management plane, which defines the primitives that allow you to configure the behavior of the service, and the slow path, which manages packets that cannot be fully processed in the kernel;

- Polycubectl represents the command line (CLI) which is independent of the service and which allows you to control the entire system, such as starting/stopping services, creating service instances and configuring/querying the extension of each individual service. This module cannot know a priori which service it will have to control and therefore its internal architecture is service-agnostic, i.e. it is able to interact with any service through a well-defined control/management interface that must be implemented in each service. To facilitate the programmer life in the creation of Polycube services, there is a set of tools for automatic code generation capable of creating the skeleton of the control/management interface starting from the YANG (Yet Another Next Generation) model of the service itself.

3.4 DeChainy

This section briefly describes what DeChainy [13] is and what it was for. DeChainy is an open source framework to easily build and deploy eBPF/XDP network monitoring probes and clusters of probes, in order to perform Service Programs Chain efficiently.

DeChainy is not supposed to substitute already existent framework; on the other hand, its aim is to offer a wider choice of possibilities for also those users who are experimenting eBPF with other Python-friendly technologies, like TensorFlow (for Machine Learning). In other words, DeChainy is a Polycube-like framework, where the controlplane management which is written in Python and where the cube concept does not exist. It is a recent framework, perhaps not complete where, however, it is possible to define network functions in the controlplane in an easy and fast way.

3.5 LUCID

As explained in Chapter 2, Anomaly-based detection is divided into various categories, one of these is that of Machine-learning which in turn is part of Deep Learning (DL) based detection. Deep Learning systems are very effective in discriminating DDoS traffic from benign traffic, however they can be resource-intensive from a training point of view. Going into even more detail, we find the Convolutonial Neural Networks which represent a specific Deep Learning technique that have become popular in recent times leading to important innovations. The CNN application has advanced the state of the art in some specific scenarios such as malware detection, code analysis, network traffic analysis and intrusion detection in industrial control systems.

LUCID [14] (Lightweight, Usable CNN in DDoS Detection) is a lightweight, Deep Learning-based DDoS detection framework. It leverages the properties of *Convolutonial Neural Networks* to classify traffic flows as malicious or benign and it is suitable for online resource-constrained environments. Using the most recent datasets, LUCID is able to match the accuracy of the detection to the existing state of the art, obtaining a reduction of about 40 times of the processing time, compared to the state of the art. LUCID's CNN encapsulates learning of malicious activity from traffic to allow for the identification of DDoS patterns regardless of their time positioning. An advantage of CNNs is that they produce the same output regardless of where a pattern appears in the input. This encapsulation and learning of features during model training eliminates the need for excessive feature design, classification, and selection. In order to support an online attack detection system, LUCID uses a new preprocessing method for network traffic that generates a spatial data representation used as input for CNN.

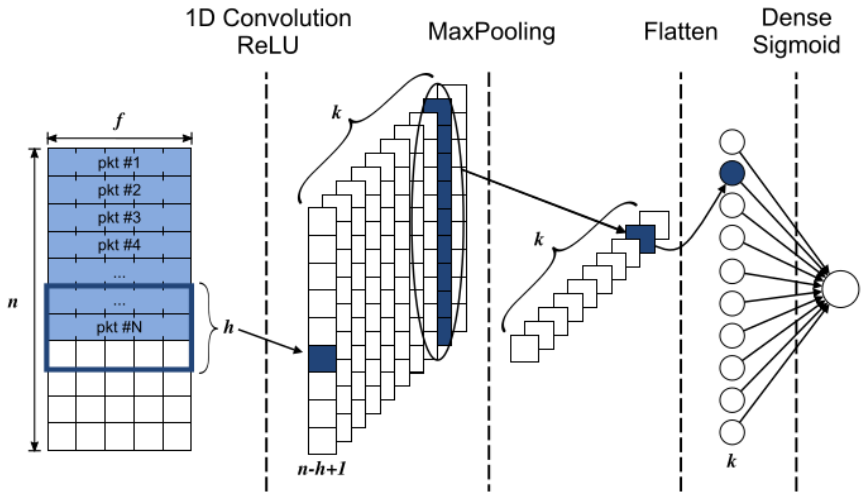

Figure 3.5: LUCID architecture

An important thing about LUCID is the concept of Custom Policy which can be applied to the results obtained from LUCID's prediction, in this way LUCID can be used in various situations and the policy can be decided and changed by the end user. LUCID's output is represented by a quintuple that is a session called flow [A], with source IP, source port, destination IP, destination port, protocol type and a value that can be True or False depending on whether LUCID considers that session as harmful or not. An example of a very simple Policy (which had been used in preliminary tests) that allows us to find out if a source IP address is malicious and separate it from the concept of quintuple is the following:

- a True result is assigned a value of +1;
- a False result is assigned a value of -1.

Finally, if the sum is greater than or equal to 0, that source IP address is deemed malicious, otherwise it is not. In this way, for example the final result of LUCID can be inserted in a DDoS mitigator.

Chapter 4

Architecture

In this chapter the two Polycube services used in the tests are explained in more detail, also analysing the architecture and the relationship between LUCID and Polycube, then the important parts of the eBPF code of the probe used for monitoring will be analysed. In the final part, the general architecture used for the tests will be shown with the appropriate configurations.

4.1 Used services

4.1.1 Helloworld service

The Helloworld [15] service represents the classic "Hello World" of any programming language. Indeed, it is the simplest service in the Polycube framework. The Helloworld service receives traffic from an interface and can:

- Forward traffic on a second interface;
- Forward traffic in the slow path;
- Drop traffic.

The important parts in the code of the ingress Data plane of this service are:

- BPF_ARRAY (action_map, uint8_t, 1); eBPF map of a single element that is used to save the action that is set using polycubectl with the *action* flag;
- BPF_ARRAY (ports_map, uint16_t, 2); eBPF map of two elements where the ID of the ports to be used are saved, the ports are always set with polycubectl.

This service was used in tests carried out with LUCID and Polycube to simulate the forwarding of traffic on the victim machine and also to have a baseline of forwarded traffic.

4.1.2 Dynmon service

The Dynmon (Dynamic Network Monitor) [16] service is a transparent service, it means, it has no forwarding capability and it is not connected to other services but is attacked. It allows the injection of eBPF code into the Linux kernel also enabling network monitoring with the consequent possibility of collecting and exporting custom metrics.

This service leverages the capabilities of Polycube to replace the eBPF code that runs in the dataplane and use the eBPF maps to share data between the data plane and the control plane. So thanks to Dynmon it was possible to inject some eBPF code (explained later) used to monitor incoming traffic and collect appropriate metrics on which LUCID will then carry out its analysis.

In order to configure the dataplane of the VNF for a cube of type Dynmon, it is not possible to use polycubectl but it is necessary to use an appropriate script to provide a configuration file in the JSON format that follows the YANG containers structures and that sends that JSON to the Polycube control plane. The configuration file must contain for the ingress-path and egress-path field the following fields:

- "name": it is the name of the configuration;
- "code": it is the eBPF code to be injected in the kernel suitably formatted;
- "metric-config": it is the list of the metrics that will be exported;
- "extraction-options": they are the extraction options of a given metric. In particular:
  - "swap-on-read": it is used to have exchangeable maps, in this way their reading is performed in an atomic way with respect to the dataplane;
  - "empty-on-read": in this case, the content of the map is cleared once it has been read.

Listing 4.1: Configuration file in JSON format.

```
{
    "ingress-path": {
        "name": "probe name",
        "code": "...",
        "metric-configs": [
            {
                "name": "...",
                "map-name": "MAP_NAME",
                "extraction-options": {
                    "swap-on-read": ...,
```

```
11                      "empty-on-read": ...
12                  }
13              }
14          ]
15      }
16  }
```

As for the metrics to be exported at the user level, they can be of two formats, each of which has its own endpoint:

- /metrics: returns metrics in JSON format;
- /open-metrics: returns metrics in OpenMetrics format.

Regarding the Dynmon map extractor, it supports all major eBPF map types (HASH, PERCPU HASH, QUEUE, STACK, ARRAY, PERCPU ARRAY, LRU HASH AND LRU PERCPU HASH).

CodeRewriter

A very important part about Dynmon's architecture, which is also done in DeChainy, is the swap of the eBPF program used as a probe. In Polycube, there is a component called CodeRewriter that takes care of dynamically adapting the injected code based on the initial configuration file. This component uses a pattern to identify, replace, and ultimately modify certain parts of the code provided during configuration. There are two types of rewrite: PROGRAM_INDEX_SWAP and PROGRAM_INDEX_RELOAD.

PROGRAM_INDEX_SWAP: this type of rewriting exploits the use and creation of program chains that the framework allows to execute. Practically:

1. Clones the injected eBPF code;
2. Modifies in the cloned code all references to all maps that have been declared as "swap-on-read";
3. Create a pivot program that, which will forward inbound or outbound traffic to the right program;
4. Injects into the system first the pivot code, then the code entered by the user and finally the cloned code.

Whenever metrics are requested from a user, the Control plane changes the index, which is contained in the shared eBPF map, which corresponds to the version of the program to be called by the pivot program. Since the maps in the two versions

of the program are different, the metrics that will be returned to the user are recovered from the one actually unused. In this way, atomicity is guaranteed as the Data plane program has started using the other map as soon as the request arrives.

PROGRAM_INDEX_RELOAD: this type of rewriting is used in case there is some error during the PROGRAM_INDEX_SWAP or the syntax of the maps used by the user in the eBPF code is wrong. This solution has some drawbacks in terms of performance. Practically:

1. Clones the injected eBPF code.
2. Change the name of the maps declared "swap-on" read in the cloned code.
3. Alternately inject the original eBPF code and the modified one into the system.

Since the obtained codes are alternately injected into the system, that is, they are always recompiled and reinjected, this second solution is slower than the first. However, even with this solution, the Control plane can atomically read the maps.

4.2 eBPF code of the probe

Here we see the eBPF [17] code that is injected into the Linux kernel. It is used to analyse incoming traffic on an interface and to extract metrics and it is the same in both Polycube and DeChainy tests, except for some small configuration changes. The following parameters are very important:

- N_SESSION: represents the maximum number of TCP sessions that are traced;
- N_PACKET_PER_SESSION: it represents the maximum number of packets of the same TCP session. If this limit is not reached, a padding function is used in the LUCID code (this is necessary to pass the data in the correct format to the neural network); if the maximum value is reached, no more packets will be captured for that session, until a request is made to obtain the collected metrics, which in our case is made by LUCID;
- N_PACKET_TOTAL: it represents the maximum number of packets that can be captured, usually N_SESSION*N_PACKET_PER_SESSION.

Each packet that belongs to a TCP session or to one of the other protocols implemented such as UDP or ICMP, is analysed, and from this, some informations are extracted and stored in the metric map for later consultation. The two important data structures in the code are the following:

```
BPF_QUEUESTACK_SHARED("queue", PACKET_BUFFER_DDOS, struct features,
    N_SESSION * N_PACKET_PER_SESSION, 0) __attribute((SWAP));
BPF_TABLE_SHARED("hash", struct session_key, uint64_t,
    SESSIONS_TRACKED_DDOS, N_SESSION) __attribute((SWAP));
```

The PACKET_BUFFER_DDOS map is a Queue (this means that they are automatically deleted when a read takes place) of size N_PACKET_TOTAL. This map contains all the packets that the probe is able to capture filtered according to their features. The SESSIONS_TRACKED_DDOS map is an Hash map of size given by the value of N_SESSION. It contains session_key as the key and a uint64_t as the value, it means that it is used to store the total number of packets captured for each session.

In particular in Listing [4.2] we can see the definition of the struct session_key. This structure is used to uniquely identify a session and it has 5 fields: source IP address, destination IP address, source port, destination port, protocol ID. In Listing 4.2, we can see the structure representing the features that need to be exported at the user level. For simplicity, the features used with Polycube have been displayed, while with DeChainy there is also the possibility to activate only some features based on the protocol of the packet.

Listing 4.2: Session identifier.

```
struct session_key {
    __be32 saddr;
    __be32 daddr;
    __be16 sport;
    __be16 dport;
    __u8   proto;
} __attribute__((packed));
```

Listing 4.3: Features to be exported.

```
struct features {
    struct session_key id;
    uint64_t timestamp;
    uint16_t ipFlagsFrag;
    uint8_t tcpFlags;
    uint16_t tcpWin;
    uint8_t udpLen;
    uint8_t icmpType;
} __attribute__((packed));
```

- id: it is the session identifier;
- timestamp: it is the timestamp of when the packet has been analysed;

- ipFlagsFrag: it represents the IP field Flags and the Fragment Offset (16 bytes in total, 3 for flags);
- tcpFlags: it is the flags specified in the TCP header;
- tcpWin: it is the size of the TCP window;
- udpLen: it is the length of the UDP packet;
- icmpType: it is the type of the ICMP packet.

For each packet that is captured [A], the probe checks the type of protocol used, then checks whether the current session identifying the packet has already reached the maximum packet value it can be stored(N_PACKET_PER_SESSION). If free space is still present, the features are extracted from the packet, otherwise, it moves on to the next packet.

The Figure [4.1] shows the relationship between LUCID and Polycube.

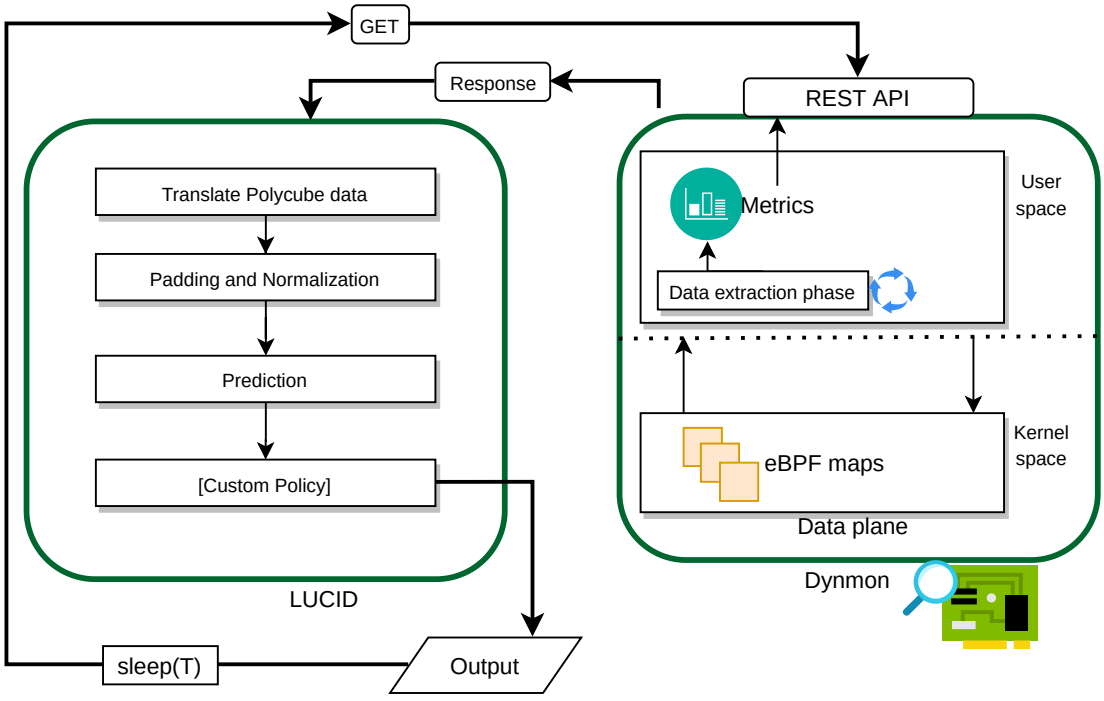

Figure 4.1: Architecture of LUCID and Dynmon

Briefly:

- LUCID makes a GET to the Polycube REST API, contacting a cube of type Dynmon (in our case it will contact a cube named lucid1);
- Dynmon receives the request and starts extracting data from the eBPF maps, then translates them into an appropriate format and replies to LUCID;
- Here LUCID performs a translation first, then there are the padding and data normalization operations and then the prediction part takes place;
- A Custom Policy can also be applied following the LUCID output;
- After LUCID gives its result, it can wait for a certain period of time (default 10 seconds, in conducted tests 0 seconds) before performing a new GET.

It has been chosen to visualize the relationship between LUCID and Polycube because it is much easier to represent and understand. In the case of LUCID and DeChainy, the path is logically similar, except for the fact that you do not go through a REST API and it is DeChainy who chooses how often to extract the data through an appropriate configuration.

4.3 Used tools

Below is an overview of the tools used to run the tests and to obtain the results.

4.3.1 MoonGen

In order to create a DDoS attack, there are many tools and many ways. Our choice fell on MoonGen [18], which is a scriptable high-speed packet generator built on libmoon. The whole load generator is controlled by a Lua script: all packets that are sent are crafted by a user-provided script. Thanks to the incredibly fast LuaJIT VM and the packet processing library DPDK, it can saturate a 10 GbE Ethernet link with 64 Byte packets while using only a single CPU core. MoonGen can achieve this rate even if each packet is modified by a custom Lua script.

Another important feature of MoonGen is that it can also be used to receive traffic and this can be set in the same userscript used to generate the attack. This was really very useful because it allowed us in a simple way to obtain statistics both on the attack and also on the traffic that is analysed by the DUT and sent back, simulating a real case, where there is a machine that does detection and that does the forwarding of traffic to servers within the data center.

In the Figure [4.2] can be seen the MoonGen architecture, where we see that the inner core is represented by a LUA wrapper for DPDK that provides functions

for generating packets and also an API to configure hardware-related features such as timestamps and speed control. Here, the important thing for us is the userscript which is used to define the logic used for generating the traffic and which we will talk about later.

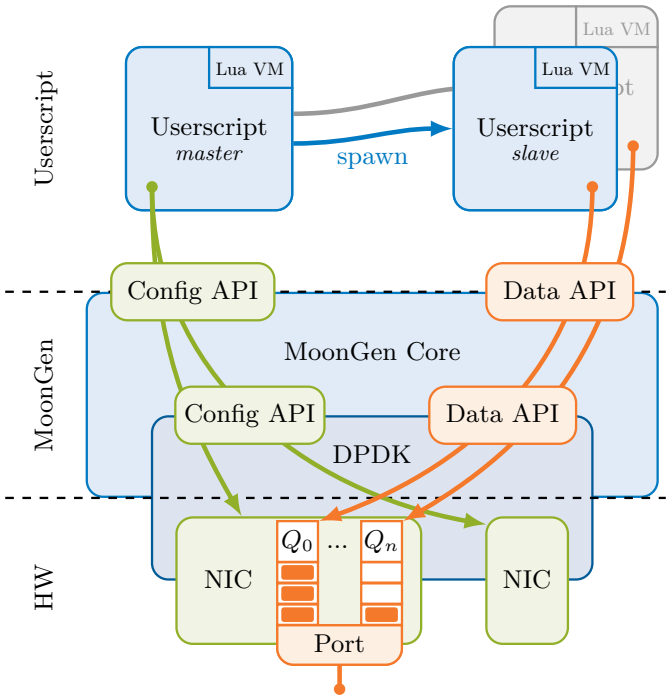

Figure 4.2: MoonGen architecture.

4.3.2 Pidstat

Pidstat [19] (PID Statistics) is the tool used in tests to obtain various statistics regarding the CPU consumption of a particular process. In particular the % user, that is the amount of CPU consumed on the user space side by LUCID and Polycube.

4.3.3 Docker stats

The docker stats [20] command returns a live data stream for running containers. To limit data to one or more specific containers, specify a list of container names or ids separated by a space. In the tests carried out it was used to obtain information on the resources consumed by the DeChainy docker which correspond to the sum of resources consumed by the various user-side components of DeChainy including the extractor and LUCID.

4.3.4 Perf stat

The perf stat [21] command is used to collect statistics from suitable counters. It was used to try to explain the decreasing trend of traffic forwarded and therefore received by the attacking machine.

4.4 Configurations

The testbed is composed by two physical machines connected with two direct links using dual-port Ethernet Controller 10-Gigabit X540-AT2 NICs. One machine operates as DUT (Device Under Test), i.e. the victim of the attack, on which the detection algorithm, coupled with Polycube or DeChainy, is executed and which forwards the packets between its two interfaces. The attacking machine represents both the attacker who generates an attack on the first interface and a server that receives traffic on the second interface in order to perform statistics. The two machines are configured as follows:

- *DUT/Victim*: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor with 4 cores (8 with hypertreading), 32 KB of L1 cache, 256 KB of L2 cache and 819 KB of L3 cache; 32 GB of DIMM DDR4 and Ubuntu 20.04.1 LTS. The version of the kernel used is 5.9.1-050901-generic, this being able to install all the frameworks and tools but also to take advantage of some operations (batch) developed in recent kernels;
- *Attacker*: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor with 4 cores (8 with hypertreading), 32 KB of L1 cache, 256 KB of L2 cache and 819 KB of L3 cache; 32 GB of DIMM DDR4 and Ubuntu 20.04.1 LTS. The version of the kernel used is 5.4.0-65-generic.

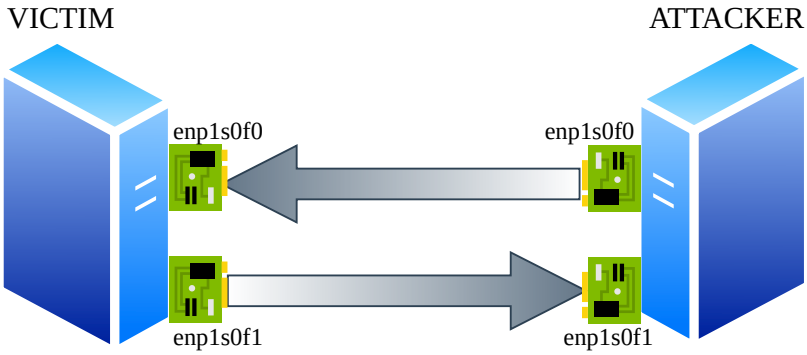

Figure 4.3: Architecture used to conduct the tests.

4.4.1 Baseline

In order to understand how much traffic analysis or network monitoring affects, two tests were conducted aimed at obtaining a baseline. This baseline represents the amount of traffic that the interface of the test machine, configured as in the following tests, is able to manage.

- The first test was done with Polycube, using the Helloworld service. In this way we can see how much traffic the interface is able to forward, if the packet is not even touched. In this case the traffic received by the attacking machine is 10.60 Mpps;

- The second test represents a somewhat more realistic baseline. In this case we used Linux's xdp_redirect_map program which does forward traffic to another interface using the bpf_redirect_map() helper. In addition to forwarding, both has a forwarded packet counter and MAC address swap. In this case the traffic received by the attacking machine is 5.59 Mpps.

4.4.2 Attacker machine

In the machine used as attacker, MoonGen has been installed, which allows you to use your own script to generate a custom attack. The script used comes from [22] which was later modified to obtain data on the traffic received on the second interface. The attack in question is a SYN Flood attack with the following parameters:

- The source IP network is 15.0.0.0/8 which means the attack uses 2^{24} different addresses;

- The range of ports used is 2000-2100;

- The rate was left at the maximum. On average, the attack has a throughput of 14.88 Mpps and with 64-byte packets, where 64 bytes is the length of the ethernet frame containing the SYN packet, including the CRC;

- The script generates packets starting from the source port and cycling on all IPs. This way we mimic the behavior of a real SYN Flood attack case, and in addition we are sure that for the entire duration of a test there are no repeated sessions.

Thanks to this attack we have put ourselves in a position to analyse traffic that always has different IP addresses for a given GET. To start the attack just run the custom script using MoonGen.

4.4.3 Victim machine

Here we see the victim machine configuration:

- The interfaces enp1s0f0 and enp1s0f1 have been configured with a single queue, which is mapped on a single core (the two queues share the same core). In this way, the obtained data are not affected by any hyper treading mechanisms and we can then obtain more data, allowing us to understand if it is worthwhile to scale; because maybe a single core/queue cannot handle all incoming packets when it is under attack. The Dynmon service eBPF code (it means the probe eBPF code) and the Helloworld service eBPF code, in the case of Polycube, and the probe eBPF code in the case of DeChainy, are executed in a single core. The user-level code is executed on multiple cores, managed by the OS;
- *pidstat* was used to monitor Polycube and LUCID's CPU. In particular for Polycube, to understand how much it consumes to extract data in userspace, given that, with pidstat, is not possible to know how much consume the eBPF program that runs in the kernel. This because when a packet arrives from outside, an interrupt is triggered, which will not be processed in the Polycube context and therefore we cannot obtain the information having the Polycube PID;
- *docker stats* was used to monitor DeChainy's CPU (which also includes the LUCID CPU).

Here we see the Data plane configuration, i.e. the eBPF code that deals with monitoring:

- **N_SESSION**: it is the number of max TCP session tracked by the eBPF code. This value starts from 256 and has been increased by powers of 2 until an upper limit due to Polycube or DeChainy is found;
- **N_PACKET_PER_SESSION**: it is the number of packets from the same TCP session. This value was set at 10. This is because in the tests carried out, the neural network used was trained using a time window of 10 seconds (10s) and a flow length of 10 packets (10p). In particular, the model used is 10s-10p-SYN2020-CNNLight, which was created just for SYN flood attacks. So to continue to have a high True Positive and obtain data as real as possible on the amount of traffic that the part that deals with monitoring and forwarding the data to LUCID can manage, the value of Packet Per Session has been imposed on 10. Indeed, even if this value had been greater, a maximum of 10 packets would still have reached the neural network.

Chapter 5

Evaluation: detection

This chapter presents a series of tests performed with Polycube and DeChainy together with LUCID, in order to evaluate their advantages and disadvantages and also find possible improvements. The tests done are intended to evaluate:

- What are the limits of Polycube and DeChainy coupled with LUCID (used sequentially) where they represent the data source on which LUCID can do his analysis;
- Based on the modification of the appropriate parameters in the eBPF code that performs the monitoring, see what changes in the level of traffic analysed, consumption of resources and so on;
- Analyse the various components of an IDS.

5.1 Tests with LUCID

5.1.1 LUCID configuration

Here we see the important parameters in the LUCID and also in the DeChainy configuration when it is used with LUCID:

- **Time Detection**: it represents the attack and detection duration. It was set to 60 seconds and it should be noted that, in some cases, it depends on the parameters such as configuration, the duration of the detection and therefore of the attack is greater (more details will be provided later). As a minimum detection and attack time, the value of 60 seconds has been chosen so that the *kernel* can apply some optimizations to the reception queues and stabilize the amount of traffic it can handle;

- **Time GET**: represents the time between two consecutive GETs made, after LUCID has produced its result. More specifically, it represents the time between two consecutive GETs that LUCID does to Polycube or the time between two GETs done by DeChainy to the method that takes care of everything (extraction and recall of LUCID). In practive, in the case of LUCID and Polycube, LUCID makes to the data provider a GET, waits until it receives the data, analyses the traffic, produces a result, waits for a Time GET and does another GET. Logically, in the case of DeChainy the same happens, only it is not LUCID who decides when to perform a new GET but DeChainy himself. In the initial tests, the Time GET was set at 1 second, then it was decided to remove it completely in order to go to maximum power, because we are in the case of hardware that only has to do traffic detection and forwarding and therefore it is useless for it to stand still even for only 1 second;
- **Time Window**: it is used to simulate the acquisition process of online systems. LUCID collects all the packets from the flow with capture time between t0, the capture time of the first packet, and time t0+t. This value was left at 10 seconds, the default value. It should be noted that, since the packets analysed in a GET are always part of a time window of less than 10 seconds, another time window could also be chosen.
- In the original LUCID code there was also a **Timeout GET**. This value has been removed precisely to measure the time required to obtain data when some parameters in the probe's data plane code change.

Now let's see how to configure the victim machine to be able to run the tests, both with Polycube and with DeChainy. To facilitate the execution of the tests, appropriate bash functions have been created that group several commands that are listed in the appendix [A].

For the interfaces configurations was used ethtool [23], in this way each interface will use a single queue which will be mapped to a single core of the machine. This choice was made because in this way we can obtain data without counting hyper threading [A].

In addition to ethtool an affinity [24] script was used, in this way, the two interfaces will use the same queue/core. This script must be used at the beginning of each test, after injecting the eBPF code (after injecting the probe code with Polycube or after starting DeChainy).

LUCID and Polycube

For these tests, Polycube was used as a data provider for LUCID. In particular, the Polycube services that have been used are:

- Helloworld of type XDP_DRV and in forward mode, connected between enp1s0f0 and enp1s0f1 [A];
- Dynmon of type XDP_DRV, connected to the enp1s0f0 interface. To create a Dynmon service a python script was used as an injector because the eBPF code must be properly formatted in JSON [A].

We use XDP_DRV which is a eBPF hooks that have to be supported by the NIC driver and with this we can have the highest dropping rate. In practice, the service is loaded as an XDP program in the so-called driver mode and that can discards the packets as soon as they arrive in the NIC driver, before delivering them to the main network components of the OS.

In order to perform the tests with LUCID and Polycube we need to follow these steps:

1. In the dataplane-test.json [A] file we put in the "code" section, the eBPF code of the probe, suitably formatted and with the value of the number of sessions chosen;
2. Start Polycube [A];
3. Create an instance of the Helloworld service of type XDP_DRV, connect it between the two interfaces and set it as "forward" action, in order to forward between the two interfaces [A];
4. Create a Dynmon service instance of type XDP_DRV and attach it to the first interface. To do this we need to use the dynmon_injector script [A];
5. Run the affinity script [A];
6. Run LUCID [A];
7. Launch the attack with MoonGen [A];
8. After a few seconds (to warm up the cache) from the first GET that returns data, start perf stat [A];
9. For the resources consumption on the user side of LUCID and Polycube, we need to give these commands. It should be noted that they must be given only after both the attack and LUCID have been started (which before performing the first GET has sleep in order to synchronize everything) [A].

In this series of tests, Polycube and LUCID work sequentially, this means that LUCID performs a GET, Polycube extracts the metrics and then responds to LUCID, which analyzes the traffic and returns a result, then another is done GET.

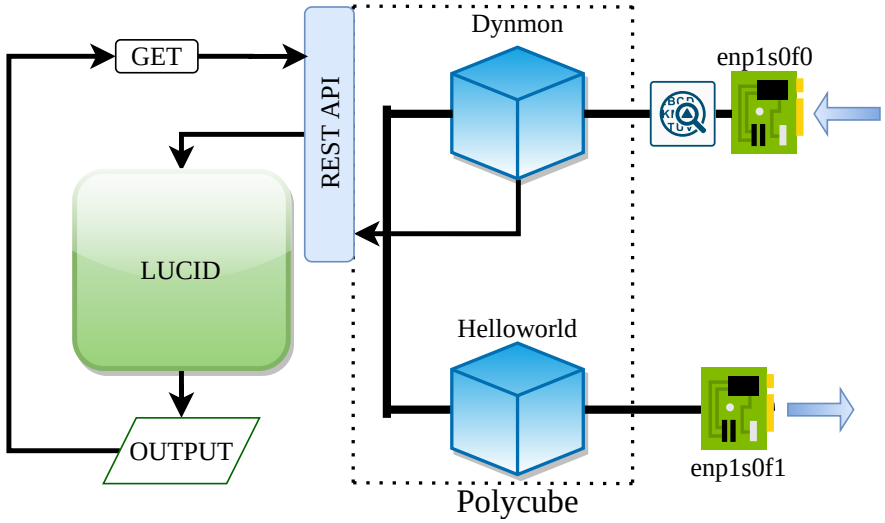

Figure 5.1: Connection of the two Polycube cubes to the two network interfaces with LUCID.

LUCID and DeChainy

In these tests, DeChainy was used as a data provider for LUCID. For ease of use, the docker version of DeChainy has been used. In order to perform the tests with DeChainy we need to follow these steps:

1. Modify the startup.json file appropriately (which for simplicity is located in the main directory of DeChainy) [A]. The startup.json represents the configuration file where to load both the dataplane and the controlplane and also specify the interface on which to place the probe and in the case of our tests also the interface to which to redirect the traffic;

2. Starting DeChainy [A]: DeChainy will start making empty GETs (that will not be counted) until it finds some data to extract from the maps which happens immediately after the start of the attack;

3. Run the affinity script [A];

4. Launch the attack with MoonGen [A];

5. As before, after a few seconds, perform perf stat [A];

6. As regards the consumption of resources, just execute this command which is used to save all the information in an appropriate file with the number of sessions chosen [A].

In this series of tests, as before, the DeChainy extractor and LUCID work sequentially, it means that a GET is done to a given method, the metrics are extracted, LUCID analyses the traffic and returns a result, then another GET is done.

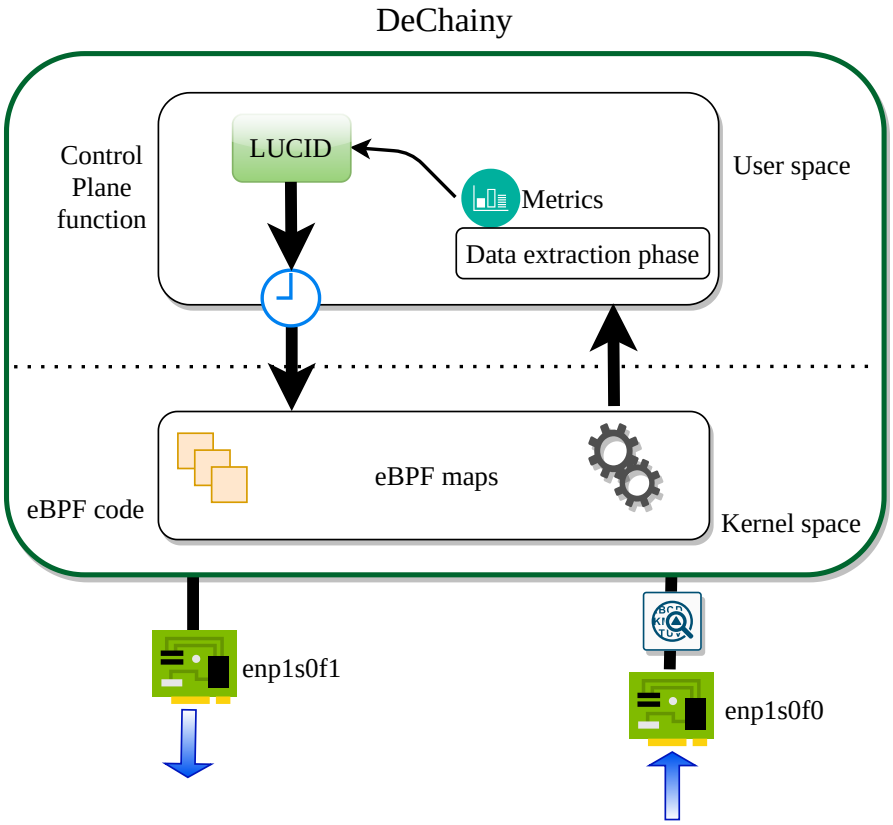

Figure 5.2: LUCID and DeChainy architecture used in tests.

5.1.2 Data to collect

Here is a list of data that is recorded at the end of each test (of which only the most important will be explained later in appropriate graphs):

- *N Session* used;
- *Real Time Detection (Time of the attack)*: as mentioned before, the time detection is fixed at 60 seconds, but because we are not in a perfect and synchronic way, due to delays in providing the data, the real time detection exceeds 60 seconds. This way we can do an integer of GETs and not truncate the data we want to get;

- *Total Time Work*: this value represents the actual working time, not counting any delays due to the writing of the information to be obtained from the tests. The value is almost identical to Real Time Detection;
- *Total Packets Analysed, Packets Analysed per Second, Packets Analysed per GET*: where by Analysed Packets we mean the number of packets that pass from kernel level to user level, before moving on to the detection algorithm;
- *Unique Total True Positive, Unique True Positive per GET, Unique True Positive per Second*: these values represent the LUCID output to which our Custom Policy is applied. Since we have 100% accuracy, this value is equal to the number of packets with unique IP addresses that we were able to detect;
- *LUCID processing Time per GET*: includes prediction time, padding time and normalization time and in the case of tests with Polycube also the data translation time;
- *Polycube/Dechainy processing Time per GET*: it is the time it takes the data provider to get the data from kernel level maps to metrics in the right format at the user level. In the case of Polycube, this time also includes the data serialization and deserialization time.
- *Number of GETs* that were performed during the test;
- *Average CPU* value of LUCID, Polycube and DeChainy at the user level, with *pidstat* and also with *docker stats*;
- *Mpps inbound* on the attacking machine, as the victim does packet forwarding on the second interface;
- *perf stat output*: some counters can be useful to explain the forwarded traffic trend;

5.2 Packets analysed

The quantity of packets captured by the eBPF probe and arriving at the user level (after the extraction phase) with both Polycube and DeChainy, used sequentially with LUCID, is now shown in various ways. Note that due to the nature of the attack and the model used for the neural network, LUCID always has 100% accuracy. This has resulted in that the amount of captured packets arriving at the user level (Packets Analysed), matches and nearly equal (due to rare overlaps) with the amount of unique malicious addresses found by LUCID (Unique True Positive) with our Custom Policy.

This thesis work mainly focused on the analysis of the various components of an IDS, in particular on the monitoring of the network and not on collecting data relating to the accuracy of the detection algorithm. Therefore, it was preferred, also for practical reasons in the visualization, to explain only the graphs concerning the quantity of packets analysed. One important thing to note is that, even if the tests had been done with all cores (4 in the victim machine), the values would not have been multiplied by the number of cores.

5.2.1 Total Packets Analysed

In Figure [5.3] is shown the total number of packets analysed trend by the eBPF probe at the end of each test, both in the case of Polycube and DeChainy, with respect to the increase in the number of sessions set in the probe code (N Session). As we can see, in the case of Polycube, the last two values of N Session are missing, this is because we realized that with the value of 131072, the Polycube extractor time was higher than the 60 second limit imposed in all tests, value beyond which the test is automatically concluded. Finally, with DeChainy we chose to stop with a number of sessions of 524288 because we saw that the value of analysed packets was stabilizing. This figure only wants to show the final value of the packets analysed at the end of each test. As said before, this value is almost equal to the number of unique IP addresses found at the end of each test, almost equal because some overlap of attacking IP addresses can occur.

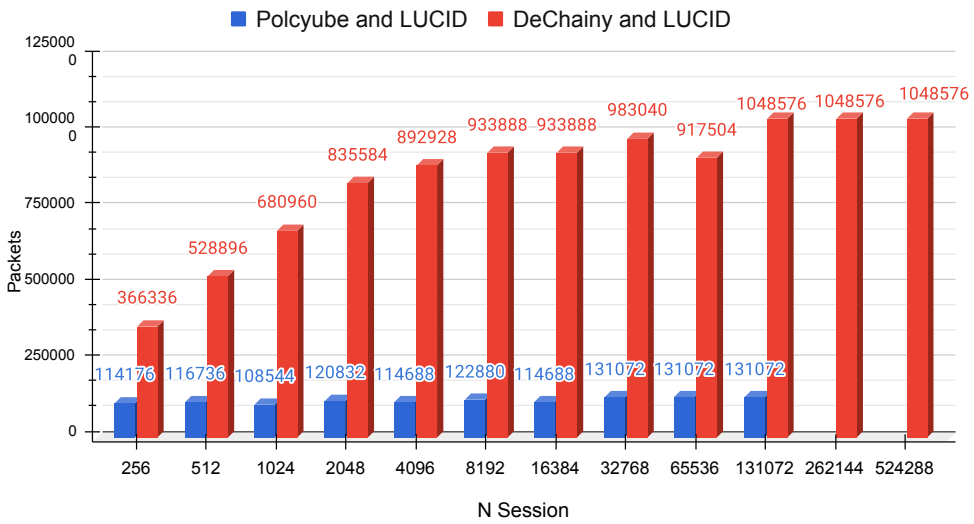

Figure 5.3: Total Packets Analysed by Polycube and DeChainy coupled with LUCID with respect to the number of sessions.

5.2.2 Total Packets Analysed per GET

The Figure [5.4] displays the packets value that are analysed in each GET performed by LUCID, both in the tests carried out with Polycube and with DeChainy. As it can be seen, the value of packets analysed in each GET is equal to the value of the number of sessions set in the eBPF probe. This is because as mentioned before, the attack used in the tests has such a high number of addresses 2^{24} and a high enough port value (100) that in 100% of cases, for each session, only one packet is captured in a single GET.

An important thing to note is that, in the tests carried out we only used malicious traffic, since we want to carry out a performance analysis on the monitoring and not on the accuracy of the chosen detection algorithm and this, in addition to giving us the opportunity to focus on the attack, it resulted in 100% accuracy in every test performed. It means that the value of packets scanned for GET matches the value of malicious unique IP addresses found for GET. This value therefore hypothetically represents the **maximum** value of malicious IP addresses that are found at each GET and by doing so we can focus only on the monitoring/detection part and not on the attack response part, perhaps using a DDoS mitigator. Trivially, there is an exponential trend and therefore by increasing the value of the number of sessions in the probe, the number of packets that LUCID can analyse in a GET also increases.

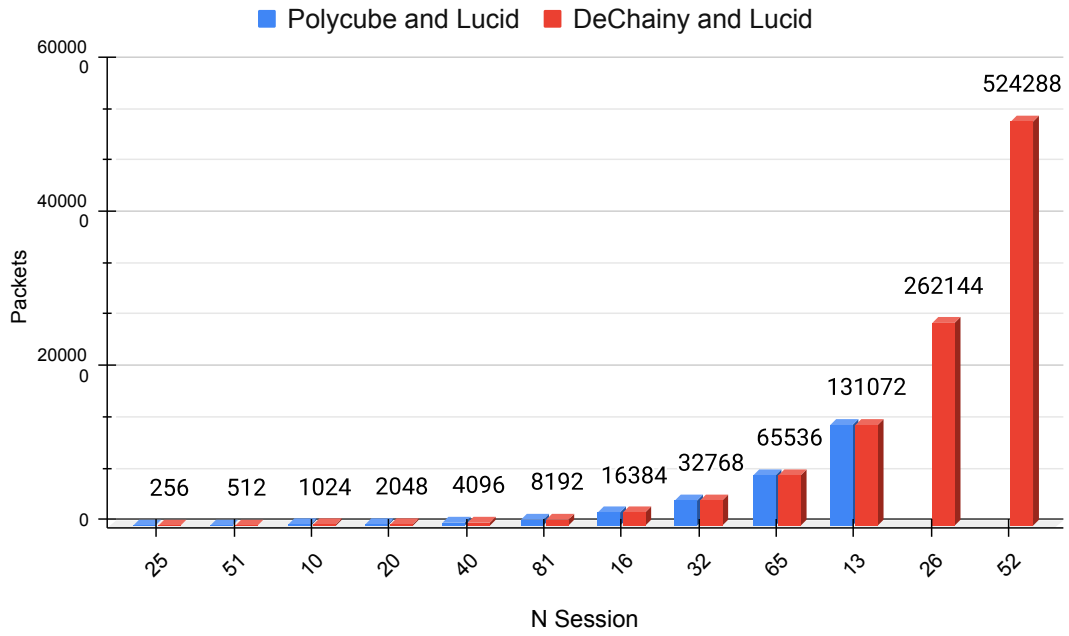

Figure 5.4: Average Packets Analysed per GET by Polycube and DeChainy coupled with LUCID with respect to the number of sessions.

5.2.3 Total Packets Analysed per Second

Since the total real time of each test may vary slightly due to different times of extracting the metrics from the eBPF maps, Figure [5.5] shows how many packets are analysed every second with respect to the value of the number of sessions chosen, both in the case of Polycube and than in the case of DeChainy.

This is one of the **most** important values from the tests carried out, indeed, thanks to this value we can actually see how many packets are analysed in the unit of time regardless of how long the test lasted.

- In the case of Polycube, we see that the packets value analysed per second oscillates between a minimum of 1874 and a maximum of 1978. The Polcyube values are lower than DeChainy, and this is due to the fact that Polycube's map extractor is slower than DeChainy's;

- In the case of DeChainy, the value oscillates between a minimum of 6136 and a maximum of 16186. The important thing to note is that, even in DeChainy's case, the value of the packets that are analysed every second is really low compared to the amount of packets passing through the interface. As mentioned earlier, the attack has a value of 14.88 Mpps, the two interfaces of the victim machine are mapped to a single queue which is mapped to a single core of the machine. The traffic baseline supported by the interfaces with this configuration is given by the value forwarded using xdp_redirect_map, about 5.59 Mpps; while in the case of DeChainy and LUCID, used sequentially, the value of traffic forwarded to the attacking machine fluctuates between 2.05 and 1.59 Mpps. This means that, even in the best of cases, about 99% of packets do not even get analysed by the detection algorithm, since the eBPF probe, once it has full maps, directly discards the packet. For the same reason, further tests were carried out.

Finally, as we can see, there is an increasing trend as the number of sessions set in the eBPF probe increases up to a certain value, beyond which the number of packets analysed per second begins to stabilize, undergoing some variations.

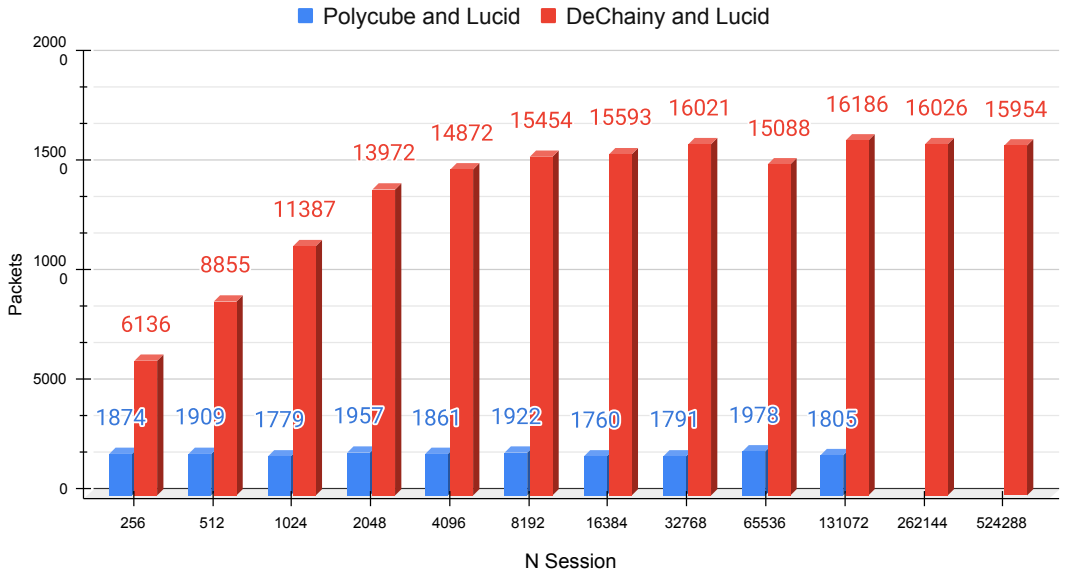

Figure 5.5: Average Packets Analysed per second by Polycube and DeChainy coupled with LUCID with respect to the number of sessions.

5.3 Processing time

The processing times of LUCID, Polycube and DeChainy are now shown. For the sake of convenience in the visualization, two different graphs have been created for each case.

5.3.1 LUCID processing Time

The Figure [5.6] shows the LUCID processing time trend for each GET with Polycube, while Figure [5.7] shows the LUCID processing time trend for each GET with DeChainy, in both case, as the number of sessions set in the eBPF probe increases.

LUCID processing time is the sum of padding, data normalization and prediction time. As we can see, the prediction time is really low, indeed most of LUCID's processing time is due to padding and data normalization. These are two necessary operations that are performed to deliver the data in the correct format to the neural network. In practice, each attribute value is normalized on a scale [0, 1] and the samples are filled with padding (all zero values) so that each sample is of fixed length N, that is, up to the maximum value imposed by the value of Packet Per Session in the eBPF probe. This is because, having fixed length samples is a requirement for a CNN to be able to learn about an entire set of samples. In our

case, there being only one packet in each session and having set 10 as the maximum packet value for each session, it means that 9 packets will be padding.

So, even if the prediction time is low, padding and normalization times are much greater, meaning that LUCID works for a not negligible time and that, indeed, LUCID occupies a value of about 9.66 seconds in the case of Polycube with number of sessions 131072 and a value of 30.3 seconds in the case of DeChainy with number of sessions 524288 for each GET. This means that for this period of time, Polycube and DeChainy remain waiting without doing anything and especially in the case of DeChainy this leads to a huge drop in performance since for the same number of sessions, the processing time of DeChainy is less (much less) than the entire LUCID processing time. This does not detract from the fact that LUCID can represent an excellent detection algorithm, which, by exploiting CNNs, can provide better results with low resource consumption providing better results. Furthermore it works perfectly with Polycube and DeChainy, and as will be discussed later, this relationship could be improved by running LUCID asynchronously or in other ways as well, for example by improving padding and normalization operations. Regarding the LUCID processing time trend as the number of sessions varies, there is simply an increasing trend due to the fact that more data are processed for each GET.

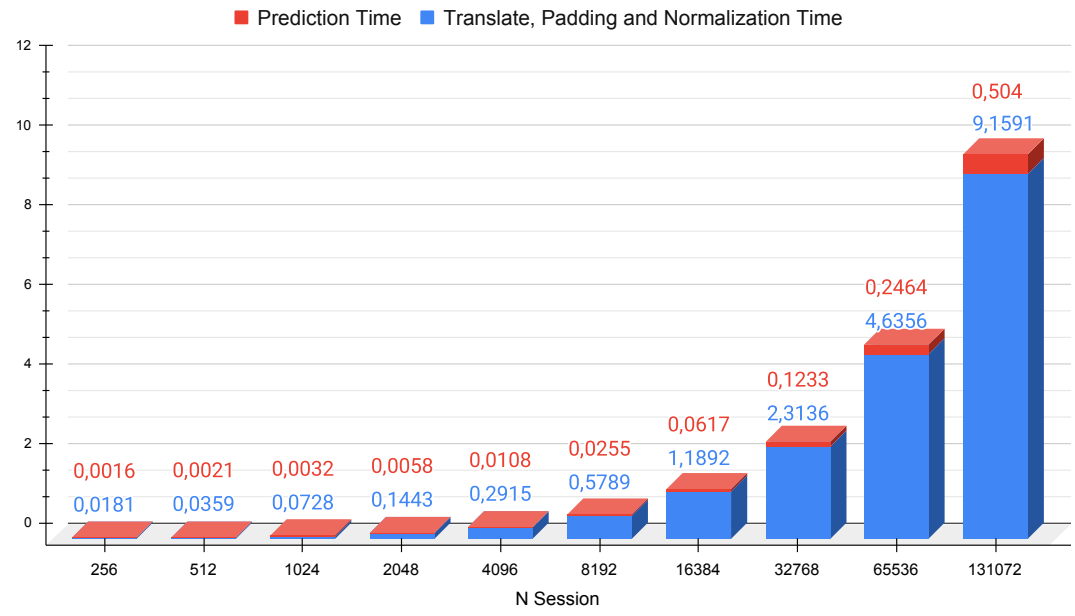

Figure 5.6: Average LUCID processing Time per GET coupled with Polycube with respect to the number of sessions.

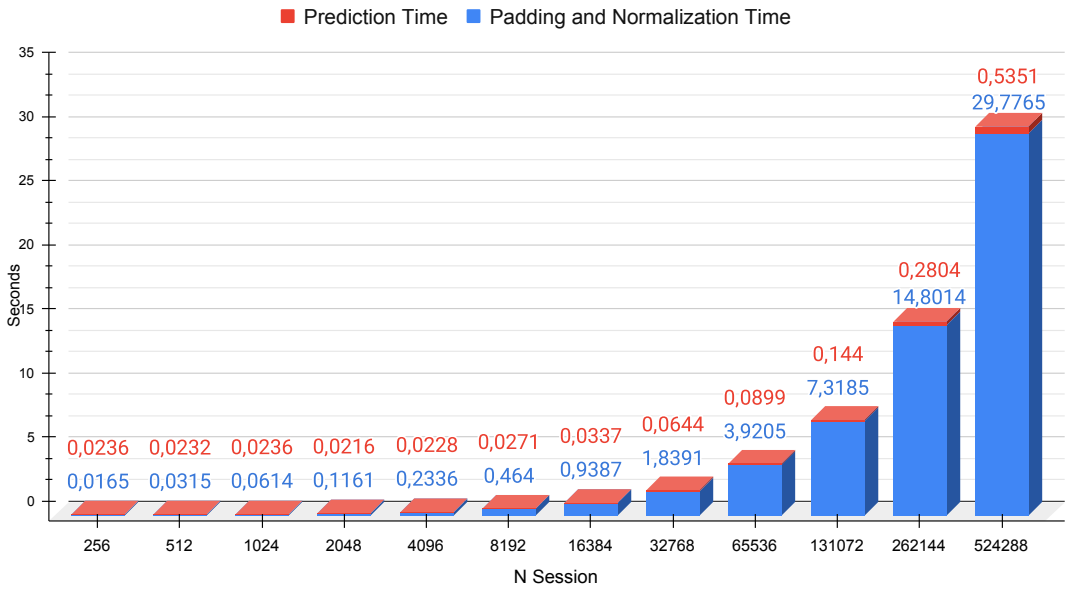

Figure 5.7: Average LUCID processing Time per GET coupled with DeChainy with respect to the number of sessions.

Even if the version of LUCID used is the same for all the tests carried out, with the same number of sessions, it can be seen that with Polycube and with DeChainy, the prediction times are slightly different. Although the prediction time has very little impact on the total processing time of LUCID, investigating more deeply, two possible explanations have been found:

1. The prediction time may vary based on the resources consumed by the machine at that particular time, therefore it depends on the consumption of Polycube, DeChainy and also on any active processes on the machine;

2. As can be seen from Figure [5.8], the first prediction time value is much higher than the others. This may be due to the fact that the first time the prediction is made, Tensorflow needs a certain period of time for some initialization processes and that it subsequently uses caching mechanisms and other optimizations. Thanks to the this point, it is possible to understand the difference between LUCID's prediction times in cases where it is used with Polycube and DeChainy, indeed for example with a value of the number of sessions of 32768, with Polycube 4 GETs are carried out while with DeChainy 30 GETs are carried out [A];

3. On the other hand, with low number of sessions values, it can be seen that LUCID's prediction times with Polycube are lower than when used with

DeChainy, this could be due to the fact that LUCID with Polycube is a separate script, which runs so in a main thread, while in the case of DeChainy, LUCID is executed in a secondary thread.

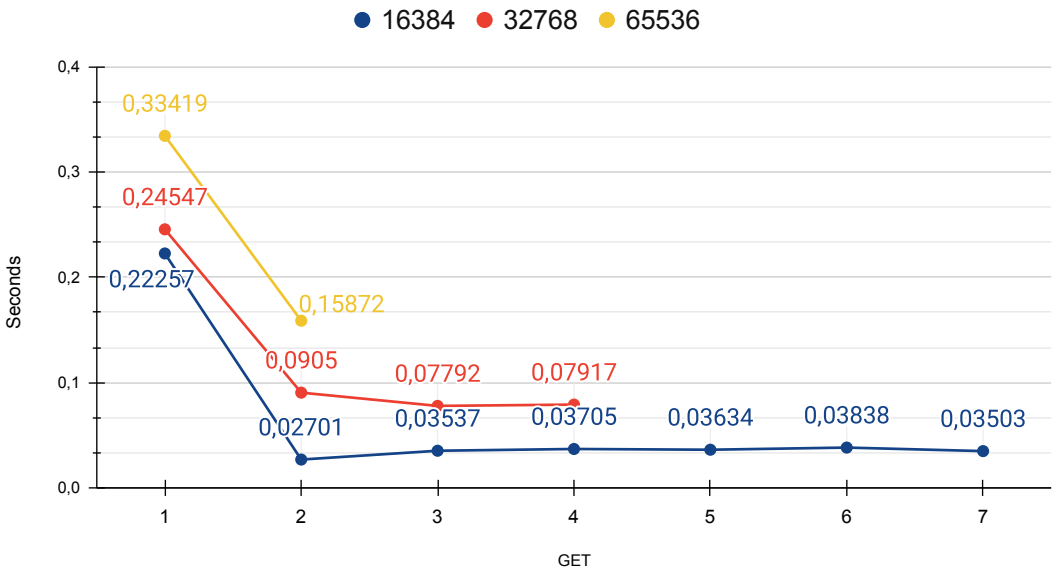

Figure 5.8: LUCID prediction times coupled with Polycube, with value of the number of sessions: 16384, 32768, 65536.

5.3.2 Polycube and DeChainy processing Time

For the sake of practicality in the visualization, two different figures have been created. In Figure [5.9] it can be seen the Polycube processing time trend for each GET that is carried out by LUCID, as the number of sessions increases. With Polycube processing time, we mean the sum of data extraction time from the eBPF maps, the data serialization and deserialization times, given that LUCID and Polycube communicate through a REST interface, although, these last two times are insignificant compared to the extraction time.

As we can see, there is an (almost) exponential trend and for high values in the number of sessions, the processing times become truly enormous. This is caused by how the extraction of maps is carried out in Dynmon, which, in addition to allowing the injection of eBPF code at runtime for the creation of custom probes, also allows you to pass data from the kernel level to the user.

The problem in the extractor, written in C++ as a programming language, is given by the fact that, for each packet from which a certain amount of information

is to be extracted, there is a conditional construct that serves to identify what type of data must be extracted (int, float, struct, etc...) and which function to call. This conditional construct is done to each packet, so there is no memory of the previous packet. To solve this problem, the parsing phase could only be done the first time, since, all subsequent times, the information to be extracted from the packets does not change. In short, it would be necessary to adopt a caching system and save the sequence of functions to be called in a suitable data structure. In addition to this, another possible solution is to eliminate the conversion of the data extracted from the maps to JSON, since, being sent to the Polycube REST interface, they are subsequently converted. During the thesis work, it was preferred to continue with the tests instead of improving the Dynmon extractor, to obtain as much data as possible, this also because in addition to Polycube, Dechainy was also used which, from preliminary tests carried out, we knew it had a very efficient extractor written in Python.

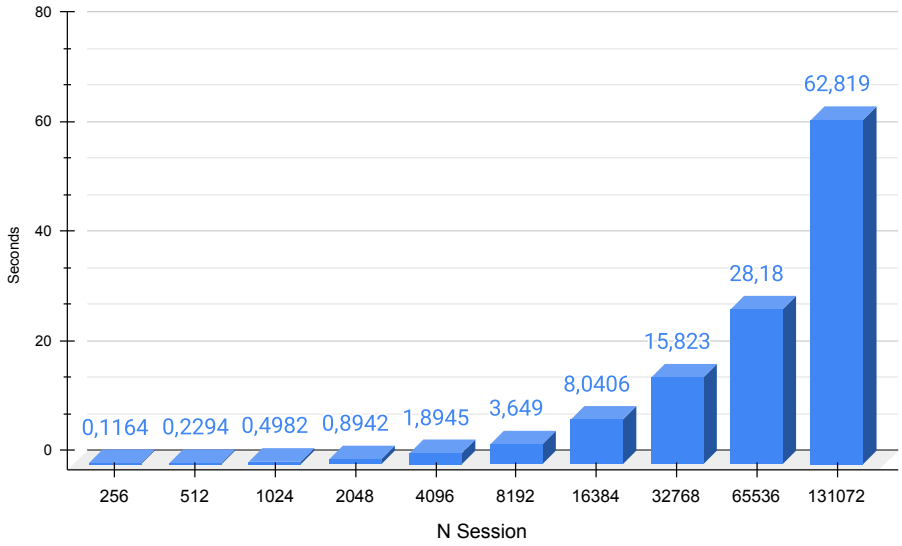

Figure 5.9: Average Polycube processing Time per GET with respect to the number of sessions.

In Figure [5.10] we see the DeChainy extractor processing time trend for each GET as the number of sessions used varies. With DeChainy processing time, we only mean data extraction time from the eBPF maps. Indeed, LUCID is not a separate entity from DeChainy but represents a part of the Control plane. As we can see, here too, there is an (almost) exponential trend, except that the maximum value is 2.55 seconds with 524288 as the value of the number of sessions. The DeChainy extractor turns out to be much faster, but, even if 2.55 seconds may

seem short, in this time frame, the attack used in the tests performed about 38 million requests. As we will see later, the time required by the extractor also depends on the fact that immediately after the extraction, it prepares the data by creating suitable flows that will later be used by LUCID, operations that can also be removed from the extractor.

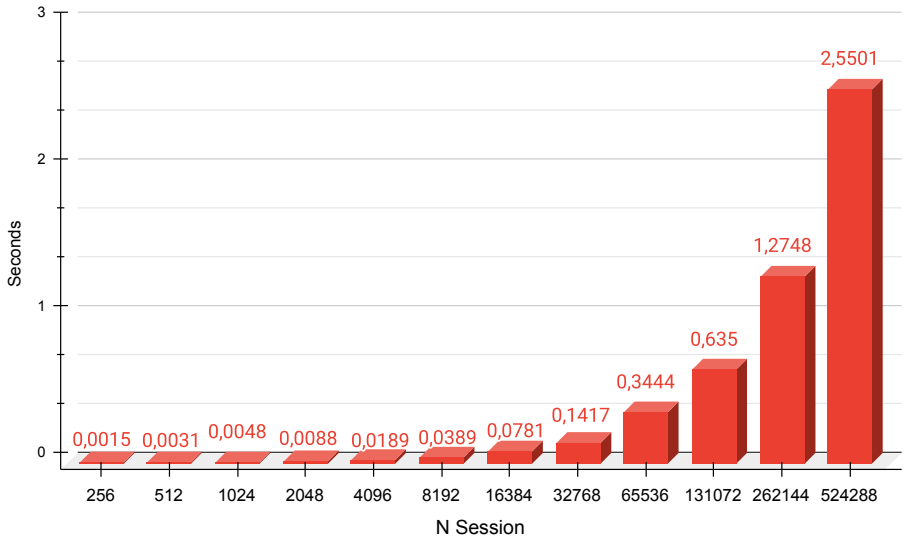

Figure 5.10: Average DeChainy processing Time per GET with respect to the number of sessions.

5.4 CPU consumption with detection

This section shows the CPU consumption on the user side of Polycube, Dechainy and LUCID as the number of sessions set in the eBPF probe varies. Obviously it must be taken into account that these values may vary due to various optimizations on the Operating System side and that with a different attack, the CPU consumed, especially by LUCID, would have been different due to the padding and normalization operations and consequently (in the case of Polycube) also to the translation of the data. It should also be noted that the CPU consumed by the eBPF code is not counted since the eBPF code (both monitoring and forwarding) has been set on a core and being the attack quite high, the CPU consumed on the **kernel** side is always at 100%, in addition, with pidstat it is possible to see only the CPU consumed by a process whose PID is provided, PID that we do not have a priori. With these data we just want to understand how much the extraction and detection itself consume.

5.4.1 Polycube

The Figure [5.11] shows the user side CPU consumed trend of Polycube (extractor and other operations to communicate with LUCID) and LUCID. As we can see, the sum of the two percentages is about 100%, so LUCID and Polycube together use (approximately) a core of the victim machine. LUCID and Polycube are used sequentially and the one displayed is an average over the entire test. Indeed, there are times when Polycube waits for LUCID and vice versa. Let's analyse two example cases in detail:

- In the case of the number of sessions set to 256, the CPU value of LUCID is 17.31%, and the CPU value of Polycube is about 80%, this is because even if it is true that the work done by the extractor of Polycube for a single GET is low, the number of GETs performed by LUCID is greater in the unit of time and therefore Polycube works continuously on new requests and new data;

- In the case of the number of sessions equal to 131072, LUCID performs a single GET and then waits for the data, waiting for about 62 seconds during which, the Polycube map extractor is working to bring the metrics to LUCID, then for all the wait LUCID will have a CPU value of about 0% and then reach an average of 16.57% (obviously with peaks of 100%).

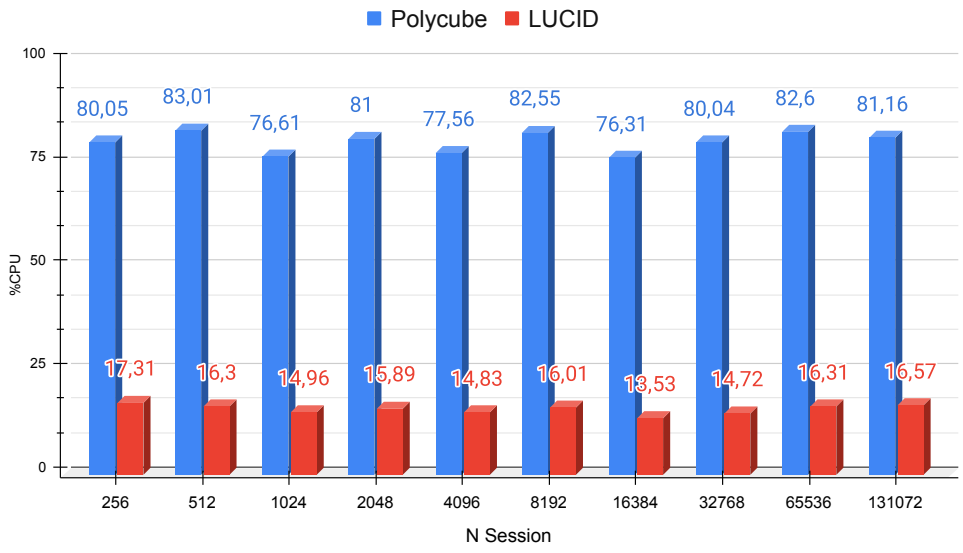

Figure 5.11: Average %CPU of Polycube and LUCID with respect to the number of sessions.

5.4.2 DeChainy

In the tests carried out with DeChainy, the docker used represents the set of the LUCID detection algorithm and the map extractor (and also the part that manages everything), so the following Figure [5.12] shows the user side CPU consumed trend of the two programs mentioned above. The % is intended on 4 cores so 100% means 1 core of the machine and this data was obtained using the docker stats command [A]. From the Figure it can be seen that there is no growing trend with the increase in the number of sessions and that at a certain point there is a certain stability. This is due to the fact that for small extractions a greater number of GETs are made in the unit of time that must be managed (same speech for large extractions).

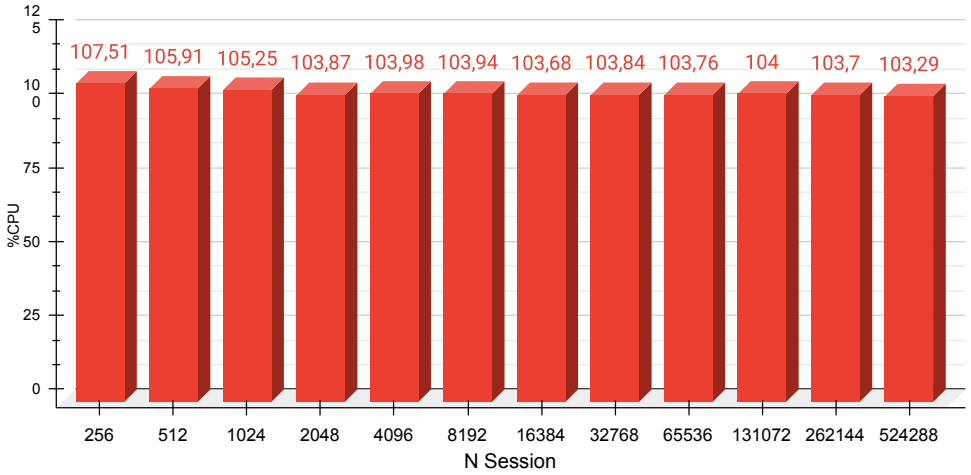

Figure 5.12: Average %CPU of DeChainy (with LUCID) with respect to the number of sessions.

The CPU consumed by Polycube and LUCID and also consumed by DeChainy (extractor and LUCID) are approximately the same. As we can see, most of the consumed resources are due to extraction and not to LUCID.

5.5 Traffic forwarded

This section, shows the traffic forwarded by the victim machine, which, in addition to doing detection, also takes care of the forwarding. In this way we simulate a machine that is positioned between the outside and inside of a data center to be protected. The traffic that arrives at the attacking machine is made up of traffic that arrives at the eBPF probe and that has traveled through it in its entirety (i.e. traffic that will later be extracted) or that has traveled it up to the full map control and therefore has been immediately dropped, i.e. in this case redirected to the second interface.

The case with Polycube and Dechainy are analysed separately to facilitate the visualization and explanation of the collected data. For both cases, the first point that catches the eye is that the amount of traffic received has dropped drastically compared with our two baselines: xdp_redirect_map where the packet is edited and Polycube's Helloworld where only forwarding is done. Note that these values may also vary slightly depending on the actual machine load.

Mpps Polycube

The Figure [5.13] shows the trend of forwarded traffic in the case of Polycube. As we can see, there is a decreasing trend as the number of sessions increases.

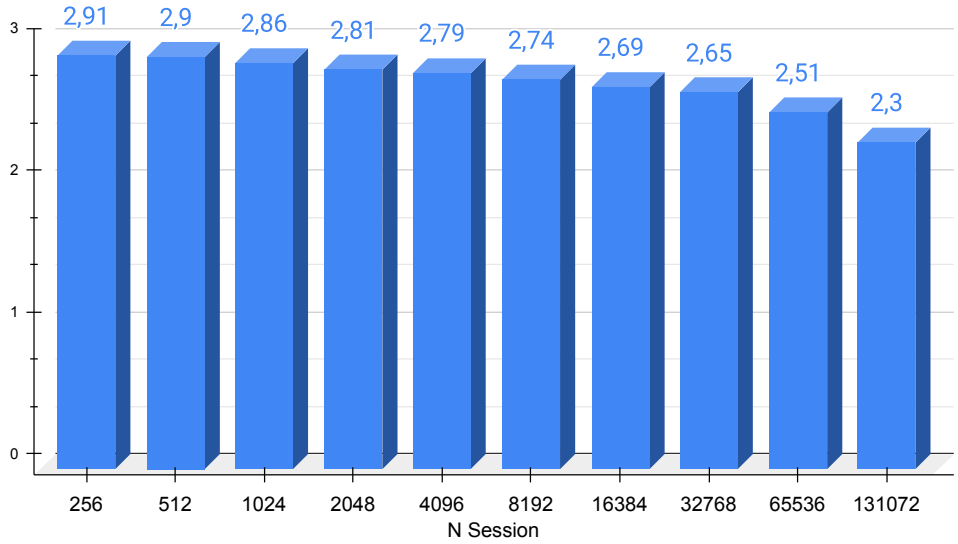

Figure 5.13: Traffic forwarded by the victim machine with Polycube coupled with LUCID and received by the attacker machine with respect to the number of sessions.

Mpps DeChainy

The Figure [5.14] shows the trend of forwarded traffic in the case of DeChainy. With low values of the number of sessions, the traffic forwarded is not really descending, starting from 2048 we see the decreasing trend as the number of sessions increases.

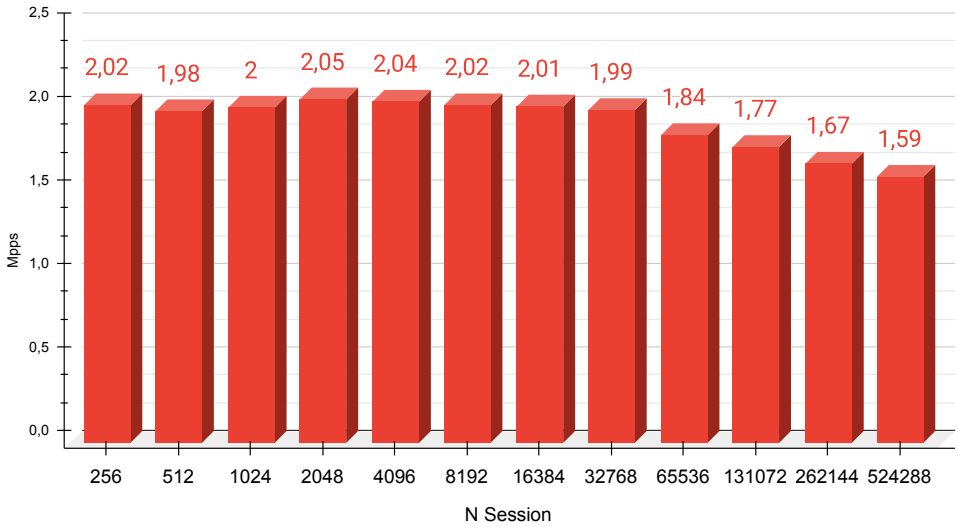

Figure 5.14: Traffic forwarded by the victim machine with DeChainy coupled with LUCID and received by the attacker machine with respect to the number of sessions.

Explanation of forwarded traffic

The inversely proportional trend to the increase in the number of sessions is due to the increasing quantity of packets that are analysed at each request (GET) made by LUCID; since increasing the number of sessions, increases the number of times the eBPF probe will be in the case of the map not fully and therefore the packet will not be discarded (redirected) immediately, but the probe will do all the necessary checks and then push the packet in the appropriate map (slow path).

These results are in contrast with the value of Packets Analysed per Second, which could have led us to think that increasing the number of sessions in the eBPF probe, increases the performance at all points. In fact, if the number of sessions increases, the number of packets analysed increases, the traffic forwarded decreases, so the solution to an effective detection is not so easy.

Although still to be thoroughly investigated, another promising and possible explanation for the decreasing trend in forwarded traffic could be the following: cache problem. For this reason perf stat was used and as can be seen in the

appendix [A], various counters were used and the monitoring time of perf stat was set to 40 seconds, not to get too close to the end of the test and to get the data as close as possible to the steady state trend. The command is launched a few seconds after the start of the detection in order to warm up the caches.

Of all the counters used, the most important are (where LLC stands for Last Level Cache):

- LLC-load indicates the number of times a load request occurs in level 3 cache;
- LLC-load-miss indicates the number of times in which, against of a load request in the level 3 cache, there has been a miss, that is, the data was not found. Excludes hardware and software prefetching.

The increasing trend of LLC-loads and LLC-load-misses as the value of the number of sessions increases, i.e. the size of the SESSIONS_TRACKED_DDOS map in the eBPF probe, indicate that, the larger the map, the more accesses there are to both the cache L3 than to the main memory, and therefore it also increases the response time when the map lookup is performed. This can therefore explain the decreasing trend of forwarded traffic as the number of sessions increases.

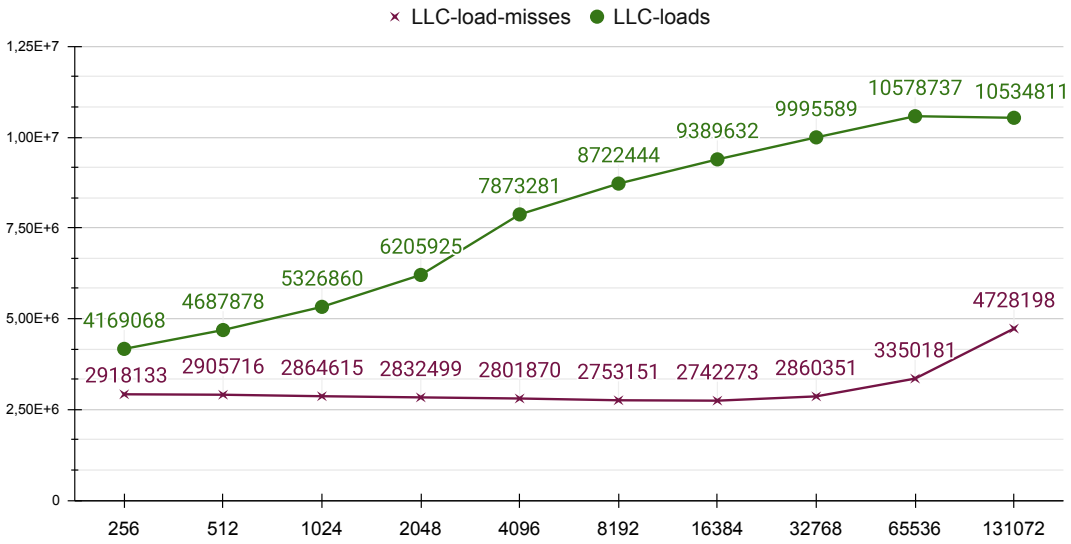

Figure 5.15: LLC-loads and LLC-load-misses of Polycube used with LUCID with respect to the number of sessions.

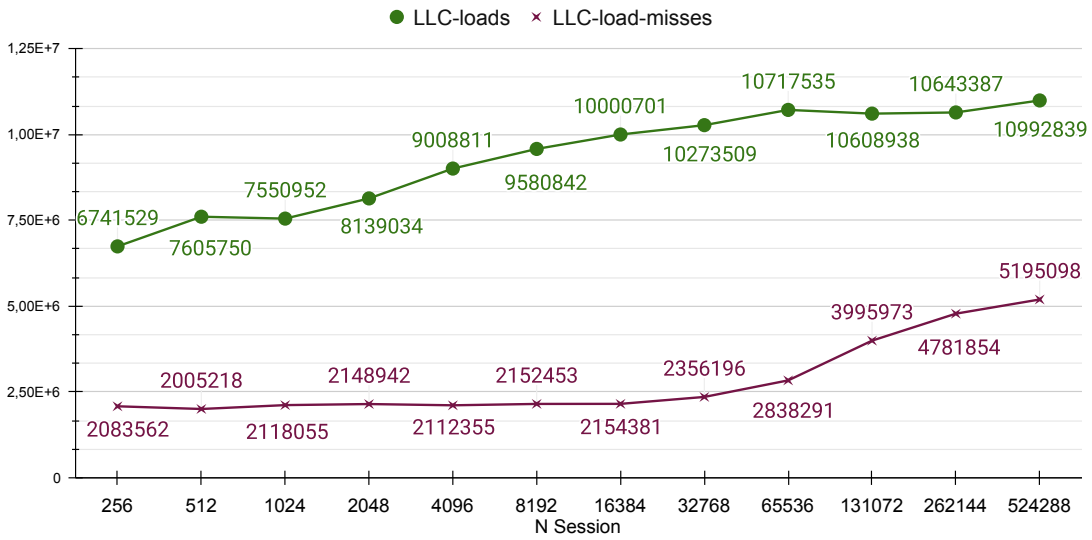

Figure 5.16: LLC-loads and LLC-load-misses of DeChainy used with LUCID with respect to the number of sessions.

If this observation turns out to be correct, since the level 3 cache is shared among all cores, it might be a good idea to switch to a distributed version. In fact, using N nodes instead of just one, we will multiply by N the value of the level 3 cache (and also of the main memory). Although, there would still be other factors to consider such as sharing the L3 cache with other processes that do not deal with detection.

A note about the difference in forwarded traffic between Polycube and DeChainy, it mainly depends on two things:

1. The mechanism with which Polycube implements the redirect is different from that of DeChainy: the packet follows a different path;

2. In the case of DeChainy, the number of GETs performed by LUCID is much larger than the number of GETs performed in the case of Polycube. For this reason, more traffic is analysed per unit of time and at the same time less is forwarded. This second point does not represent a real disadvantage given that in any case the traffic analysed and which is subsequently detected is greater. A proof in favor of this point is given by the results of the traffic forwarded by DeChainy in the case of extraction only, indeed, here the number of GETs per unit of time increases dramatically, and the forwarded traffic decreases, especially for larger values of the number of sessions.

Chapter 6

Evaluation: extraction

After having seen all the results of the tests carried out in the detection case, it was decided to perform another type of test with DeChainy only used as an eBPF probe and as a data extractor. In this way we have seen how much traffic it is able to analyse, that is how much traffic is brought from the kernel level to the user level. These tests are very important since, in the case of DeChainy and LUCID, it is the padding and normalization operations of LUCID that slow down DeChainy, who must wait for the detection results instead of being able to do another extraction. So, we want to know how much traffic can DeChainy manages used alone without detection, in such a way as to have a baseline and not be strictly tied to a single detection algorithm used sequentially.

To be able to perform this type of test, just follow the steps specified in the previous chapter and modify the Control plane appropriately, not calling LUCID after extraction. As for the previous tests, the Time GET is set to 0 seconds while the Time Detection to 60 seconds.

In this case, the packets extraction is intended as extraction from the queue without the creation of flow with relative features, indeed this part can possibly be done asynchronously and re-enter as LUCID's work since those given features are made ad hoc for LUCID. In brief, by extraction we mean only the pop from the QUEUE map, PACKET_BUFFER_DDOS map.

6.1 Packets analysed

This section shows the packets captured and extracted trend by DeChainy as the number of sessions set in the eBPF probe varies. As we can see, the value of analysed packets is literally greater. Indeed, as mentioned earlier, DeChainy was slowed down by having to wait for LUCID since his extraction time is shorter.

6.1.1 Packets analysed per second

Figure [6.1] shows the packets value extracted trend per second by DeChainy. There is no real growing trend as the number of sessions increases and in general the average is around 424,000 packets per second. This represents a huge increase compared to the tests carried out with detection, where the average was about 14,000 packets. The value of packets analysed per second by DeChainy opens up new avenues. One of this, is to abandon the use of DeChainy and LUCID sequentially and then start using LUCID asynchronously.

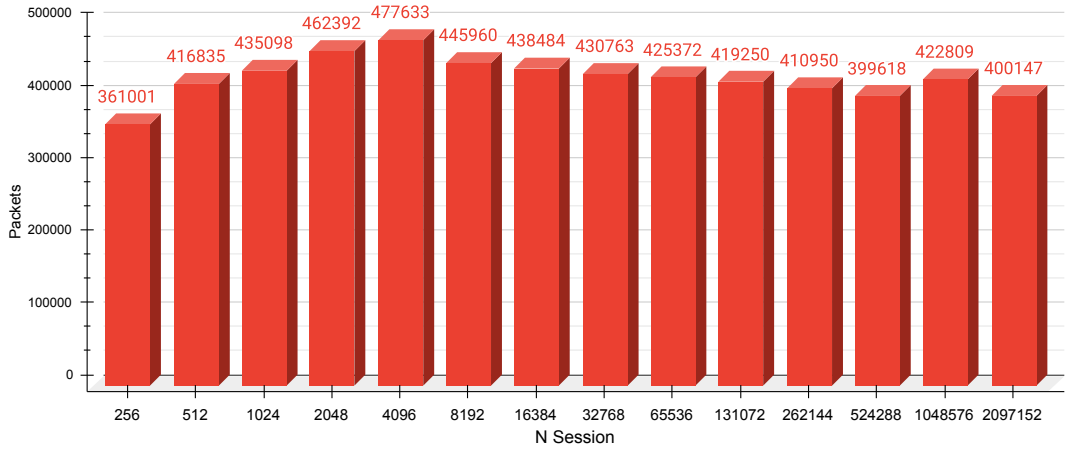

Figure 6.1: Total Packets Analysed per Second by DeChainy with respect to the number of sessions.

6.2 Processing Time

In Figure [6.2] we can see the DeChainy processing time which in this case represents only the extraction time of the packets from the eBPF map. Compared to the tests in which DeChainy was used with LUCID, the DeChainy processing time has decreased, because, as said before, in this case the extraction function has been changed which now only deals with retrieving all packets from the PACKET_BUFFER_DDOS queue and not creating the flows. In a simple while loop, a pop is done from the queue, this is because unfortunately Queue does not support the batch operation.

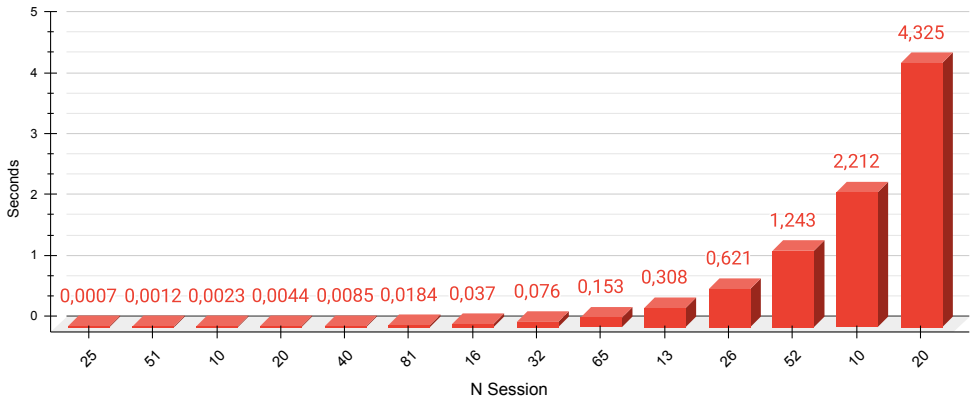

Figure 6.2: Processing Time per GET of DeChainy with respect to the number of sessions.

6.3 CPU consumption

Figure [6.3] shows the trend of the DeChainy user level CPU as the number of sessions varies. Due to the fact that the number of extractions per unit of time has increased, the resources consumed have also increased. The average value is around 111, which means it takes up around one core.

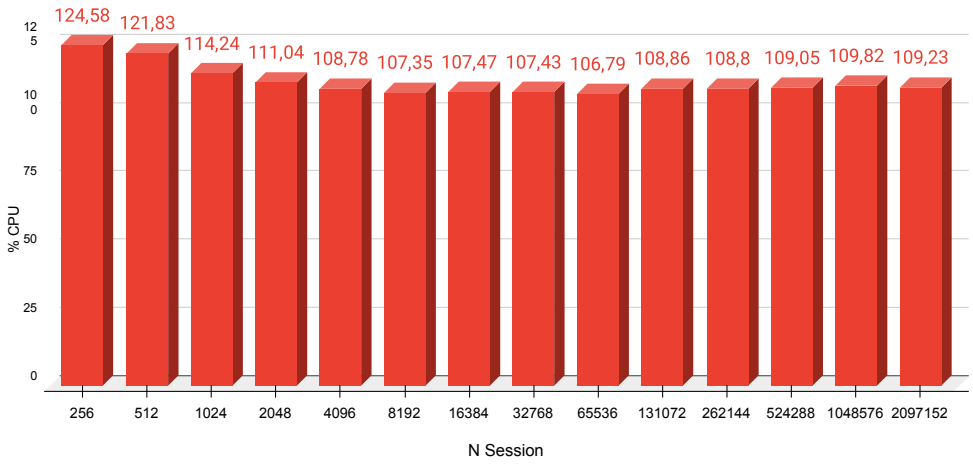

Figure 6.3: Average %CPU of DeChainy with respect to number of sessions.

6.4 Traffic forwarded

Figure [6.4] shows the traffic forwarded trend by DeChainy. Leaving aside the first values, it can be seen that also in this case, as in the case of DeChainy used together with LUCID, there is a decreasing trend as the number of sessions set in the eBPF probe increases. Since DeChainy no longer has to wait for LUCID, it performs a greater number of extractions per unit of time (GET), so the eBPF probe code will be found for less time in the case of a full map and a packet will not be redirected immediately, but extra instructions will be made (slow path). For this reason, with the same number of sessions, there is a decrease in forwarded traffic, not very visible for the first values of the number of sessions, since the processing time between two GETs could be more or less similar in the two cases (detection and extraction), but more evident for high values, where the LUCID processing time in the first case become more evident. Thanks to the baseline of forwarded traffic, to the figure that shows the forwarded traffic in the detection case (always with DeChainy) and to the latter figure, it is possible to understand how much the monitoring, the extraction of data in the unit of time and the relationship affect between extracted/analysed traffic and forwarded traffic. Indeed, the higher the extracted/analysed traffic, the lower the forwarded traffic and this can be seen especially for larger values of the number of sessions.

From Figure [6.4] we can also see that, DeChainy extractor is excellent, indeed, for example, taking the case with the number of sessions of 5424288, the forwarded traffic is 1.44 Mpps, while the value of packets extracted per second by the extractor is of about 400 thousand. This means that about 27.78% of the traffic could be analysed by a detection algorithm asynchronously and 72.22% of the traffic is not extracted. Even if 72% of the traffic is not analysed, compared to the case of DeChainy and LUCID used sequentially these values are much better.

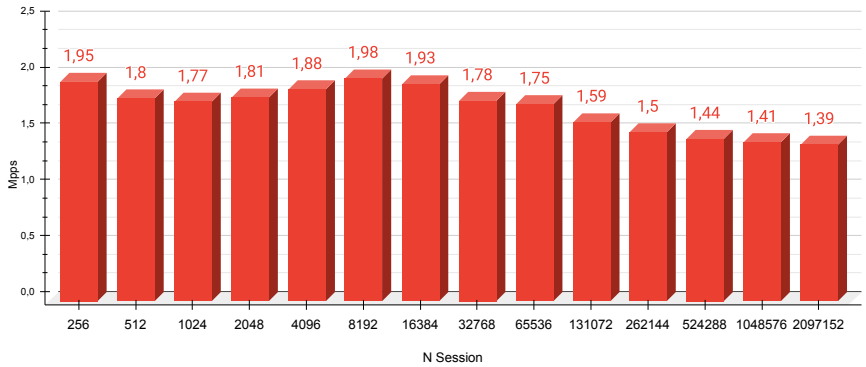

Figure 6.4: Traffic forwarded by DeChainy with respect to the number of sessions.

Finally, as in the case of tests with detection, in Figure [6.5] we can see the trend of the LLC-loads and LLC-load-misses values as the value of the number of sessions varies. These values are a further confirmation of what was said in the previous chapter about the possibile cache problem.

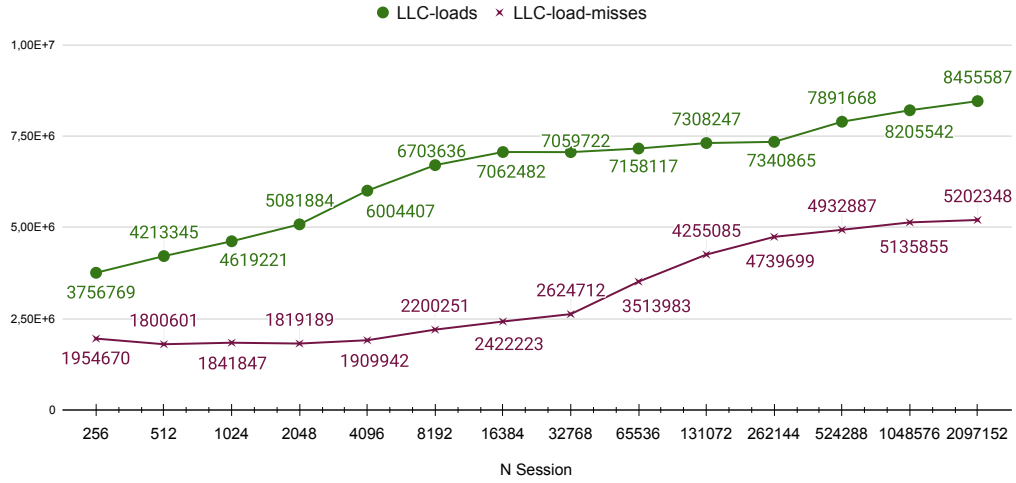

Figure 6.5: LLC-loads and LLC-load-misses of DeChainy with respect to the number of sessions.

Chapter 7

Conclusions

The work described in this thesis represents an analysis of the various components that are used to create an IDS, starting from monitoring, passing through extraction and arriving at detection. A series of actions to be evaluated in order to improve the current state-of-the-art for IDS systems are also considered, e.g. which parameters can be more or less useful to obtain an efficient network monitoring (for example the value of the number of analysed sessions), which parts should be improved from an engineering point of view and which components are really important.

One of the most interesting results that can be drawn from this thesis work is that data extraction is much more CPU-demanding than the detection itself. In fact, it has been seen that, even if eBPF/XDP offers an interesting support for traffic monitoring, monitoring is the part that mostly impacts the amount of traffic that can be handled by a machine. In the considered case, in fact, the limit on the number of analysed flows is not determined by LUCID but is determined by the component that deals with monitoring. Finally, operations carried out to provide the data, in a certain format, to the component that deals with the prediction represent a very critical object.

From the tests carried out, various conclusions can be drawn regarding both Polycube, DeChainy and LUCID taken individually but also coupled together to form a real IDS:

- As far as Polycube is concerned, it can be seen that the main technological limit is represented by the map extractor, an extractor that, perhaps with another paradigm, could certainly lead to better results. However, Polycube represents an excellent framework, which has a huge amount of services to offer, allows you to easily create service chains and in our case, thanks to Dynmon, it allows you to create custom eBPF probes.
- DeChainy, even if not yet complete, represents a brilliant framework that allows you to easily write the Control plane in Python and quickly test custom

eBPF probes. The map extractor proved to be excellent both in terms of performance and ease of changing data extraction policies.

- As for LUCID, the results show that this algorithm represents an excellent solution in the detection of DDoS attacks, which can be easily modified with appropriate Custom Policies to meet any type of end-user need and which finally integrates perfectly with Polycube and DeChainy. As for the LUCID times, it should be noted that the prediction phase has very low times and that most of the LUCID processing time is due to the padding and normalization of the data when the traffic value starts to be high, operations that can certainly be improved from an engineering point of view.

7.1 Future works

The choice of Open Source components, their modularity and their ease of integration, leave wide space for future work, both from an implementation and research point of view. Some of the future works that can be based on this thesis work can be the following.

A first direction consists in continuing the analysis of the components discussed, using for example other types of attacks, that include both malicious and benign traffic or in any case other attack patterns in order to obtain other data on detection performance.

Since the DeChainy extractor takes much less time with respect to the total LUCID processing time, a possible further direction could be to perform the detection asynchronously with respect to the extraction. For instance, immediately after extracting data from eBPF maps, we could invoke LUCID in an asynchronous thread/process and immediately perform another extraction. In this way, the DeChainy extractor would not remain on hold and could continue to extract traffic which would later be analysed by LUCID. All the operations necessary for the detection, starting from the creation of the flows, the padding, the normalization and finally the prediction could be done asynchronously. Obviously, this solution could also bring both advantages and disadvantages such as excessive consumption of resources and therefore in this case it would be convenient to continue to use hardware dedicated only to detection.

Finally, the most challenging direction could be to move into a fully distributed world, using Polycube or DeChainy as a data provider and LUCID as a detection algorithm. An example of architecture is the one shown in Figure [7.1]. This direction follows the research activities that are currently carried out by several companies such as Cloudflare, leader in the field of cybersecurity and also Facebook, with its Katran load balancer.

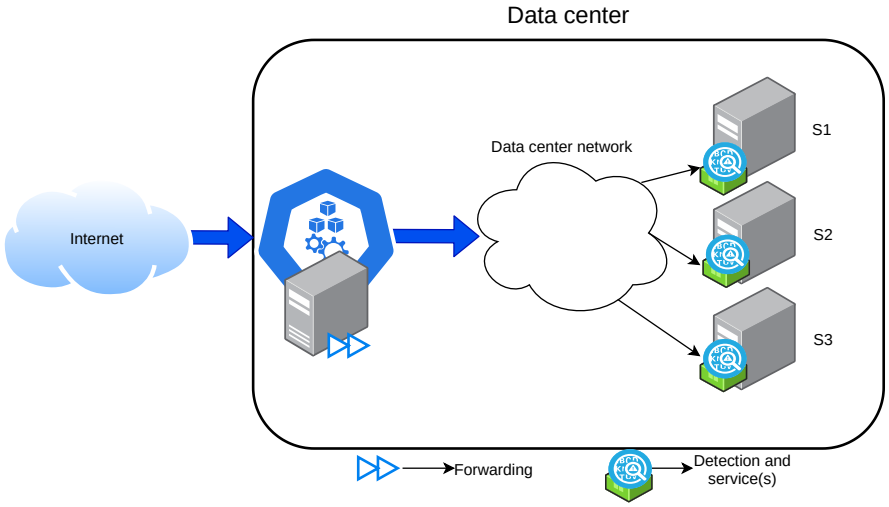

Figure 7.1: Possible architecture for distributed IDS.

The possible advantages of a fully distributed IDS are the following:

- Scalability;
- Ability to reduce false positives;
- Reduce resource consumption on devices that perform detection;
- (Hopefully) Increase the number of attackers per unit of time;
- (It depends) Do not use servers dedicated only to IDS, but integrate this job on normal servers that process also other workloads.

Having a distributed version of a detection algorithm does not only brings advantages but also opens the way to possible problems:

- How does a network device know that someone is attacking it, if it does not have full vision but only sees some traffic?
- How to make a decision effectively?
- Is there a convenient and effective way to make N nodes on which detection is made collaborate?

Furthermore, in case of a distributed approach, we also need to know who should decide whether a given source is an attacker or not. In the literature there are various solutions with pros and cons:

- Master node that decides: having a complete view of the network thanks to the fact that all nodes send it information on possible attackers. The solution

taken may actually be the right one since the information in its possession is a lot, but the master node can be a bottleneck;

- All nodes decide on their own on the basis of information that is exchanged periodically, perhaps using a *distributed hash table*;
- A node is elected as master only for a certain period of time: this is the policy implemented by the *RAFT* protocol, which is the basis of the `etcd` distributed database in *Kubernetes*.

Appendix A

Commands and configurations

Here it is shown the various commands used to set up and run the tests and some important pieces of code.

Listing A.1: Configuration JSON file used in LUCID-Polycube tests.

```
{
    "ingress-path": {
        "name": "Packets feature extractor LUCID-Polycube",
        "code": "...",
        "metric-configs": [
            {
                "name": "PACKET_BUFFER_DDOS",
                "map-name": "PACKET_BUFFER_DDOS",
                "extraction-options": {
                    "swap-on-read": true,
                    "empty-on-read": true
                }
            },
            {
                "name": "SESSIONS_TRACKED_DDOS",
                "map-name": "SESSIONS_TRACKED_DDOS",
                "extraction-options": {
                    "swap-on-read": true,
                    "empty-on-read": true
                }
            }
        ]
    }
}
```

Listing A.2: How to run Polycube.

```
    sudo polycubed

    # OR

    docker run -p 9000:9000 -d --name polycube-docker --privileged --
    network host -v /lib/modules:/lib/modules:ro -v /usr/src:/usr/src:
    ro -v /etc/localtime:/etc/localtime:ro polycubenetwork/polycube /
    bin/bash -c 'polycubed'
```

Listing A.3: How to create a Helloworld service and attach it to the two interfaces in forward mode.

```
function add-helloworld() {
    polycubectl helloworld add hw0 type=XDP_DRV
    polycubectl hw0 ports add port1 peer=enp1s0f0
    polycubectl hw0 ports add port2 peer=enp1s0f1
    polycubectl hw0 set action=forward
}
```

Listing A.4: How to create a Dynmon service and attach it to the an interface

```
./dynmon_injector.py lucid1 -m XDP_DRV -d enp1s0f0 ../dataplane-test.
    json
```

Listing A.5: How to run LUCID.

```
function run-lucid-polycube() {
    cd ~
    cd lucid-polycube
    conda activate test1
    python3 cnn-ddo-detection-keras.py --predict polycube --model
    trained-models/10s-10p-SYN2020-CNNLight.h5 --dataset_type SYN2020
    --victim_net <victim-net> --session_number "$1"
}
```

- –predict <type>: indicates in which mode LUCID is used. If "polycube" is put, LUCID will use Polycube as the data provider.
- –model <path>: indicates the file that contains the model that will be used by the neural network.
- –dataset_type <type>: indicates the type of dataset to use. Possible values are IDS2012, IDS2017, IDS2018, SYN2020.
- –victim_net <victim-net>: Indicates the victim network.

- –session_number "$1": this is the only parameter you need to input. Represents the value of the number of sessions chosen. In this way, the information we want to obtain from the test will be printed in a file with as "lucid-polycube-SN-value".

Listing A.6: Information on the consumption of resources on the user side of LUCID and Polycube.

```
function lucid-cpu-info() {
    cd ~
    PIDL=$(ps -u test1 | grep -E 'python3' | grep -v '?' | awk '{
    print $1}')
    LC_NUMERIC=C.UTF-8 pidstat -u -t -p $PIDL 1 >> lucid-polycube/cpu
    -info/lucid-SN-"$1".pidstat
}

function polycube-cpu-info() {
    cd ~
    LC_NUMERIC=C.UTF-8 pidstat -u -t -p $(pidof polycubed) 1 >> lucid
    -polycube/cpu-info/polycube-SN-"$1".pidstat
}
```

Listing A.7: Configuration file for DeChainy: startup.json.

```
{
    "probes": [
      {
        "plugin": "adaptmon",
        "name": <probe name>,
        "mode": <mode>,
        "interface": <interface_1>,
        "redirect": <interface_2>,
        "time_window": value,
        "ingress": "...",
        "cp_function": "...",
        "files": {
            "model": "..."
          }
    }
  ]
}
```

1. "name": is the name of the probe.

2. "mode": is the type of the probe. It was set as XDP_DRV to have the highest packet drop rate.

3. "interface" and "redirect": respectively represent where the probe is positioned and to which interface to redirect traffic.

4. "ingress": the eBPF code of the probe must be entered, suitably formatted with the formatter script.

5. "cp_function": "cp" stands for Control Plane. It is the Python code properly formatted with the formatter script. In this code, both the extraction of the data taken from the kernel level maps and the LUCID part (padding, normalization and prediction) is done.

6. "time_window": indicates how often (in seconds) the function that takes care of passing the results from kernel level to user level will be called. In the tests carried out, this section was not actually used but the time between two GETs was manually set in an appropriate DeChainy configuration file (*utility* file).

7. "model": represents the model used to train LUCID's neural network. The model can be passed by putting the path to the correct file or it can be put as Base64 using a suitable formatting script.

Listing A.8: Run DeChainy docker with libraries for machine learning using the startup json in the root folder and the code of the local version.

```
function dechainy-docker-run() {
    sudo docker run --name dechainy-docker --rm --privileged --
    network host -v /lib/modules:/lib/modules:ro -v /etc/localtime:/
    etc/localtime:ro -v /usr/src:/usr/src:ro -v /home/test1/dechainy:/
    app/:ro s41m0n/dechainy:ml-cpu
}
```

Listing A.9: How to get information about the CPU consumed by DeChainy using docker stats.

```
function dechainy-docker-stats() {
    cd ~
    sudo docker stats dechainy-docker >> CUSTOM_PATH/DeChainy-SN-"$1
    ".stats
}
```

Listing A.10: How to start the attack.

```
function start-attack() {
    sudo ./build/MoonGen ./custom-attacks/l3-tcp-syn-flood-rx.lua
}
```

Listing A.11: How to perform perf with suitable counters.

```
function perf-stat-info(){
    sudo perf stat -e cache-references,cache-misses,L1-icache-load-
    misses,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores,
    l2_rqsts.all_demand_miss,l2_rqsts.code_rd_hit,l2_rqsts.
    code_rd_miss,l2_rqsts.demand_data_rd_hit,l2_rqsts.miss,l2_rqsts.
    demand_data_rd_miss,l2_rqsts.all_demand_references,LLC-loads,LLC-
    load-misses,LLC-stores,LLC-store-misses -C 0 -r 40 sleep 1
}
```

Listing A.12: Example of how a packet is analyzed by the probe.

```
    case IPPROTO_TCP: {
      // 1
      struct tcphdr *tcp = data + sizeof(struct eth_hdr) +
    ip_header_len;
      if ((void *) tcp + sizeof(*tcp) > data_end) {
        return RX_OK;
      }

      // 2
      struct session_key key = get_key(ip->saddr, ip->daddr, tcp->
    source, tcp->dest, ip->protocol);
      uint64_t *value = SESSIONS_TRACKED_DDOS.lookup_or_try_init(&key
    , &zero);
      if(!value) {
        break;
      }
      *value += 1;

      // 3
      if (*value > N_PACKET_PER_SESSION) {
        return RX_OK;
      }

      // 4
      struct features new_features = {.id=key, .timestamp=
    pcn_get_time_epoch(), .ipFlagsFrag=bpf_ntohs(ip->frag_off),
        .tcpWin=bpf_ntohs(tcp->window),
        .tcpFlags=(tcp->cwr << 7) | (tcp->ece << 6) | (tcp->urg << 5)
     | (tcp->ack << 4)
                  | (tcp->psh << 3)| (tcp->rst << 2) | (tcp->syn << 1)
    | tcp->fin};

      // 4
      PACKET_BUFFER_DDOS.push(&new_features, 0);
      break;
    }
```

Let's just take the case of TCP:

1. Level 4 is parsed and then a packet size check is performed.
2. Check if the session of the captured package has already been traced in the past, otherwise it is initialized, all this thanks to the lookup_or_try_init method.
3. Check whether the maximum number of packets that can be captured has been reached for this session.
4. Extract all the necessary info and fill the struct features.
5. Extracted features are added to the PACKET_BUFFER_DDOS map.

Listing A.13: LUCID output after prediction in human readable format

```
['15.0.0.5' '2000' '192.168.1.2' '80' 'TCP' 'True']
['15.0.0.6' '2000' '192.168.1.2' '80' 'TCP' 'True']
['15.0.0.7' '2000' '192.168.1.2' '80' 'TCP' 'True']
['15.0.0.8' '2000' '192.168.1.2' '80' 'TCP' 'True']
```

N Session	Polycube	DeChainy	N Session	Polycube	DeChainy
256	446	1431	16384	7	57
512	228	1033	32768	4	30
1024	106	665	65536	2	14
2048	59	408	131072	1	8
4096	28	218	262144		4
8192	15	114	524288		2

Table A.1: Number of GETs made by Polycube and DeChainy with LUCID.

Listing A.14: Configure interfaces with ethtool

```
sudo ethtool -L enp1s0f0 combined 1
sudo ethtool -L enp1s0f1 combined 1
```

Listing A.15: Configure affinity

```
sudo ./set_irq_affinity.sh <core> enp1s0f0 enp1s0f1
```

Pay attention to which core the eBPF code fixes, as it may happen that for optimization reasons the LUCID code is fixed on the same core as the eBPF code and therefore the performance is lower. For completeness, in the tests carried out with Polycube, the core value was different from 0, since it was noticed that LUCID (maybe due to miniconda) was fixed on core 0.

Bibliography

[1] Cloudflare. *What is a DDoS Attack?* URL: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/. (accessed: 02:2021) (cit. on p. 5).

[2] Cloudflare. *SYN Flood Attack*. URL: https://www.cloudflare.com/learning/ddos/syn-flood-ddos-attack/. (accessed: 02:2021) (cit. on p. 6).

[3] Cloudflare. *Network-layer DDoS attack trends for Q4 2020*. URL: https://blog.cloudflare.com/network-layer-ddos-attack-trends-for-q4-2020/. (accessed: 02:2021) (cit. on p. 6).

[4] Hongyu Liu and Bo Lang. «Machine learning and deep learning methods for intrusion detection systems: A survey». In: *applied sciences* 9.20 (2019), p. 4396 (cit. on p. 8).

[5] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. «Intrusion detection system: A comprehensive review». In: *Journal of Network and Computer Applications* 36.1 (2013), pp. 16–24 (cit. on p. 8).

[6] Hanan Hindy, David Brosset, Ethan Bayne, Amar Seeam, Christos Tachtatzis, Robert Atkinson, and Xavier Bellekens. «A taxonomy and survey of intrusion detection system design techniques, network threats and datasets». In: *arXiv preprint arXiv:1806.03517* (2018) (cit. on p. 12).

[7] Zilong Han, Xiaofeng Wang, Fei Wang, and Yongjun Wang. «Collaborative detection of DDoS attacks based on chord protocol». In: *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*. IEEE. 2012, pp. 1–4 (cit. on p. 12).

[8] Tommy Chin, Xenia Mountrouidou, Xiangyang Li, and Kaiqi Xiong. «An SDN-supported collaborative approach for DDoS flooding detection and containment». In: *MILCOM 2015-2015 IEEE Military Communications Conference*. IEEE. 2015, pp. 659–664 (cit. on p. 12).

[9] The Cilium Authors. *BPF and XDP Reference Guide*. URL: https://docs.cilium.io/en/v1.9/bpf/. (accessed: 02:2021) (cit. on pp. 14, 20).

[10] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. «Creating complex network services with ebpf: Experience and lessons learned». In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, pp. 1–8 (cit. on pp. 14, 18).

[11] The Linux Manual Authors. *BPF-HELPERS*. URL: `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`. (accessed: 02:2021) (cit. on p. 15).

[12] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, Jianwen Pi, and Aasif Shaikh. «A Service-Agnostic Software Framework for Fast and Efficient in-Kernel Network Services». In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–9 (cit. on p. 21).

[13] Simone Magnani. *DeChainy*. URL: `https://github.com/s41m0n/dechainy`. (accessed: 03:2021) (cit. on p. 22).

[14] Roberto Doriguzzi-Corin, Stuart Millar, Sandra Scott-Hayward, Jesus Martinez-del-Rincon, and Domenico Siracusa. «LUCID: A practical, lightweight deep learning solution for DDoS attack detection». In: *IEEE Transactions on Network and Service Management* 17.2 (2020), pp. 876–889 (cit. on p. 23).

[15] The Polycube Authors. *Helloworld Service*. URL: `https://github.com/polycube-network/polycube/tree/master/src/services/pcn-helloworld`. (accessed: 02:2021) (cit. on p. 25).

[16] The Polycube Authors. *Dynmon Service*. URL: `https://github.com/polycube-network/polycube/tree/master/src/services/pcn-dynmon`. (accessed: 02:2021) (cit. on p. 26).

[17] Simone Magnani. *ddos probe*. URL: `https://github.com/s41m0n/dechainy/blob/doc/improvements/examples/src/ddos_analyzer/ebpf.c`. (accessed: 03:2021) (cit. on p. 28).

[18] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. «Moongen: A scriptable high-speed packet generator». In: *Proceedings of the 2015 Internet Measurement Conference*. 2015, pp. 275–287 (cit. on p. 31).

[19] The pidstat Authors. *pidstat*. URL: `https://man7.org/linux/man-pages/man1/pidstat.1.html`. (accessed: 02:2021) (cit. on p. 32).

[20] The Docker authors. *docker stats*. URL: `https://docs.docker.com/engine/reference/commandline/stats/`. (accessed: 03:2021) (cit. on p. 32).

[21] The Linux Manual Authors. *perf-stat(1)*. URL: `https://man7.org/linux/man-pages/man1/perf-stat.1.html`. (accessed: 03:2021) (cit. on p. 33).

[22] Federico Parola. *eBPF synflood detector*. URL: https://github.com/FedeParola/ebpf-synflood-detector. (accessed: 03:2021) (cit. on p. 34).

[23] The Linux Manual Authors. *ethtool(8)*. URL: https://linux.die.net/man/8/ethtool. (accessed: 03:2021) (cit. on p. 37).

[24] Alexey Ivanov. *set_irq_affinity.sh*. URL: https://gist.github.com/SaveTheRbtz/887547. (accessed: 03:2021) (cit. on p. 37).

Acknowledgements

I think that no goal is due to a single person and for this reason I would like to dedicate a few lines to those who, throughout my university career and during the thesis, supported and endured me.

To Professor Risso, who made me passionate about the world of Networks and who taught me many things, not only in the academic field but also in the workplace and, also submerged by calls, found some time to advise and encourage me during the thesis, where it really made me understand the meaning of research.

To PhD students Federico and Simone who, even remotely, helped me in any way, with very long voice notes and calls, not only colleagues but also friends.

To all the wonderful Netgroup and CrownLabs colleagues who made me understand what it really means to work in a good team who always push you to learn more.

To the group of colleagues, friends and roommates of Turin: Leonardo, Francesco, Giuseppe, Vito, Simone, Riccardo, Alessandro, Hamza and all the others who have lightened these years of study with laughter, advices, panelle and even more study.

To my dear friends Marisa and Giuseppe, always ready to lend a hand when needed and to offer pistachio sweets.

To Francesca and her family, like a second family for me. She's the one who put up with the "nerdy computer engineering student" more than anyone else.

Last but not least, a huge and immense thank you goes to my parents who have always put me and my studies first, encouraging me and making me see that anything is possible. To my mother who continues to advise me both as a mother but also as a teacher. To my father who, since childhood, made me passionate about the world of technology and who continues to look at me and help me even from up there. All I have done or will ever be able to do is thanks to you two.

"Audentes fortuna iuvat"

Virgilio