POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Opportunistic Traffic Monitoring with eBPF

Supervisor

Prof. Fulvio RISSO

Candidate

Simone MAGNANI

October 2020

"Technology is nothing. What's important is that you have a faith in people, that they're basically good and smart, and if you give them tools, they'll do wonderful things with them."
Steve Jobs

Abstract

The growth of new technologies has opened new horizons for the network traffic monitoring and analysis. Innovative solutions like eBPF and XDP marked a clear distinction between traditional methodologies and new ones, which lead to a more personalized and, sometimes, more efficient filtering. Although, despite their flexibility and effectiveness, these technologies may seriously harm system performance, since they move the entire monitoring engine into the lowest layers of the operative system, introducing new problems related to the significant delay that an inefficient program may cause. This thesis proposes unusual and innovative usages of these new technologies, strengthening and favouring an in-kernel analysis of packets, and dynamically inserting or removing user-defined monitoring programs, exporting only the desired metrics using lightweight and standard data-interchange formats. *Polycube* is the framework used as reference, an open source research project developed by the Computer Network Group of Politecnico di Torino, which enables the creation of virtual networks and provides fast and lightweight network functions, as *bridge*, *router*, *nat* and many others. Within this complex and efficient framework, the service *Dynmon* has been created, starting from an early prototype, in order to accomplish dynamic network monitoring. The performance of this new service has been compared to a well-known and widely used protocol, *NetFlow*, and the promising and surprising results point out the efficiency of this new monitoring method. The advantage that *Dynmon* introduces is the possibility to perform adaptive network monitoring, choosing the granularity of data to be extracted, while the state-of-the-art tools extract a default set of features, independently by the type of the analysis, and it could result in an inefficient and heavy monitoring. Finally, this thesis presents also a real use case scenario, the TOSHI project, where *Dynmon* has been used in a more complex infrastructure, with the aim of detecting different cybersecurity attacks using eBPF/XDP as the packet analysis and features extraction method. Its usage perfectly meets the project need, which is to provide different dynamic network traffic monitoring probes, in order to extract packets features, according to the considered cybersecurity attacks.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

ML

Machine Learning

TCP

Transport Control Protocol

UDP

User Datagram Protocol

IP

Internet Protocol

ICMP

Internet Control Message Protocol

HTTP

HyperText Transfer Protocol

JSON

JavaScript Object Notation

VNF

Virtual Network Function

Chapter 1

Introduction

This chapter introduces the bases on which the thesis took place. Network monitoring is an essential component of IT security, but there are many drawbacks; traditional state-of-the-art techniques result inefficient in modern distributed service architectures, and this led to the growth of new solution for security management, based on completely different approaches.

Finally, this chapter presents the objective of the thesis and the real use case where the result of this research has been adopted, the TOSHI project.

1.1 Network monitoring

Network monitoring is a software that allows users to infer the condition of both a network and devices that are part of it, by tracking all the related problems (e.g., non-properly functioning devices or resources consume). It is carried out by means of diagnostic tools or specific hardware appliances, which are connected to the network and are capable of analysing the traffic passing through the specific network card.

Through network monitoring it is possible to generate alarms to appropriately signal the warning support system, which, according to the alarm type, can automatically react to such issues by applying counter measures, like firewalling policies. Despite the different enhancements that each network monitoring system may have, all these instruments offer the possibility to generate reports on the state of the network where all the problems are detected, in order to allow users to consult, by means of a web interface or a command line one, such results.

For security reasons, the ability to analyse traffic is essential, because it allows infrastructures to automatically find out when an unexpected or undesired behaviour is happening on the network, and to actively react to such threats.

1.2 The TOSHI project

The TOSHI (Total System Shield) project is a European project funded by EiT-Digital, a leading European digital innovation and entrepreneurial education organization. TOSHI aims to provide a high-performance, automated and adaptive defensive shield against cyber-attacks, exploiting efficient kernel-based technologies that are self-adaptive, configured to monitor and counteract security threats under the assistant of AI/ML mechanisms.

As later described in Section 7, TOSHI combines novel in-kernel Linux technologies such as eBPF/XDP with AI/ML algorithms to analyse events, detect and mitigate anomalies on the host machines. It provides and entire infrastructure, composed by many modules, in order to ensure scalability and robustness to the entire system.

In order to achieve a lightweight network monitoring and an effective anomalies' detection, the software injects to all the TOSHI-enable host devices two ad-hoc monitoring program, to extract information from traffic. Every pre-established period of time, a central entity gathers the collected data from the hosts, and performs the analysis using a complex neural network structure previously trained, in order to increase the accuracy of the detection engine. Once finished, the entity decides whether to insert into the hosts firewalling rules, in order to block the detected threat, or to keep monitoring, in the case it did not notice undesired traffic.

1.3 Thesis objective

The main objective of this thesis is to provide unusual and innovative network monitoring techniques adopting the latest technologies like eBPF/XDP. The proposed solution must be as versatile and flexible as possible, allowing creating networking probes that dynamically adapt to the user needs, changing the filtering program at runtime and exporting the requested metrics. The provided service is built within the Polycube framework, where eBPF and XDP are widely used to provide network functions efficiently exploiting these new technologies.

Moreover, to provide a concrete use case, the solution is used in the previously anticipated TOSHI project scenario. This extraordinary possibility allows to test the dynamic network monitoring provided by the newly introduced techniques during a cybersecurity attack. For this scenario, it is given also a performance evaluation, to make sure that, once deployed the monitoring program, the system keeps working efficiently due to the irrelevant overhead introduced.

Finally, a performance comparisons with the traditional network monitoring protocol *NetFlow* is performed, in order to check the validity of this thesis solution. The aim is understanding the relevance and effectiveness of this new approach, in order to evaluate if it can lead to better results, other than performing network monitoring within a completely new perspective.

Chapter 2

Problems and Limitations

This chapter introduces the main limitation and problems of the current state-of-the-art solution concerning network traffic and system monitoring. Among all the limits, three have been individualized and analysed, since they perfectly represent the reason why this thesis aims at providing completely different monitoring methodologies:

1. monolithicity and repetition of code;
2. offline analysis;
3. static monitoring programs.

2.1 Monolithic and repetitive applications

The first problem presented is related to the massive code duplication due to the integration of standalone monolithic network monitoring applications. As will be described in the following section with some example, there are many valid tools which allow users to perform every type of traffic filtering, using different techniques and deploying, in addition, various detection engines alongside. Although, many of these tools are designed for a specific application domain, limiting and hardening integration with similar software. In fact, when two different products are combined to perform a wider analysis, the resulting system will likely to have a lot of monitoring code duplicated, not only for the packet filtering itself, but also for its analysis.

In modern infrastructure, where almost all the services are deployed in a cloud-base environment with a significantly high networking capacity, the overhead introduced by the duplicate code can worsen the service performance, resulting in an inefficient usage of the system. Considering that nowadays both end-user and networking devices are increasing their packet processing capacity, deploying such infrastructures would mean losing the entire advantage that modern network interface cards bring.

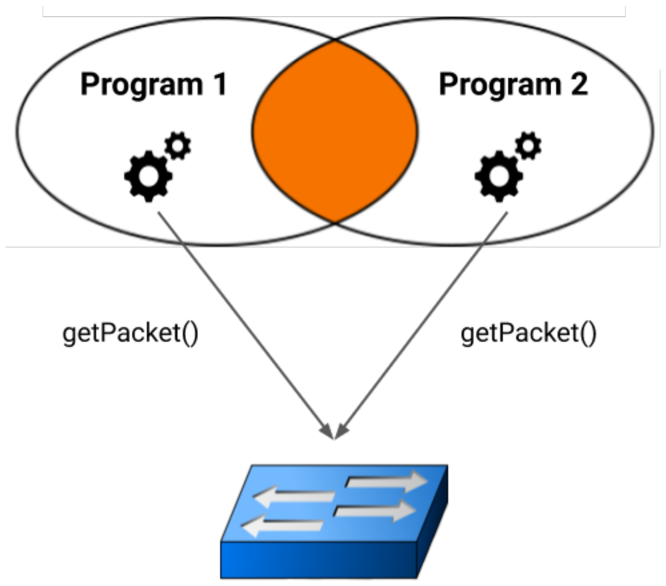


Figure 2.1: Example of two overlapping monitoring programs.

An example of such situation is provided in Figure 2.1. As it is depicted, there are two different network monitoring programs (e.g., one IDS and one simple analyser like Wireshark [1]) running on the same device, which in this case is a switch. The intersection area coloured in orange between the two program indicates that both of them are executing the same sub-set of instruction on the same packet, which will surely result in twice the computation time that an efficient integration would have taken. Moreover, in this case the performance is affected not only by this code duplication, but also from the same function used to retrieve the incoming or outgoing packet. In fact, the programs may not even know that they are both executing at the same time and that they could share the retrieved packet, in order to avoid asking one more unneeded time the packet to the underlying kernel. To avoid copying twice the packet to the user-space, these programs may exploit the *kernel bypass* technique, which allow to directly access the privileged kernel memory where the packet is stored. Although, this means that both of these programs have to deliver their custom device drivers, since the one offered by the kernel is not usable as it is completely bypassed. When dealing with custom device drivers, there is a high chance of incompatibility between the two of them, thus not all the program with that feature can be combined.

2.2 Offline analysis

The second problem that came up when dealing with state-of-the-art techniques, is the inefficiency of the “offline" analysis. Traditional tools may even use kernel BPF filters in order to capture only a certain sub-sets of the packets, but then they perform the packet analysis and feature extraction in the user-space program, meaning that the entire data structure of the packet has to be copied or at least accessed entirely more than once.

As a matter of fact, copying the packet introduces a not negligible overhead which may cause packets drop. On the other hand, in case the packet is directly accessed by the application and not copied to the user-space, the additional overhead introduced would be less than the previous case, but the running application which performs the analysis is still a user-space application, meaning that it does not have all the needed privileges to perform all the operations (e.g., system calls). Thus, there is still the risk to hang the packet for more time than an in-kernel analysis would have done, worsening performance.

Furthermore, as depicted in Figure 2.2, when performing an offline analysis, it is not possible to modify the packet default path, meaning that it cannot be dropped or routed to a different direction. As a result, the result of the analysis, like a firewall rule, can be applied only to the following packets, while the “malicious" one has already continued is path to the destination.

Finally, the user-space application is still a process running in the system, which requires CPU computational power in order to perform all its tasks. Therefore, the system has to host two different programs, one running in the kernel-space to filter the packets, and one running in the user-space to perform the analysis. While the former would run faster since it only has to match the packet according to the user-defined rule (e.g., TCP, IP source), the latter takes more time to execute since potentially it has to dig deeper the packet, meaning that the entire payload could

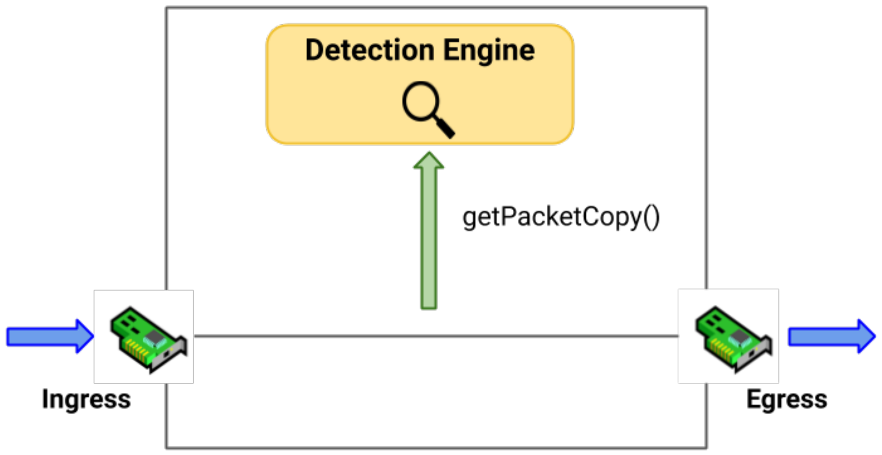


Figure 2.2: Offline analysis schema.

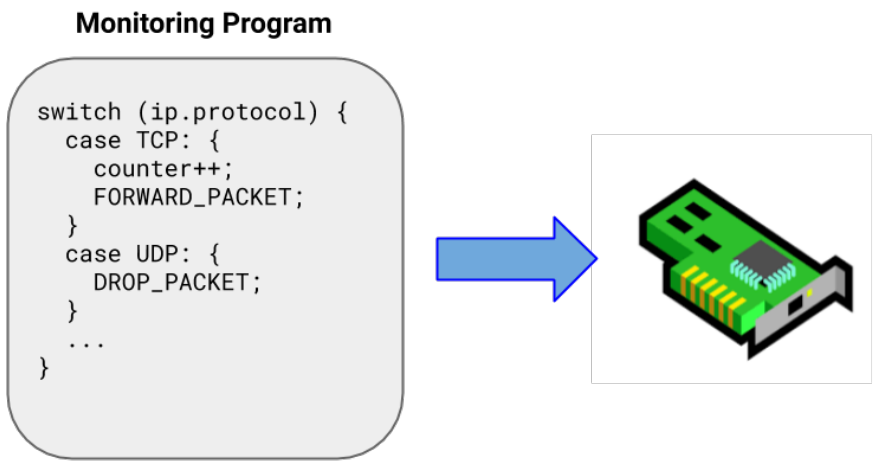
be inspected, and this surely cannot keep up in terms of speed with the respective kernel program, which can capture more packets than the application can handle.

2.3 Static pre-defined monitoring logic

The last consideration regards the poor flexibility of the monitoring programs. Traditional software tends to provide a lot of customizable filtering options, but they need to be pre-defined by the user before running the application. In case the user wants to change the logic at runtime, he/she needs to restart the entire application, or at the best case at least the filtering process.

The Figure 2.3 illustrates an example of a simple pre-defined monitoring program. It is clear that, once the program is injected in the system, the defined instructions (e.g., counter++) are going to be executed every time a packet is received, but in case the user wants to modify the instructions by adding or removing some code, he/she cannot do it, unless the entire program is stopped and recompiled. Despite the effectiveness and efficiency of the monitoring, having a static program could limit the entire analysis, harming the system with numerous instruction, even when not needed. For instance, a more in-depth analysis of headers and payloads could be enabled after a certain amount of time, when unusual traffic is recorded.

New technologies like eBPF, as it will be described later in this thesis, allow creating service chains and/or modify at runtime the network monitoring code injected in the system, in order to perform a more in-depth or superficial packet



```
switch (ip.protocol) {
  case TCP: {
    counter++;
    FORWARD_PACKET;
  }
  case UDP: {
    DROP_PACKET;
  }
  ...
}
```

Figure 2.3: Static monitoring program example.

analysis. This allows to build adaptive programs that, according to some constraints and user-defined logic, are able to save computational power when not required, increasing the networking capacity of the device.

Unfortunately, despite being easy to implement in the user-space application, most of the state-of-the-art techniques do not consider that option, or at least they perform an adaptive analysis in user-space, which do not affect the network filtering program running in the kernel or in the network interface card. This results in a high inefficiency of the application, which, instead of considering only the sub-set of packets requested, have to deal with all the traffic, and decide only later in the upper OS layer whether to perform a generic or a more detailed analysis.

Chapter 3

Related Work

This chapter briefly presents other network traffic monitoring tools to highlight their limits, since they have been taken into account both before and during the development of *Dynmon*, comparing their methodologies to the innovative proposed by the service.

3.1 Traffic monitoring with NetFlow

NetFlow [2], as described by authors, is an embedded instrumentation within Cisco IOS Software to characterize network operations, introduced on Cisco routers in 1996. This protocol provides the ability to collect incoming and outgoing IP network traffic, gathering data and producing statistics concerning all the detected flows. A typical flow monitoring setup, illustrated in Figure 3.1, consists of three main components:

- flow exporter, it aggregates packets into flows and exports flow records towards one or more flow collectors;
- flow collector, responsible for receiving, storing and pre-processing flow data received from a flow exporter;
- analysis application, which analyses received flow data in the content of intrusion detection or traffic profiling.

The Flow exporter is a user-space component which receives the filtered network traffic and aggregates packets into flows, identified by many parameters like IP addresses, ports, IP protocol, type of service and the interface. It handles both IPv4 and IPv6 traffic, preparing the data to be collected by the collector component. Once gathered enough data within a specified time window, the analyser applications runs a detection engine in order to perform the desired task (e.g., intrusion detection or profiling).

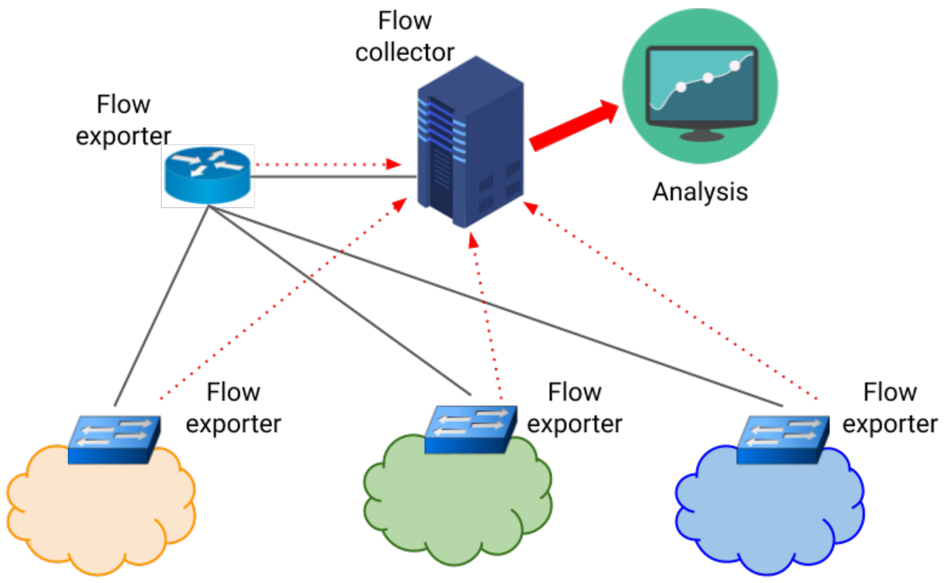


Figure 3.1: Netflow architecture.

Interestingly, the infrastructure may include also a web-based component, as provided in *ntopng*, to visualize data and provide users/admins some useful chart. In fact, while *NetFlow* is the main protocol developed by Cisco, a lot of different software have been developed to use and extend it, like *nprobe*, *ntop* and *ntopng*, and *ndump* to read from a command line interface the gathered data. The Listing 3.1 provides an example of output.

Listing 3.1: NetFlow output example.

```
Date flow start          Duration Proto  Src IP Addr:Port       Dst IP Addr:Port     Packets   Bytes Flows
2010-09-01 00:00:00.459    0.000  UDP    127.0.0.1:24920    -> 192.168.0.1:22126      1          46     1
2010-09-01 00:00:00.363    0.000  UDP    192.168.0.1:22126  -> 127.0.0.1:24920        1          80     1
```

Despite the terrific service that these *NetFlow*-based tools provide, they rely on the same traffic analysis method, which consists in copying the entire network traffic detected into the user-space, where all the packets are aggregated. As a matter of fact, copying packets from the kernel to the user application is a high-cost operation which worsen the global performance of the device (e.g., if the networking capacity was 40 Gbit/s, it is likely to drop to 10 Gbit/s or even less). As a result, these tools, that years ago have been proudly used for network monitoring, are beginning to lose their primacy, being overcome by all the innovative in-kernel analysis techniques. However, some of these tools like *nprobe* exploits the advanced kernel bypass PF_RING Zero Copy technique to ensure high speed traffic monitoring by directly accessing the packet, but as will be later described in the Section 8.4, applying filters and monitoring logic from the user-space slows down the entire process as well, resulting in worse performance and bitrate.

Even though this protocol allows collecting network traffic statistics, it is not flexible enough, since each monitoring node has to support the NetFlow protocol.

3.2 Domain-specific tools

Several tools exist for Data Plane network monitoring tasks, whose authors aimed at creating the most complete product according to their own functional objectives. This results in very powerful and complete tools, but that are oriented to a domain-specific application, such as traffic monitoring (e.g., for statistics, billing, accounting) and security (e.g., network anomalies detection, blocking possible attacks).

In the real world, the above tools are usually deployed side-by-side, with the obvious impact in terms of processing overhead (e.g., many functions are duplicated among them, or some function is active even if not needed). As a matter of fact, this results in a significant inefficiency and waste of resources.

The advantages of having many domain-specific tools are the interoperability, scalability and modularization of the final infrastructure, since users can aggregate different tools together and build their favourite topology. Although, despite these qualities may also belong to non domain-specific tools, there are a lot of disadvantages previously cited, which in a high-performance environment should be taken into account, as the obtained performance will likely to be way worse than expected.

Few examples of the most famous per-domain tools are:

- *ntop* [3]: as many of the solutions presented in the previous section, this tool is a high performance network monitoring solution, build over the NetFlow principle. Despite being widely used and having a handsome web-based user interface with customizable filters and packets inspection, *ntop* uses a Zero-copy packet distribution technique among different threads, which significantly enhance the monitoring program. Although, the techniques requires user-defined device drivers in order to correctly run, as the entire kernel layer is bypassed. Even though authors have built their own driver to interact with the network card, this process does not favour reusability and integrability with other software modules, which may use completely different techniques.

- *Snort* [4]: a free open-source network-based IDS (Intrusion Detection System) maintained by Cisco Systems. It is one of the most efficient IDS solutions available, both for the high-accuracy and extensibility of its detection engine. Despite the accuracy, *Snort* requires a lot of resources in order to correctly run all its component, which relies on libraries used to copy the packets matching the user defined rule. For that reason, networking performance may significantly decrease while using this tool, not for the analysis chain performed in the user-space, but for the overhead introduced by copying the matched packets into the slow path.

- *Suricata* [5]: a worthy rival of all the IDPSes (Intrusion Detection and Prevention System), which not only is widely used, but also it has been strongly enhanced, supporting also AI/ML modules for a more precise analysis. As all the other IDPSes, *Suricata* requires many resources to run the entire infrastructure, but it is the most accurate and complete tool in its category. Moreover, it contains a lot of programmable modules that makes it easy to extend, but as all the other tools alike, when integrating it with similar products there are going to be many similar functionalities repeated.

- *Netify* [6]: this tool gathers many features of both IDS and firewalls, implementing *Deep Packet Inspection* logic, which allows digging deeper packets payload in order to perform a more accurate analysis. This feature is already present in many other solutions, as an in-depth packet analysis shares the same principles of what in an IDS could be a “rule-based" detection method, where all the payload is being analysed in order to match a specific pattern. In addition, every scenario should have its own rule and analysis engine running, thus a duplication of the entire processing process is inevitable.

3.3 Traffic monitoring with eBPF

Some applications of eBPF/XDP concerning network monitoring and cybersecurity have already been published. For instance, an example quite similar to the TOSHI project, but with a completely different aim, is provided by this [7] paper. The aim of the paper is to measure the efficiency of offloading network functions like DDoS mitigator into the hardware, the SmartNICs, exploiting the usage of these new technologies to perform controls at low level and at high speed. The paper presents the DDoS mitigator scenario, where all the incoming malicious traffic belonging to a DDoS attack should be blocked. In order to do so, the authors used eBPF/XDP to extract features from the incoming traffic, and compute data in user-space using heuristics algorithms, which are less precise than an entire neural network structure. Once the computation finishes, the results (malicious IPs) are injected into the eBPF programs, which will deny all traffic incoming from those sources.

Concerning only observability in a micro-services cloud-based environment, the ViperProbe [8] framework has been proposed. The tool has been built to strengthen both network and system monitoring, exploiting the different type of eBPF probes which can be currently created. In fact, eBPF can be efficiently used also to track system events, like system calls, in order to gather statistics and record the entire system behaviour. ViperProbe proved to have limited overhead, providing an analysis of eBPF metric performance, examining Envoy's metric performance profile, and showing that its eBPF metrics were significantly more effective for horizontal autoscaling.

Finally, a worth mentioning growing framework is *Cilium* [9], an open source software for transparently securing the network connectivity between application services deployed using Linux container management platforms like Docker and Kubernetes. At the foundation of Cilium is a new Linux kernel technology called BPF, which enables the dynamic insertion of powerful security visibility and control logic within Linux itself. Because BPF runs inside the Linux kernel, Cilium security policies can be applied and updated without any changes to the application code or container configuration.

Chapter 4

Exploited Technologies

This chapter presents the main technologies that have been used in order to create both the architecture and implementation proposed in the following chapters. Most of the information that this chapter provides are strongly inspired on the relative software documentations present in the respective websites.

4.1 eBPF and XDP

Proposed by Alexei Staravoitov in 2013, eBPF [10] is the enhanced version of BPF (Berkeley Packet Filter), an important technology which provides on some Unix-like OSes a raw interface to data link layers in a protocol-independent fashion, to allow network monitoring by multiple applications running in user space. With the next version, the entire architecture has changed, including both modification to the underlying virtual CPU and the possibility to directly access and modify the packet from the kernel. In fact, eBPF deploys a sort of in-kernel virtual environment where users can run their monitoring code, managing the real packet (and not a copy), hence enabling a new breed of applications such as bridging, routing, NATting and more.

The Figure 4.1 depicts the architecture of eBPF, considering the main components involved during the entire compilation process and life cycle of a program.

To start with, eBPF code can be written in a restricted version of C, which allows easier program development and more powerful functionalities with respect to bare assembly code. The C code is then compiled by the LLVM/Clang [11] compiler, which translated the entire source code into bytecode. The bytecode

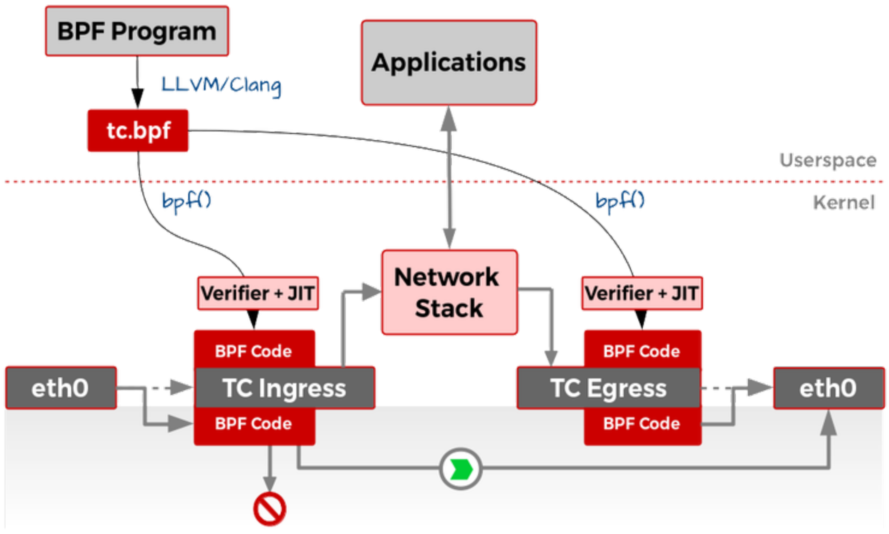


Figure 4.1: Architecture of eBPF.

is then forwarded, using a Linux dedicated system call, to the kernel, where the eBPF Verifier and JIT (Just in Time compiler) perform various controls, in order to ensure that, after injecting the program, the system will not be harmed. Once the program successfully passes the validation phase, it is attached to the specified hook, which can refer to both the packet reception engine or any Linux generic kernel events represented by a system call. From that moment on, the program is called every time the specific event represented by the hook is generated.

Considering the networking scenario, there are two main hooks exposed by every OS: TC Ingress and TC Egress. As their name suggests, they are the entry and exit points of packets within the kernel of an OS. Attaching a program to these hooks allows both reading and modifying the real structure of the packet before it reaches the higher OS layers, and, more interestingly, redirecting or even dropping the packet.

Every time a packet passes through the Ingress or Egress hooks, the eBPF program is triggered and a volatile “packet memory" is defined, which is a temporary memory valid only for the current packet, meaning that in this type of memory cannot be stored information concerning subsequent packets.

Interestingly, to address the need of storing data among different packets, eBPF has introduced a set of memory areas called maps. Maps are data structures where the user can store arbitrary data with a key-value approach: data can be inserted in a map by providing the value and a key which will be later used to reference it.

That said, there are many eBPF maps defined for almost all the possible scenarios, and they not only replicate most of the data structures that higher languages provide like arrays and hash maps, but they also introduce more specific map types to perform efficient network monitoring, like histograms, queues, PER-CPU maps and many others. Moreover, other important advantages of using maps are that their content is preserved across program execution, and that they can be shared both between programs belonging to the same or to different hooks. This feature allows building complex network service chains, which can monitor both incoming and outgoing traffic sharing data between them.

In addition to all the innovative features introduced by eBPF, there is XDP [12], a programmable, high-performant packet processor in the Linux networking Data Path, which provides an addition hook to be used with eBPF programs in order to intercept packets in the driver space of the network adapter, before even reaching the Linux kernel. A significant advantage introduced by this early processing mechanism is that it avoids the overhead and memory consumption added by the kernel to create the socket buffer structure, which wraps the entire packet intercepted in the standard TC mode. Once the packet reaches the NIC, all the attached eBPF programs are executed, and they can decide whether to drop the packet or let it continue through the networking stack. One of the main use cases is pre-stack processing for filter or DDoS mitigation, where, given an eBPF map (probably filled by the Control Plane logic) containing the IPs and/or ports to blacklist, the program can instantly drop the incoming packets. Unfortunately, XDP is still a growing technology, and it does not support Egress hook in the NIC yet, even because it would not make any difference for all those packets which are directly sent by the machine running the programs, as it would have to go through the entire networking stack spending time for analysis anyway. Thus, a more complex service chain cannot be delivered fully in the XDP mode yet, but there are going to be improvements in the following months, as already announced.

4.2 Polycube

Polycube [13] is an open source framework developed as research project by the Computer Networks Group of Politecnico di Torino, which allows the creation of Virtual Network Functions (e.g., bridge router NAT, load balancer firewall and DDoS mitigator) capable of efficiently inspecting and manipulating the network traffic, and creating high-performant service chains exploiting the eBPF/XDP technology.

All Polycube VNFs features are unified in a common point of control, which enables the configuration of high-level directives and structure of the desired service chain and topology. This model is integrated in service agnostic user space daemon, *polycubed*, a program in charge of interacting and managing the different services, and provides accessibility to all external user through standards REST APIs. Each VNF, commonly called *cube*, is defined by a service, a C++ group of classes which define both the internal operation and functionalities, and the accessible endpoints within the daemon. The cubes are similar to plugins, since they can be installed, launched at runtime, and they are compiled as external libraries, which are loaded by the daemons if necessary. Once the daemon registers the service, users can create different instances of it by using the REST interface.

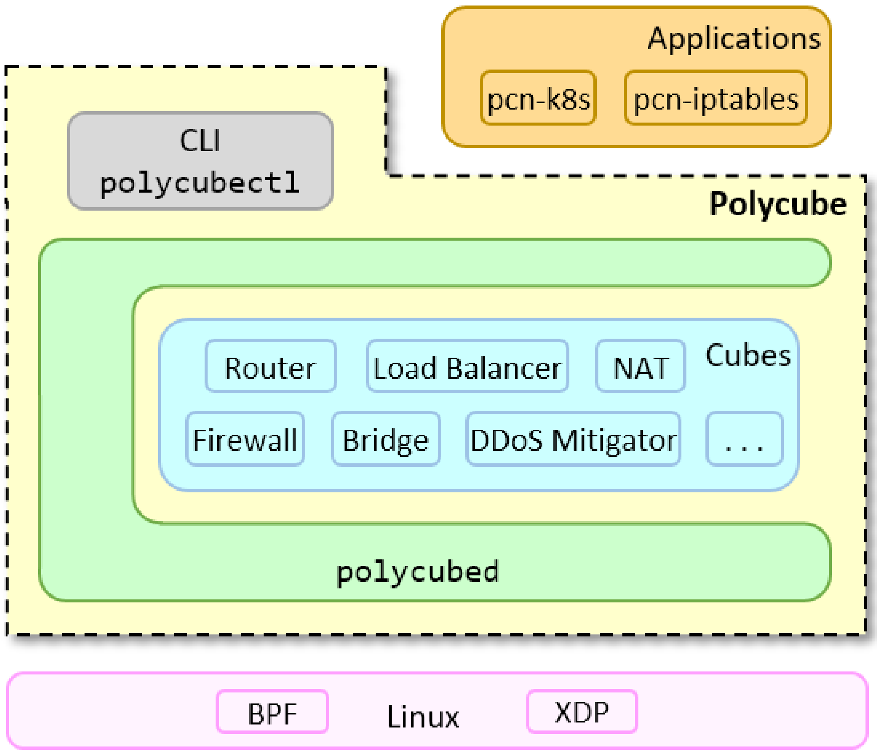


Figure 4.2: Polycube architecture.

Figure 4.2 depicts the Polycube architecture. There are two standalone application, *pcn-k8s* and *pcn-iptables*, realized to integrate and export the known functionalities of *Kubernetes* and *iptables* within the framework. The central layer represents all the user-space components, like the *polycubed* daemon, the command line interface tool *polycubectl*, and all the Control Plane of the services (router, firewall and others). Finally, the lowest layer contains all the eBPF/XDP programs running the services Data Plane, which are in charge of handling packets at high speed by applying fast decision-making logic.

Interestingly, every cube is characterized by a fast path (Data Plane), commonly known as the eBPF code injected and running in the kernel, and a slow path (Control Plane), running in user-space within the service itself, which takes into account all those packets that cannot be fully processed in kernel or that require additional operations and may slow down the processing of all the following ones.

The Data Plane portion of a (virtual) network service is executed per packet, with the consequent necessity to keep its computational cost as small as possible. For each packet, the fast path retrieves both the entire packet structure and metadata associated to it from the reception queues, then all the eBPF code injected is executed. Usually, such programs contain fast and precise operations, such as packet parsing, memory lookup and statistics. However, while eBPF offers the possibility to perform some complex and arbitrary actions on packets, it suffers from some well-known limitations due to his restricted instruction set allowed, in order to preserve the integrity and safety of the system. To overcome those limitations, Polycube introduces an additional Data Plane component no longer limited by the eBPF virtual machine, allowing programs to execute arbitrary code. Moreover, a set of helper functions to redirect packets to the Control Plane and perform other useful functionalities (e.g., checksum re-computation if the packet has been modified) has been introduced, to ease the development and increase code-readability.

On the other hand, the Control Plane of a network is the place where all the out-of-band tasks, to both control the Data Plane and react to more complex events (e.g., routing protocols and spanning tree), are implemented. Moreover, it represents the entry point of all the interaction with external entities (users or other services) that needs to access resources, modify or consult service parameters and receive notification from the service fast path through a shared event buffer. Control plane tasks are way slower that Data plane ones, thus calling such functions from the fast path is strongly suggested only in certain cases, despite it provides an in-depth packet analysis.

Chapter 5

Dynmon Architecture

This chapter presents the architecture of the proposed solution: *Dynmon (Dynamic Network Monitor)* [14]. The architecture, which extends the one of a previous prototype, has been studied to efficiently exploit Polycube VNFs and, as the reference use case, achieve the best outcomes for the TOSHI project.

Considering the current state of the framework, which does not allow detaching and remotizing the Control Plane from the Data Plane, the chosen solution provides a local self-adaptive service able to handle generic monitoring programs and exporting the defined metrics accordingly.

5.1 Overview

Every Polycube service follows the structure illustrated in Figure 5.1. While the Control Plane is a unique entity, the Data Plane can be composed by different programs injected in the system in pipeline, which allow building complex and modularized infrastructure.

As the names suggest, these two entities have different but complementary aims. The Data Plane is the more sensible entity, because it directly affects the device's networking capacities, meaning that every instruction in its programs is weighted and should be essential, otherwise it will slow down the entire packet management process. Briefly, it contains the logic responsible for:

- monitoring traffic and filtering packets according to the eBPF programs;
- reading and/or filling shared data structures with Control Plane to improve

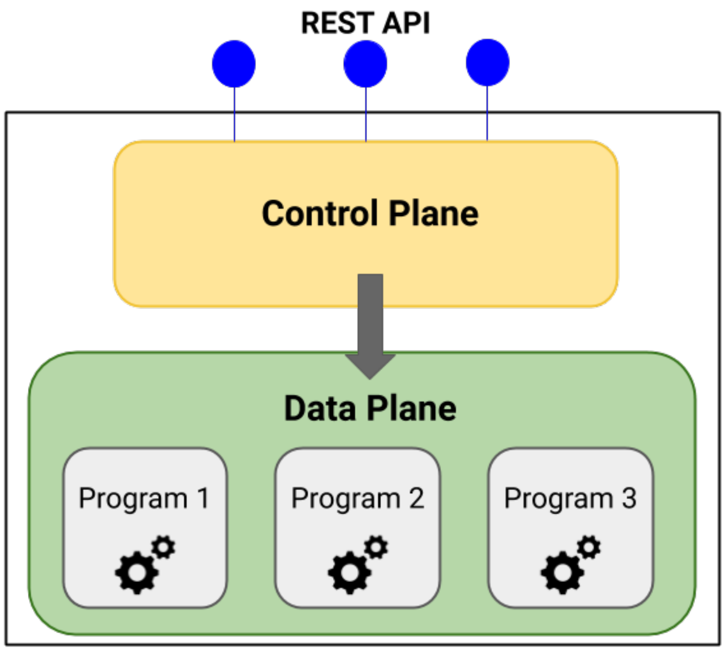


Figure 5.1: Polycube service architecture.

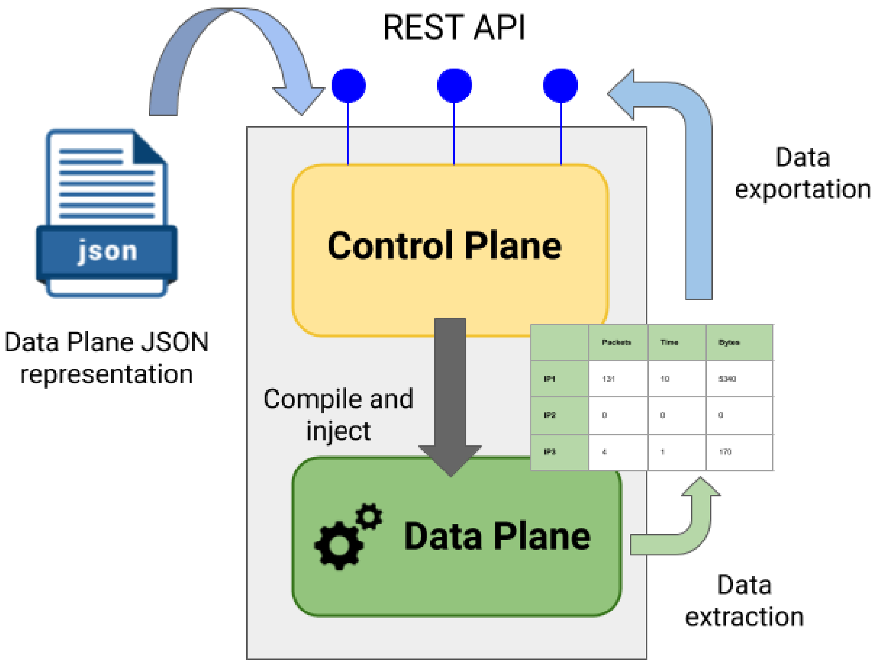
traffic analysis and to export monitoring values outside.

On the other hand, the Control Plane generally does not affect at all networking capacities, since it is an independent process running in user-space which computes data by applying a specific logic. Although, this concept does not apply in those cases where, in order to analyse and produce results, the Control Plane needs to interact with structures shared with the Data Plane to manage the values (read, write or erase). Generally, the main tasks of a Control Plane program are:

- to manage the entire Data Plane, injecting and removing program when needed;
- to manage shared structures, reading the gathered data and act accordingly;
- to provide access to external users, exploiting the REST interface to export information concerning the service.

5.2 Self-adapting Control Plane

To support as much as different types of users dynamic injected configurations, *Dynmon* presents a general purpose self-adapting smart Control Plane, which is able, given the desired Data Plane representation, to adapt its extraction mechanisms according to the specified metrics. Therefore, the Control Plane is the core of the entire service.



	Packets	Time	Bytes
IP1	131	10	5340
IP2	0	0	0
IP3	4	1	170

Figure 5.2: Dynmon architecture.

As depicted in Figure 5.2, the structure follows exactly the principles of a generic Polycube service, allowing interaction via REST API. Interestingly, while all the other services have a constant Data Plane, *Dynmon* aims at dynamically accepting user defined monitoring programs, which may vary over time at user convenience. Thus, a basic Data Plane is provided and injected when an instance is created, but the service is ready to accept and replace new configurations.

Since Data Plane refers both to Ingress (incoming) and Egress (outgoing) interfaces, for each of them a configuration contains:

- a name, to represent the current configuration;

- the eBPF code, which corresponds to the Data Plane program to be injected;
- a list of metrics, containing all the specifications of the desired data structures (eBPF maps) to be exported.

A few advantages of this approach are:

- single point of control: the VNF is instantiated in the same machine of the monitored host, thus, there is no need to export data to remote controllers, handling theoretically thousands of connections;
- continuity of operations: accepting dynamic configurations, the service is able to substitute current settings with new one, without the need to delete and create a new instance;
- dynamicity and versatility: the service is built to change and adapt at will whenever required;
- flexibility: supporting a wide range of the current existent data structure, *Dynmon* ensures that the largest number of user programs are correctly compiled and running as expected.

5.3 VNF Workflow

The Figure 5.3 illustrates the typical workflow of interactions between an external user and Dynmon, which are:

1. *Service Creation*: interacting with the Polycube daemon (*Polycubed*) a basic instance of Dynmon is created and ready to accept a new configuration. Unfortunately, during this phase it is not possible to specify the Data Plane to be injected, but still other parameters like the network interface to be attached to can be specified (otherwise a user can decide to attach the service to an interface later in a different REST request).
2. Data Plane injection: the user contacts the daemon, which relays the request to the *Dynmon* instance, specifying the new configuration to be used.
3. Data Plane tuning, compilation and injection: this is the most delicate phase, since the configuration is meticulously analysed, the code is compiled and,

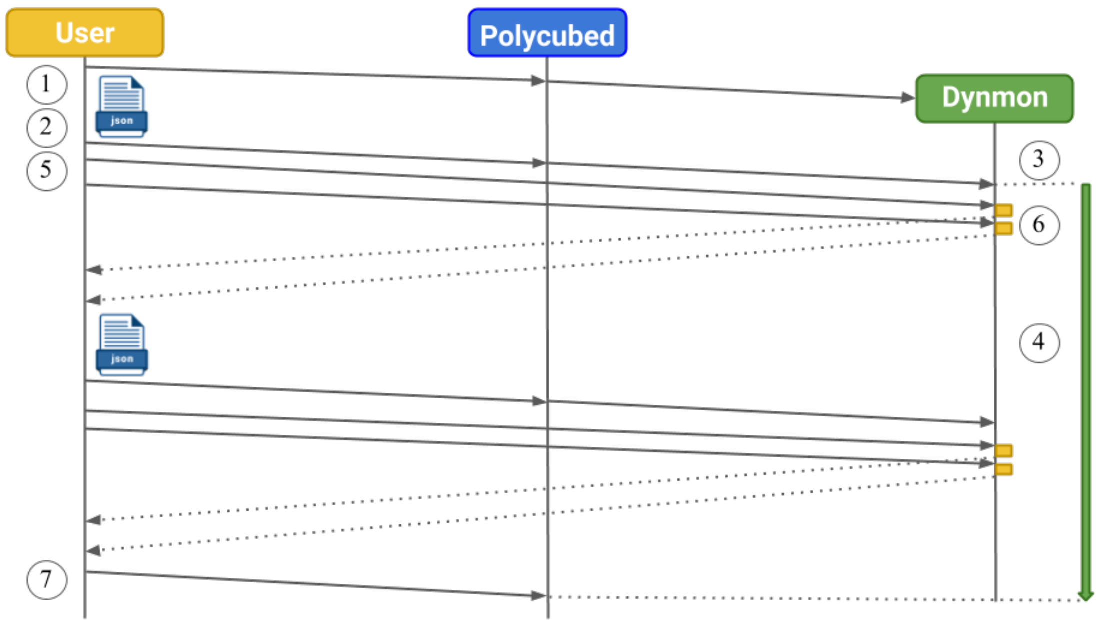


Figure 5.3: VNF typical workflow.

if successfully, injected in the system. Before compiling, a few advanced enhancements to the code are performed according to optional parameters specified in the configuration, but this is later discussed in Section 5.7.

4. Data collection: once the new Data Plane is injected, every time a packet goes to through the apposite hook (Ingress/Egress) the injected program is executed and it properly updates the BPF maps corresponding to the metrics to be exported.

5. Metric request: the user can require all the collected metrics, a few of them or just the one of a specific program chain (Ingress/Egress) by querying different endpoints.

6. Metric extraction: the service, once received the request, retrieves all the information to extract data from the BPF map (size, value types, etc.) and exports their content using the JSON format. At this point, the service may also perform, according to the additional parameters in the configuration, other operations like emptying the map once read.

7. Service deletion: when the Polycube daemon is stopped or the user appositely requires it, the service and all its related eBPF programs are removed, to keep

the system safe avoiding harming performance.

5.4 YANG model description

Polycube framework allows developers to define the structure of a service using YANG [15], a language which, taken a model as input, produces the code source code accordingly, which not only saves developers time by providing a working basic structure, but it is also compliant to the RESTCONF protocol, defined in RFC-8040 [16].

Here follows the data model defined for *Dynmon*. Some parts have been omitted for simplicity (mandatory values, descriptions, etc.).

Data Plane container

The Data Plane container is the outer container containing both Ingress and Egress containers and it has a simple structure as follows. *Dynmon* requires both the two inner containers to be defined in the injected configuration, even when the user wants to insert only one type of program (Ingress/Egress), otherwise it throws an error to avoid an undetermined behaviour. An example of the YANG container definition is provided in Listing 5.1.

Listing 5.1: Data Plane container YANG model.

```
container dataplane-config {
    container ingress-path {
        presence ingress;
        uses path-config;
    }
    container egress-path {
        presence egress;
        uses path-config;
    }
}
```

Ingress/Egress container

These two containers (Listing 5.2), which they differ only for the name, are the most essential one, since they contain:

- *name*, a string used to help the user identifying its service;
- *code*, the eBPF code to be injected;
- *map-name*, the name of the BPF map which contains the value of the metric;

- *extraction-options*, additional parameters to customize extraction;
- *open-metrics-data*, the exported metrics in the OpenMetrics [17] format.

Listing 5.2: Ingress/Egress container YANG model.

```
grouping path-config {
    leaf name { type string;}
    leaf code { type string;}
    list metric-configs {
        key "name";
        leaf name { type string;}
        leaf map-name { type string;}
        container extraction-options { ... }
        container open-metrics-metadata { ... }
    }
}
```

Open Metrics container

The following container represents the exported metrics in the OpenMetrics format. The resource follows the OpenMetrics syntax, exporting data specifying their name, value and the type (histogram, normal data, etc.), as reported in Listing 5.3.

Listing 5.3: Open Metrics container YANG model.

```
container open-metrics-metadata {
    leaf help {type string;}
    leaf type {
        type enumeration{
            enum Counter;
            enum Gauge;
            enum Histogram;
            enum Summary;
            enum Untyped;
        }
    }
    list labels {
        key "name";
        leaf name { type string;}
        leaf value { type string;}
    }
}
```

Extraction options container

Concerning the optional parameters and extraction enhancements, this container (Listing 5.4) defines two possible tuning which will be analysed in-depth in Section 5.7. In brief:

- *empty-on-read*, a boolean value used to teach *Dynmon* to erase the content of the map whenever the user requires it;
- *swap-on-read*, another boolean value used to teach *Dynmon* to perform all operation on maps atomically, without hanging the underlying Data Plane.

Listing 5.4: Extraction options container YANG model.

```
container extraction-options {
    leaf empty-on-read { type boolean;}
    leaf swap-on-read { type boolean;}
}
```

5.5 Dynamic Data Plane configuration

Given the previously defined YANG models, in order to configure the Data Plane of the VNF, the user must provide a configuration file in the JSON format that follows the YANG containers structures. As a consequence, the configuration file must contain both *ingress-path* and *egress-path* field, each one composed by:

- *name*, the name of the configuration;
- *code*, the eBPF monitoring code to be injected in the system;
- *metrics-configs*, the list of the exported metrics with all their additional parameters like *open-metrics-metadata* and *extraction-options*.

Listing 5.5: Data Plane configuration example.

```
{
    "ingress-path": {
        "name": "Packets counter probe",
        "code": "\r\n BPF_ARRAY(PKT_COUNTER, uint64_t, 1);\r\n ...",
        "metric-configs": [
            {
                "name": "packets_total",
                "map-name": "PKT_COUNTER",
                "open-metrics-metadata": {
                    "help": "Number of packets.",
                    "type": "counter",
                    "labels": []
                },
                "extraction-options": {
                    "empty-on-read": true
                }
            }
        ]
    },
    "egress-path": {}
}
```

The configuration provided in Listing 5.5 defines a simple eBPF program to count the number of packets passing through the network interface, exporting such value as a metric named *packets_total*, which refers to the BPF map *PKT_COUNTER*. In addition to the OpenMetrics metadata specified, there is also the extraction option *empy-on-read*, which specifies to reset the map whenever the user requires that specific metric.

The example contains only Data Plane for the Ingress hook, but a user can reuse the same configuration also for the *egress-path*, counting the outgoing packets. Although, despite the Egress map name which may be equal to the Ingress one, the two programs do not share the same structure for the map, but a new one with the same name within a different scope. In order to use the same map in the two hooks, users should specify something like *BPF_ARRAY_SHARED(...)* in one program and *BPF_ARRAY("extern"...)* in the other, to make the map visible.

5.6 Metrics

Metrics are exported in two different formats to ease integration within other data visualization services (e.g., Prometheus [18]):

1. JSON: one of the most used and lightweight data-interchange format, easy to use, efficient and capable of reproducing the content of a BPF map (both the primitive types and the structured ones);
2. OpenMetrics: the new standard designed for exposing metric data for observability (values over time). Despite being widely used, this format does not support structured data structures (union, struct or nested); thus, only primitive data types and arrays can be exported.

The service exposes two respective endpoints to make data accessible in the desired format:

1. */metrics*, which returns collected data in JSON format;

2. */open-metrics*, which return metrics in OpenMetrics format.

The Polycube framework, exploiting the use of YANG models to generate the base code of a service, generates also other useful endpoints to allow access to specific metrics and their fields:

1. */metrics/ingress-metrics*, to access the entire metrics list of the Ingress program;

2. */metrics/egress-metrics*, to access the entire metrics list of the Egress program;

3. */metrics/{in-e}gress-metrics/{metric-name}*, to access a single metric by providing its name, referring to the specific program type;

4. */metrics/{in-e}gress-metrics/{metric-name}/value*, to access only the value of a single metric by providing its name, referring to the specific program type.

In the example provided in Listing 5.5, to access the *packets_total* metric the service can be queried sending an HTTP GET request to */metrics/ingress-metrics/packets_total* and the output would be like in Listing 5.6

Listing 5.6: Metric extraction output example.

```
{
    "name": "packets_total",
    "value": [
        6850
    ],
    "timestamp": 1599403699607360
}
```

Otherwise, if just the value needs to be obtained, an HTTP GET request to the endpoint *.../metrics/ingress-metrics/packets_total/value* would produce a response containing only *[6850]*.

5.7 Run-time code enhancements

The last important architectural component to describe is the CodeRewriter component. It is an extremely advanced code optimizer to adapt user dynamically injected code according to the provided configuration. It basically performs some optimization in order to provide all the requested functionalities keeping high performance and reliability. Moreover, it relies on eBPF code patterns that identify a map and its declaration, so the user does not need to code any additional information other than the configurations for each metric he wants to retrieve.

Currently, there are only two additional parameters supported, as described in the previous section: *empty-on-read* and *swap-on-read*. While the former is used to specify that the content of a map should be erased once read, the latter triggers the CodeRewriter component, since it specifies that all the operations concerning the associated map should be atomic. Few examples are provided in Section 6.2.2.

The component exploits the use of pattern to match, substitute and modify parts of the code provided in the injected configuration. There are two different types of rewrites performed: PROGRAM_RELOAD and PROGRAM_INDEX_SWAP.

PROGRAM_INDEX_SWAP rewrite

This option is the most advanced one, aiming both at meeting user specifications and speeding up the entire process. It exploits the usage and creation of program chains that the framework allows doing. The steps it reproduces are as follows:

1. clone the injected eBPF code;

2. modify in the cloned code all references to all those maps declared *swap-on-read*;

3. create a pivoting program which, according to the current state of the service, will forward the incoming/outgoing packets to the right program;

4. inject in the system the pivoting code as first, then the user injected code and the cloned one.

The three programs are all injected once in the system, so there are no problems nor delays due to re-compilations and re-injections. An example of functioning is provided in Figure 5.4: for every packet, the pivoting program is called and, using internal BPF maps accessible also from the Control Plane to manage program indexes, decides whether to call the original user defined program (Program v1) or the cloned and modified one (Program v2).

Whenever a user requires the metrics, the Control Plane changes the index, contained in the shared eBPF map, which corresponds to the version of the program to be called by the pivoting program. Therefore, since the maps contained in the two versions of the program are different, the collected metrics returned to the user are retrieved from the actual unused one, ensuring atomicity as the Data Plane program started to use the other one as soon as the request arrives.

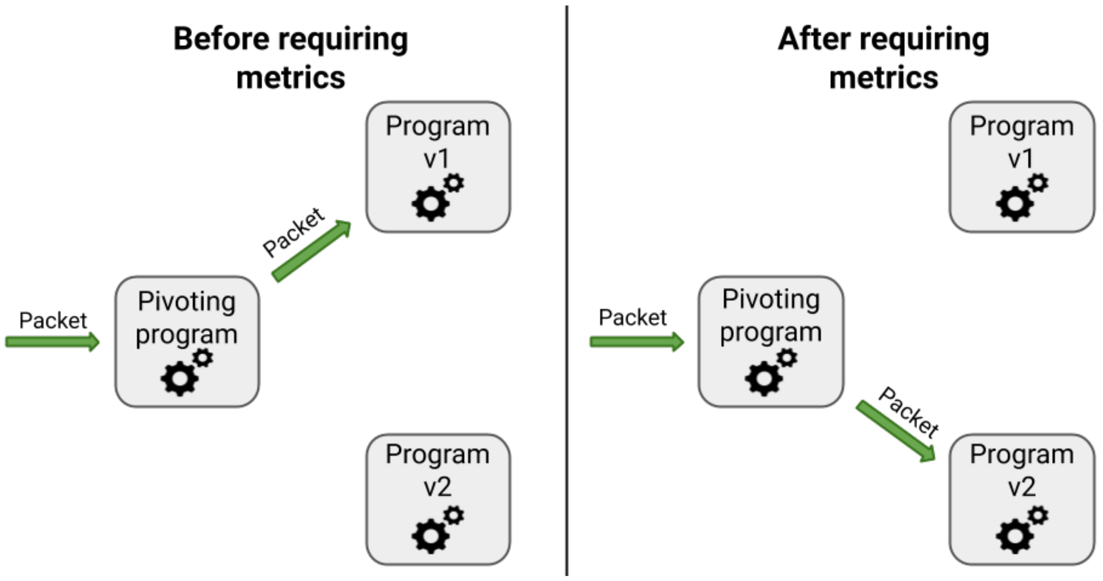


Figure 5.4: Packet path with PROGRAM_INDEX_SWAP rewrite type.

PROGRAM_RELOAD rewrite

The PROGRAM_RELOAD option is simpler to understand and it is used as a fallback option in case some error occurs during the PROGRAM_INDEX_SWAP one or the syntax of the maps used by the user in the eBPF program is wrong. Despite meeting the user specification ensuring atomicity among maps, this solution has some drawback in terms of performance.

The steps reproduced are:

- clone the injected eBPF code;
- modify the name of the maps declared *swap-on-read* in the cloned code fictitiously;

- inject alternatively in the system the original code and the modified one.

From these steps it is clear that this solution is slower than the previous one, since the obtained codes are alternatively injected in the system, meaning that they are always re-compiled and re-injected. When a user requires the collected metrics, the Control Plane, before even extracting them, changes the currently loaded program with the other one that, even though it contains the same instructions, uses a different map (fictitious one). As a result, the Control Plane is able to read the correct map atomically, even thought it surely it handles more slowly user HTTP requests.

Chapter 6

Dynmon Implementation

This chapter presents the implementation of the previously described architecture, adding more technical details concerning the chosen solution. Moreover, other problems related to the Data Plane part are discussed, describing also the choices to address such issues. Finally, it will discuss of two user-friendly tools created to facilitate both deployment of a *Dynmon* instance and metrics extraction.

The source code can be found on the GitHub repository of the Polycube framework [14].

6.1 Used languages

According to the Polycube framework's standards, the main programming language used to implement *Dynmon* was C++, an object-oriented and performing version of C. A second language used for the model of the service was YANG, thanks to a basic code structure for the service is created. Concerning the user-friendly tools, the language Python was used, for its versatility and programmability, which allow us to build high level programs to interact with Polycube using HTTP requests easily.

Finally, for the tests it has been used BASH scripting integrated with the Polycube command line tool (*polycubectl*). The tests aim at covering the most possible use cases in Dynmon, to confirm its correct functioning.

6.2 Main classes

Following the YANG code generation, a lot of classes and packages are automatically defined, including those where the accessible APIs are described. However, the service as it is would not be usable at all, since their operational logic needs to be implemented. Thus, in this section are listed and analysed the essential classes introduced.

6.2.1 Dynmon

The *Dynmon* class represents the instance of the service, which contains all the Control Plane logic. In addition, it represents also the entry point of the service, as, depending on the HTTP request performed, it contains all the methods to correctly produce a response.

This class, other than all the structures needed to handle concurrency, contains a reference to the *DataplaneConfig* object, which represents the current Data Plane of the VNF. Every time a new configuration is received, if it is correct and does not produce errors (both compile-time and injection-time) the reference is updated, otherwise the previous program is kept.

The *Dynmon* class contains also a *SwapStateConfig* object for each program time, which is used to keep track of the current enhancements, if any, applied by the *CodeRewriter* class. Finally, all this class methods are linked to the service REST API, and they perform all the functions described in the VNF workflow.

The Listing 6.1 provides a code snippet of this class, in particular of the method used to retrieve the collected metric specified by the name referring to the Ingress program.

Listing 6.1: Snippet of the *Dynmon* class.

```
std::shared_ptr<Metric> Dynmon::getIngressMetric(const std::string &name) {
  logger()->debug("[Dynmon] getIngressMetric()");
  auto ingressPathConfig = m_dpConfig->getIngressPathConfig();
  auto metricConfig = ingressPathConfig->getMetricConfig(name);

  std::lock_guard<std::mutex> lock_in(m_ingressPathMutex);
  triggerReadIngress();

  try {
    // Extracting the metric value from the corresponding eBPF map
    return do_get_metric(name, metricConfig->getMapName(), ProgramType::INGRESS,
        metricConfig->getExtractionOptions());
  } catch (const std::exception &ex) {
    logger()->warn("{0}", eg.what());
    std::string msg = "Unable to read " + metricConfig->getMapName() + " map";
    logger()->warn(msg);
    throw std::runtime_error(msg);
  }
}
```

6.2.2 CodeRewriter

The *CodeRewriter* is not a real C++ class, but it actually is a namespace which defines functions used to parse and enhance the injected code. For future services and/or improvements, its functions has been made accessible to anyone, in case its tuning are needed elsewhere.

The enhancement methods rely on patterns to match in the provided eBPF code, in order to locate and modify maps declaration and usage within the code. Moreover, the logic implemented in the methods that provide the PROGRAM_INDEX_SWAP rewrite is really advanced, as it tries to cover the most scenarios and maps usage. On the other hand, the PROGRAM_RELOAD optimization is always ensured and does not produce any error, since it simply clones the code as it is modifying only the map names.

Concerning the PROGRAM_INDEX_SWAP option, the detailed list of performed steps is as follows:

1. Fix non-swappable maps: in this phase all the maps belonging to the program but not declared specifying the *swap-on-read* parameter need to be slightly modified as well, to make them accessible between the two new versions injected of the program. This step is the trickiest one as there are a lot of map types that needs to be checked and that may belong to different scopes. In fact, for each map declaration the function looks for:

    (a) “extern" or “pinned" maps, as their declaration does not need to be modified in the cloned code;

    (b) standard (e.g., *BPF_TABLE(...)*) or customized (e.g., *BPF_ARRAY()*) map declarations, since the former is supported while the latter is not;

    (c) “shared" or “public" maps, as in the cloned code they need to be changed into *extern* (which refers to the map declared by the original program);

    (d) every declaration that does not match any of the previous controls is modified both in the original eBPF code and in the cloned one, adding the “shared" attribute to all these maps in the former and “extern" in the latter.

2. Fix swappable maps: the second phase consists in substituting the name of maps declared with the *swap-on-read* parameter with a new fictitious one (e.g., from *MAP_A* to *MAP_A_1*). The maps of interest are those declared as “shared", “public", “extern" or “pinned", since all the other ones are program specific, thus declaring two maps with the same name in two different programs effectively creates two different maps.
3. Creation of the pivoting code: this final step consists in taking the default pivoting code provided in the namespace and modifying it according to the program type (Ingress/Egress).

In the Listing 6.2 is provided a snipped of the only public function of the *CodeRewriter* namespace, the one used to decide which type of enhancements to use and return the appropriate configuration.

Listing 6.2: Snippet of the *Code Rewriter* namespace.

```cpp
void CodeRewriter::compile(std::string &original_code, ProgramType type,
        const std::vector<MetricConfigJsonObject> &metricConfigs,
        SwapStateConfig &config, const std::shared_ptr<spdlog::logger>& logger){
  bool is_enabled = false;
  std::vector<std::string> maps_to_swap;
  /*Checking if some map has swap-on-read, otherwise no compilation*/
  for(auto &mc : metricConfigs) {
    if (mc.getExtractionOptions().getSwapOnRead()) {
      maps_to_swap.emplace_back(mc.getMapName());
      is_enabled = true;
    }
  }
  /*If enabled, try enhanced compilation, otherwise basic as fallback*/
  if(!is_enabled) {
    config = {};
    logger->info("[Dynmon_CodeRewriter] No map marked as swappable, no rewrites
    performed");
  } else if(try_enhance_compilation(original_code, type, maps_to_swap, config)) {
    logger->info("[Dynmon_CodeRewriter] Successfully rewritten using
    PROGRAM_INDEX_SWAP technique.");
  } else if(do_base_compile(original_code, maps_to_swap, config)) {
    logger->info("[Dynmon_CodeRewriter] Successfully rewritten using
    PROGRAM_RELOAD technique.");
  } else {
    config = {};
    logger->info("[Dynmon_CodeRewriter] Error, no rewrites performed");
  }
}
```

6.2.3 MapExtractor

The *MapExtractor* class is responsible for the actual extraction of the gathered data from the respective eBPF maps into one of the two different export formats (JSON or OpenMetrics). This component presents a static method to be called from the instance of the service, the *Dynmon* class, which receives the user request and waits for the apposite results to be ready.

Internally, this component has different methods to be used according to the type of the map and its related information, like the contained value types. In fact, each map type is treated differently, since it may support different operations than the others. That said, every method uses the same utility functions to extract the “leaf" data once the map has been navigated: the effectiveness of *Dynmon* is given by the usage and export of data belonging to maps that the service does not know a priori. Interestingly, the service has been programmed to read and parse the eBPF table description object and act accordingly, calling a different method for each data type.

Another strength of this class is the support it provides for the newest eBPF map lately defined in the Linux kernel and the high efficiency of the low level extraction methods. According to the current kernel version the service is being used on, it enables/disables the so called “batch" operations, which are very high performing operation on maps to perform multiple actions at a time, instead of iterating over the map sequentially. This feature allows both the Control and Data planes to be more efficient, since access to shared maps takes less time in terms of locking mechanism and system calls.

Listing 6.3: Static main method of *MapExtractor*.

```
json MapExtractor::extractFromMap(BaseCube &cube_ref, const string& map_name,
            int index, ProgramType type,
            std::shared_ptr<ExtractionOptions> extractionOptions) {
  // Getting the TableDesc object of the eBPF table
  auto &desc = cube_ref.get_table_desc(map_name, index, type);

  if(desc.type == BPF_MAP_TYPE_HASH || desc.type == BPF_MAP_TYPE_LRU_HASH)
    return extractFromHashMap(desc, cube_ref.get_raw_table(map_name, index, type)
                        ,extractionOptions);
  else if(desc.type == BPF_MAP_TYPE_ARRAY)
    return extractFromArrayMap(desc, cube_ref.get_raw_table(map_name, index, type)
                        ,extractionOptions);
  else if(desc.type == BPF_MAP_TYPE_QUEUE || desc.type == BPF_MAP_TYPE_STACK)
    return extractFromQueueStackMap(desc, cube_ref.get_raw_queuestack_table(
                        map_name, index, type), extractionOptions);
  else if(desc.type == BPF_MAP_TYPE_PERCPU_HASH || desc.type ==
    BPF_MAP_TYPE_PERCPU_ARRAY || desc.type == BPF_MAP_TYPE_LRU_PERCPU_HASH)
    return extractFromPerCPUMap(desc, cube_ref.get_raw_table(map_name, index,
                        type), extractionOptions);
  else
    // The map type is not supported yet by Dynmon
    throw runtime_error("Unhandled Map Type " + std::to_string(desc.type) + "
    extraction.");
}
```

The Listing 6.3 reports the entry point of the *MapExtractor* class where, according to the retrieved map type, it is decided which internal extraction method to use. Moreover, in case the metrics is contained in a map not supported by any of these methods, an error is raised. However, this class provides support for the most used and common BPF maps, which are:

- arrays, the simplest linear data type whose data is accessible through an index;
- hash, a key-value map where the used key, which can be a structured user define type, is hashed to improve accesses;
- LRU (Least recently used) hash, a hash map with an algorithm applied which ensures that, when a new data needs to be inserted, the oldest entry in the map is substituted;
- PERCPU array, an array map for each CPU core (thus potentially an array of arrays), to avoid race conditions and locking mechanisms;
- PERCPU hash, a hash map for each CPU core;
- PERCPU LRU hash, an hash map with an LRU structure for each CPU core.

Another interesting snippet is provided in the Listing 6.4: the depicted method *recExtract()* is used to decide whether the current value to be extracted represents a primitive type or a more complex structure. The latter case would call recursively the same method until it reaches the leaf values of the structure (e.g., a “struct" would call *recExtract()* for each of its fields). And the end of the recursion, when therefore the value under analysis represent a “leaf" primitive type of the structure, the method *valueFromPrimitiveType()* is called, which will extract in JSON format the information according to its type (integer, char, float, etc.).

Listing 6.4: *MapExtractor* method to extract values.

```
json MapExtractor::recExtract(json object_description, void *data, int &offset) {
  // Identifying the object
  auto objectType = identifyObject(object_description);

  switch (objectType) {
  case Property: { // the object describes a property of a C struct
    auto propertyName = object_description[0].get<string>();
    auto propertyType = object_description[1];
    return propertyName, recExtract(propertyType, data, offset);
  }
  case ArrayProperty: { // the object describes a C array
    auto propertyName = object_description[0].get<string>();
    auto propertyType = object_description[1];
    auto len = object_description[2][0].get<int>();
    // array of primitive type
    if (propertyType.is_string()) // this string is the name of the primitive type
      return valueFromPrimitiveType(propertyType, data, offset, len);
      // array of complex type
    else {
      json array;
      for (int i = 0; i < len; i++)
        array.push_back(recExtract(propertyType, data, offset));
      return value_type(propertyName, array);
    }
  }
  case Value: // the object describes a primitive type
    return valueFromPrimitiveType(object_description.get<string>(), data,
                                  offset);
  case Struct: // the object describes a C struct
    return valueFromStruct(object_description, data, offset);

  case Union: // the object describes a C union
    return valueFromUnion(object_description, data, offset);

  case Enum: // the object describes a C enum
    return valueFromEnum(object_description, data, offset);
  }
  throw runtime_error("Unhandled object type " + std::to_string(objectType) + " -
    recExtract()");
}
```

6.3 Data Plane injection

As described in the previous chapter concerning the architecture, the core aspect of the *Dynmon* service is the capability to dynamically change the behaviour of the Data Plane, while the Control Plane performs the same operations independently by the underlying injected program.

The runtime injection is possible thanks to the usage of functions like *reload()*, which are accessible to all the Polycube services via inheritance of the *BaseCube* class. This method is responsible for replacing the current eBPF program at a certain index of the program type chain (e.g., the 2nd program in the Ingress hook), ensuring that the new one is correctly compiled. *Dynmon* uses this function every time a new Data Plane configuration is sent by the user, or in case the *CodeRewriter* class performs the BASE tuning to the code.

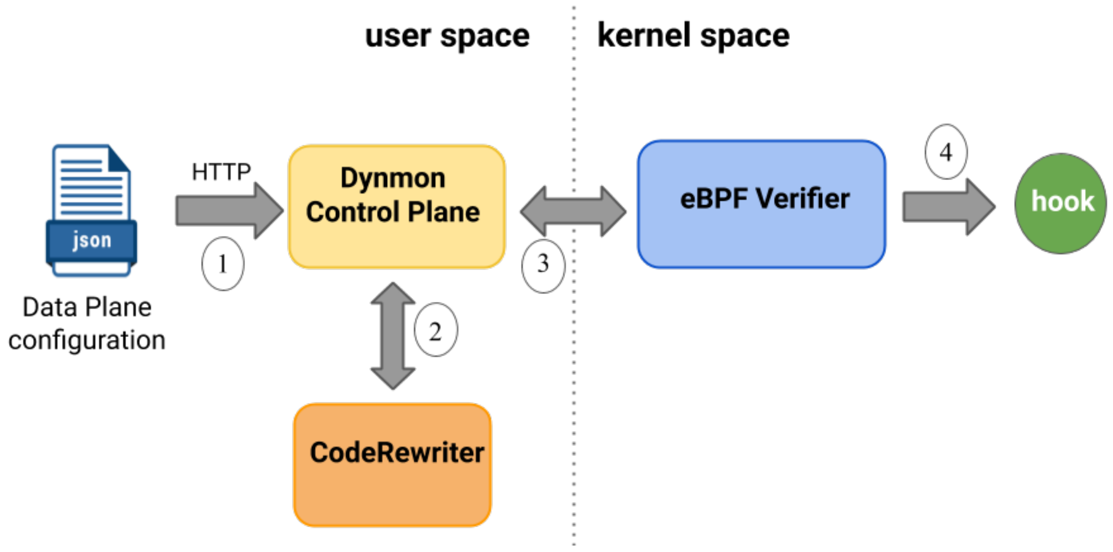


Figure 6.1: Data Plane injection workflow.

The Figure 6.1 depicts the typical workflow and all the main interactions between components when the user provides a new configuration. To start with, the Control Plane, once it has received the configuration via an HTTP POST request, forwards the new Data Plane to the *CodeRewriter* class which, according to the specified parameters, decides whether to perform or not some enhancement and returns the configuration to the Control Plane. At this point, the new artefact is forwarded to the *eBPF Verifier* in kernel-space, where it compiles and validates the code using the apposite *LLVM* compiler infrastructure, which produces the low-level executable to be injected in the system. If no error occurs during the compilation phase, the program is inserted in the apposite hook (Ingress/Egress).

6.4 Supported data extraction

One of the greatest feature of this service is the capability to export user-defined metrics in the JSON lightweight data-interchange, extracting values from the respective underlying eBPF maps independently by the contained data.

As previously described in Section 6.2.3, the main class involved in the extraction process is *MapExtractor*. In order to access the low-level eBPF map description structure, the class exploits the use of *TableDesc* class, provided by the *BCC* [19] library. Moreover, all the extraction methods use the handler functions provided by this library (which relies on *libbpf* [20]), as they both ease the system call invocations, and they are always up-to-date with the functions and structures provided by the kernel.

The *TableDesc* object of a specific map is obtained using the get_table_desc() method provided by the parent *BaseCube* class. This object contains both key_desc and key_desc, to JSON objects which describe the structure and type of the eBPF map's entries. These objects, are internally parsed, using apposite built methods to allocate variable to hold the result according to the apposite specified type. In fact, the key_desc and key_desc JSON objects, once correctly parsed, can describe C values as primitive data types (e.g., int, char, double, unsigned long long and more), enums, unions and structs.

In order to extract the eBPF map content, both key_desc and key_desc are parsed recursively to cast the memory block of the map's entries accordingly to their structure. The parsing recursion method recognizes these objects as one of the aforementioned data types, then calls the corresponding extraction method that will cast the obtained memory block of a map entry in order to extract the contained value.

Since the C structured objects have nested data types, for each one of their fields, the recursive method is called on each of them. This way, the entire structure is successfully extracted using only one method, rec_extract().

The Figure 6.2 depicts the extraction scenario. At first, the *MapExtractor* class identifies the type of the map using the *TableDesc* object. Then, if it discovers that the map type is supported, it uses the key_desc and key_desc elements to parse the entire map. Calling the recursive recExtract() method on each value, this class can identify the type of the node obtained and call one of the following apposite methods:

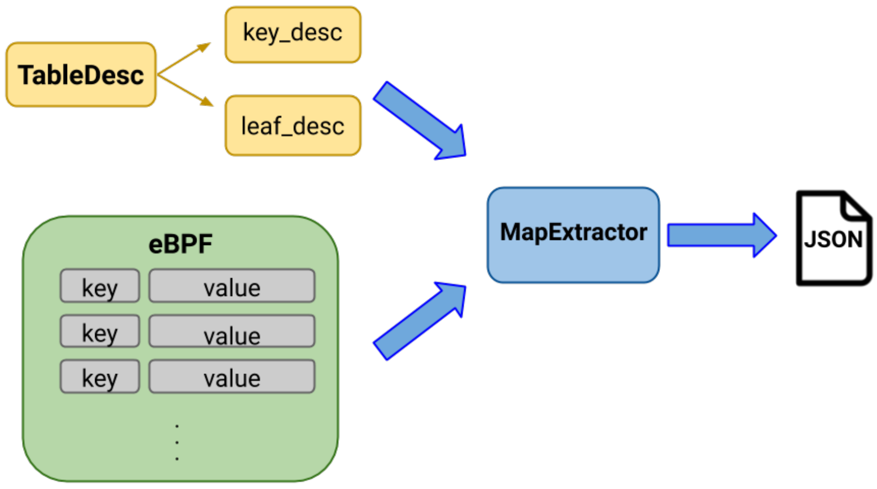


Figure 6.2: Map extraction pipeline.

- valueFromStruct(): this method parses an entire memory block corresponding to a C struct and produces a JSON object; this object is composed by the key-value pair, representing each property name and value belonging to the struct;

- valueFromUnion(): it parses a memory block corresponding to a C union. Unfortunately, since the real type of union is decided by the program and cannot be known at runtime, this method extract all the possible types contained in the union, in order to let the user decide which of the gathered values has to be used. Thus, the result corresponds to a JSON entity representing the union as a set of key-value pairs, where the key refers to the attribute type, and the value is the result of the memory casting operation to that specific type;

- valueFromEnum(): this method parsed a C enum memory block in order to extract the specific value of the enum.

- valueFromPrimitiveType(): it parses a memory block corresponding to a C primitive type (e.g., int, uint, float, etc.) and produces a JSON object containing the extracted value, appositely cast to the primitive type. In order to correctly parse and locate the primitive type, the method compares the one retrieved from the *TableDesc* fields (*key_value* and *leaf_value*), which is stored as a string, with some statically defined string (e.g., “unsigned int", “float", etc.).

A few example to better understand how the *TableDesc* class stores information concerning keys and values types are provided in Figure 6.3.

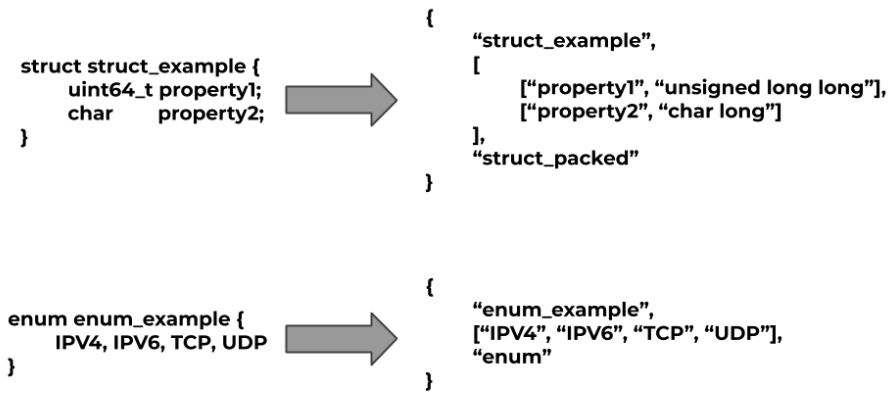


Figure 6.3: C struct and union to *key_type*/*leaf_type* JSON objects.

6.5 Timestamping

One of the most known limitations of eBPF is the lack of a precise and standard way of obtaining packets' timestamps. The structures corresponding to the raw packet are:

1. *sk_buff*, if the eBPF is loaded in Linux Traffic Classifier (TC) mode, meaning that it is attached to the hooks of the Linux networking TCP/IP stack;
2. *xdp_md*, in case the program is loaded in the XDP mode, that is directly injected in the network interface card.

Listing 6.5: *xdp_md* structure defined in the Linux Kernel.

```
struct xdp_md {
    __u32 data;
    __u32 data_end;
    __u32 data_meta;
    __u32 ingress_ifindex;
    __u32 rx_queue_index;
    __u32 egress_ifindex;
};
```

According to the Linux kernel, as shown in Listing 6.5, the *xdp_md* structure does not contain a field representing the timestamp of the packet. Thus, when injecting the program in XDP mode, there is no way to extract the timestamp from the packet structure. On the other hand, when the program is loaded in the TC mode, as reported in Listing 6.6, the *sk_buff* structure has a field representing the packet reception/sending timestamp (*struct skb_timeval tstamp*, assigned by the Linux networking stack).

Listing 6.6: *sk_buff* structure defined in the Linux Kernel.

```
struct sk_buff {
  struct sk_buff * next;
  struct sk_buff * prev;
  struct sock * sk;
  struct skb_timeval tstamp;
  .
  .
  .
  unsigned char * data;
  .
  .
  .
};
```

Although, the *tstamp* value has an undefined behaviour, meaning that sometimes it may contain the real packet timestamp in the Unix Epoch format, or it could contain an integer value retrieved by the kernel timer, or it could even be empty. Due to this unpredictable value, since the eBPF program must contain only vital instruction to avoid delaying the packet, it has been introduced in Polycube a helper to retrieve the time in the standard Unix Epoch format.

The function `pcn_get_time_epoch()`, defined in the *BaseCube* class, is a method accessible to all the eBPF programs in Polycube to compute the timestamp in Unix Epoch format. This method retrieves the value of a kernel timer calling the function *bpf_ktime_get_ns()*, and increment a base value *EPOCH_BASE* which is used as a reference. In fact, *EPOCH_BASE* represent the Unix Epoch time of the kernel timer, computed at a certain in the Control Plane, and later injected in the Data Plane. This way, the `pcn_get_time_epoch()` called from the Data Plane can simply add to the base reference value the kernel timer value of when the function is called, obtaining, as a result, the Unix Epoch time of that specific moment.

Despite the effectiveness of this method and the accuracy of the computed timestamp, there are few drawbacks to take into account, that hopefully are going to be fixed in the next Linux kernel release as announced:

1. *bpf_ktime()* and *bpf_ktime_get_ns()* are very expensive functions which, depending on the system they are called from (e.g., used in a VM can lead to worst results than in a normal system), may introduce few nanoseconds (40 ns) of overhead;

2. *bpf_ktime()* and *bpf_ktime_get_ns()* retrieve the value of the clock monotonic kernel timer, which, unfortunately, does not include the time spent by the system during *sleep*, *hibernate* and *suspend* states. Thus, until the device running Polycube and these eBPF programs is up and running the computed timestamp is coherent and real, but as soon as it suspends, the retrieved value will be misaligned. However, few enhancements and checks in the Control Plane can be introduced to re-align the value, adding the system inactivity time.

6.6 Dynmon Injector tool

The *Dynmon Injector* tool is a high level user-friendly Python REST client which communicates to a specified Polycube daemon to facilitate the creation of a *Dynmon* service instance and the immediate injection of a Data Plane configuration, without dealing with the Polycube command line tool. Moreover, the tool has been developed to address an important lack of the command line tool: the complex data type injection. In fact, while for some kind of operation the command line tool is really useful (e.g., retrieving service information and statistics), for others like a JSON file injection it turns out to be unusable.

The input parameters that the tool accepts are:

- *cube_name* (mandatory): the name of the *Dynmon* service instance to handle (and also create if it does not exist yet);

- *peer_interface* (mandatory): the name of the network interface (e.g., *eth0*) to which attach the service;

- *path_to_configuration* (mandatory): the local path to the Data Plane configuration to be injected;

- *address* (optional): the address where the Polycube daemon instance is listening (default is localhost);
- *port* (optional): the port where the Polycube daemon instance is listening (default is 9000);
- *version* (optional): prints the current tool version and exists.

When running the tool, it checks whether the service with the desired name exists or not. In case the tool does not receive a response, it means that there is not a Polycube daemon listening at the provided IP:PORT. Otherwise, if the service does not exist yet, a new instance is created. Finally, when exists an instance with the desired name, the tool contacts the daemon to attach the existent service instant to the specified network interface.

Once the initial operations are correctly performed, the tool sends the provided Data Plane configuration to the service instance Control Plane, to be parsed and injected in the system.

A flowchart to summarize the tool behaviour is provided in Figure 6.4

TheListing 6.7 provides an example of output both when creating a service instance that does not exist yet, and when attempting to creating an instance with the same name, but with a different Data Plane configuration file (the existent service is patched).

Listing 6.7: *Dynmon Injector* output example.

```
test@test:~$ ./dynmon_injector monitor eth0 dataplane.json

Creating new dynmon instance named monitor
Attaching monitor to wlp59s0

test@test:~$ ./dynmon_injector monitor eth0 dataplane2.json

Dynmon monitor already exist
Injecting the new dataplane
```

6.7 Dynmon Extractor tool

The *Dynmon Extractor* tool is another user-friendly tool developed in Python to contact a running Polycube daemon instance in order to extract gathered metrics.

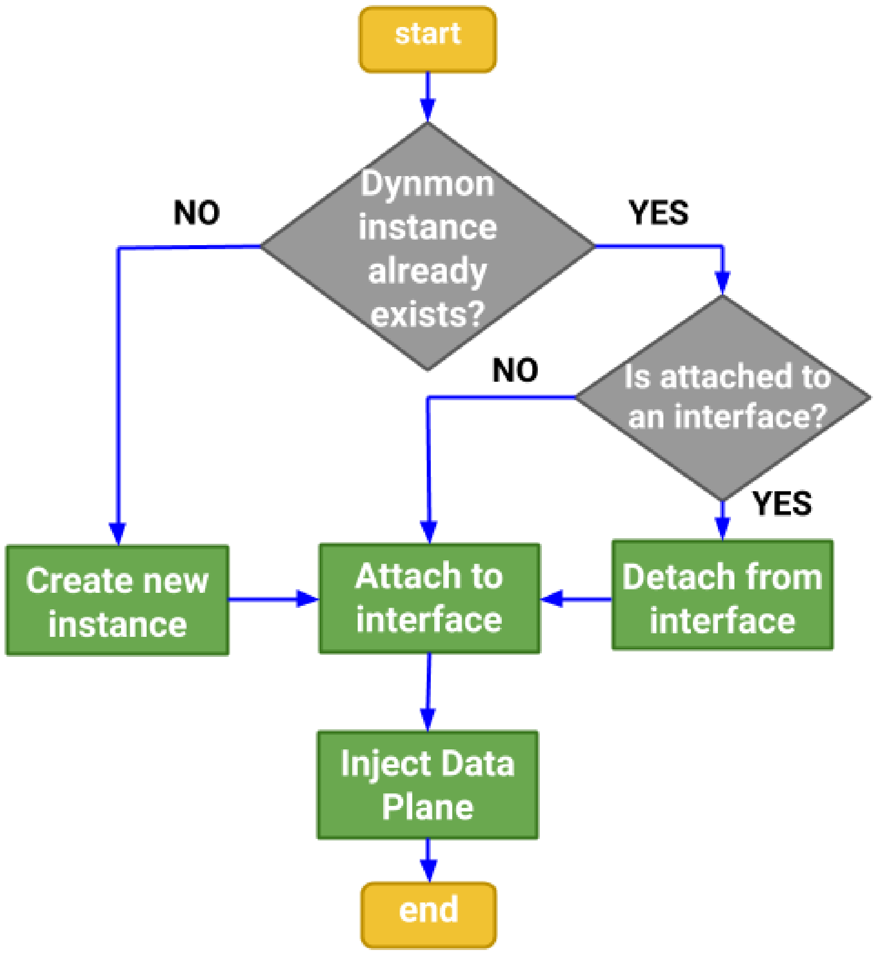


Figure 6.4: *Dynmon Injector* workflow.

This tool has been developed both to ease metrics extraction, and to address the Polycube command line tool lack of support for JSON objects output.

The input parameters that the tools accepts are:

- *cube_name* (mandatory): the name of the *Dynmon* service instance to handle (and also create if it does not exist yet);
- *address* (optional): the address where the Polycube daemon instance is listening (default is localhost);
- *port* (optional): the port where the Polycube daemon instance is listening (default is 9000);

- *save* (optional): this parameter specifies that the retrieved metrics must be stored in a file, by default named “dump.json";
- *type* (optional): the program type which the metrics should be extracted from (e.g., only the Ingress metrics), by default both Ingress and Egress defined metrics are extracted;
- *version* (optional): prints the current tool version and exists.

In order to provide an example of the tool output, considering the short configuration provided in Listing 6.8 as the Data Plane configuration to be injected in the service (the full code can be retrieved from the Polycube repository). This configuration specifies two programs to count packets passing through the specified interface.

Listing 6.8: Configuration used in the example.

```
{
    "ingress-path": {
        "name": "Ingress packets counter probe",
        "code": "...",
        "metric-configs": [
            {
                "name": "packets_total",
                "map-name": "PKT_COUNTER"
            }
        ]
    },
    "egress-path": {
        "name": "Egress packets counter probe",
        "code": "...",
        "metric-configs": [
            {
                "name": "packets_total",
                "map-name": "PKT_COUNTER"
            }
        ]
    }
}
```

After injecting the configuration using the *Dynmon Injector* tool, the metric *packets_total*, which is different between Ingress and Egress programs, can be retrieved as depicted in Listing 6.9.

Listing 6.9: *Dynmon Extractor* example.

```
test@test:~$ ./dynmon_extractor.py monitor
{
  "ingress-metrics": [
    {
      "name": "packets_total",
      "value": [19],
      "timestamp": 1599837827721618
    }
  ],
  "egress-metrics": [
    {
      "name": "packets_total",
      "value": [18],
      "timestamp": 1599837827721652
    }
  ]
}
```

Even though the Ingress and Egress metrics are defined with the same name, in this example the two “packets_total" retrieved are different, and they only refer to the packet passed through the specific hook. However, the user can build a more complex program with a unique eBPF map shared among the hooks, exactly like the monitoring developed for the TOSHI project presented in Section 7.

Chapter 7

TOSHI Infrastructure

This chapter analyses the architecture and, mainly, the implementation of the TOSHI project, previously presented in Section 1.2. In particular, the implementation presented concerns only the components that I developed, which exploits the newly realized *Dynmon* service to adapt to the different needed detection scenario.

However, all the components involved in TOSHI made by other partners are explained, to provide a complete high-level vision of the project.

7.1 Overview

The objective of the TOSHI product is to provide a high-performance, automated and adaptive defensive shield against cyber-attacks. TOSHI empowers Linux-based host machines (e.g., servers, computers, IoT gateways) with efficient kernel-based technologies that are self-adaptively configured to monitor and counteract security threats under the assistance of AI mechanisms.

TOSHI combines novel in-kernel Linux technologies such as eBPF/XDP with AI/ML techniques to analyse events, detect and mitigate anomalies on the host machines. As described in the following sections, these components work in conjunction with the Redborder cyber-security platform, which integrates functions for data collection, data analysis, threat detection and response.

TOSHI is the result of a collaboration of the following partners:

1. Politecnico di Torino (prof. Fulvio Risso and I), responsible for data generation and extraction using eBPF/XDP;

2. Fondazione Bruno Kessler[1] (FBK), in charge of the entire AI/ML module concerning DDoS detection;

3. Universidad Politécnica de Madrid[2] (UPM), responsible for the AI/ML module for Crypto Mining detection;

4. Telefónica[3], a Spanish multinational telecommunications company, which covers the DNS over HTTPS attack scenario, unfortunately not implemented in the MVP[4], and provides a test bed for the integration phase;

5. Innovalia[5], a Spanish private business unit, and RedBorder[6], a group within Innovalia, which are in charge of providing a broker component, to ease the integration between other components and their cybersecurity platform.

7.2 Architecture

A high-level representation of the TOSHI architecture is given in Figure 7.1. The system is coordinated by the RedBorder platform, which collects the output of the anomaly detection components executed on the host machines and takes decision about appropriate countermeasures. The logically centralized position of RedBorder (with respect to a TOSHI-enable administrative domain) enables both reactive and pro-active mitigation strategies against network threats through a coordinated deployment of traffic filtering policies aggregating the input coming from single host machines.

Most importantly, the centralized position of RedBorder ensures that coherent countermeasures policies are applied to the hosts, otherwise, if the two AI/ML components would take individual decision, potentially they could apply contrasting policies according to their analysis. Therefore, the need of a univocal decision has led to the use of this component, integrated with a broker to forward information

[1]FBK website `https://www.fbk.eu/en/`

[2]Universidad Politécnica de Madrid website `https://www.upm.es/`

[3]Telefónica company website `https://www.telefonica.com/en/`

[4]Minimum Viable Product, a version of a program with just few features to satisfy early customers and gather feedbacks.

[5]Innovalia Association website `https://innovalia.org/en/`

[6]RedBorder website `https://redborder.com/`

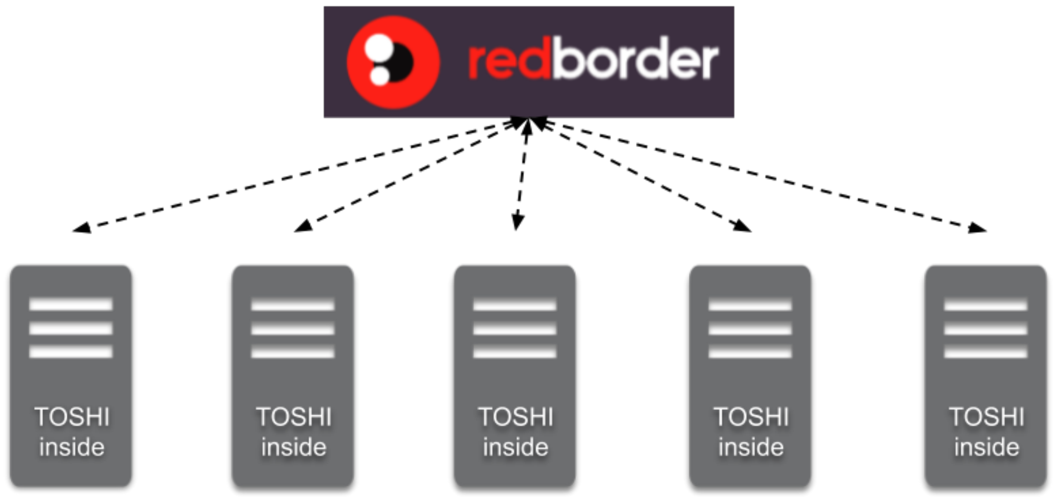


Figure 7.1: Toshi architecture overview.

from the components to RedBorder, exploiting standardized REST APIs to ease the management.

A more detailed view of TOSHI key elements and their interconnections is provided in Figure 7.2. The figure shows the internal modules of TOSHI-enable host machines and the interconnections among them and with the RedBorder platform.

The main objective behind TOSHI architectural choices is to build a solution that protects the host machines against network threats with minimal impact on the applications running on the machines themselves (e.g., web servers, cloud services, etc.). To this end, TOSHI exploits recent technologies such as eBPF, available in the kernel of recent Linux operating systems (kernel version $\geq$ 4.15). This new promising technology, is a kernel packet filtering/manipulation mechanism that can be used to implement network functions efficiently.

In the TOSHI architecture, it is adopted the Polycube framework to manage the functions deployed in the kernel space in the form of chains of eBPF programs. These functions (e.g., firewalling, packet sampling, feature extraction, etc.) are executed on the network traffic before it reaches the network stack of the operating system. This allows TOSHI to inspect the traffic and to filter the malicious packets at the earliest stages, hence saving both CPU and memory resources of the system.

Decision on how to handle the malicious traffic are taken into account by the so-called TOSHI applications running on the user space of the host machines.

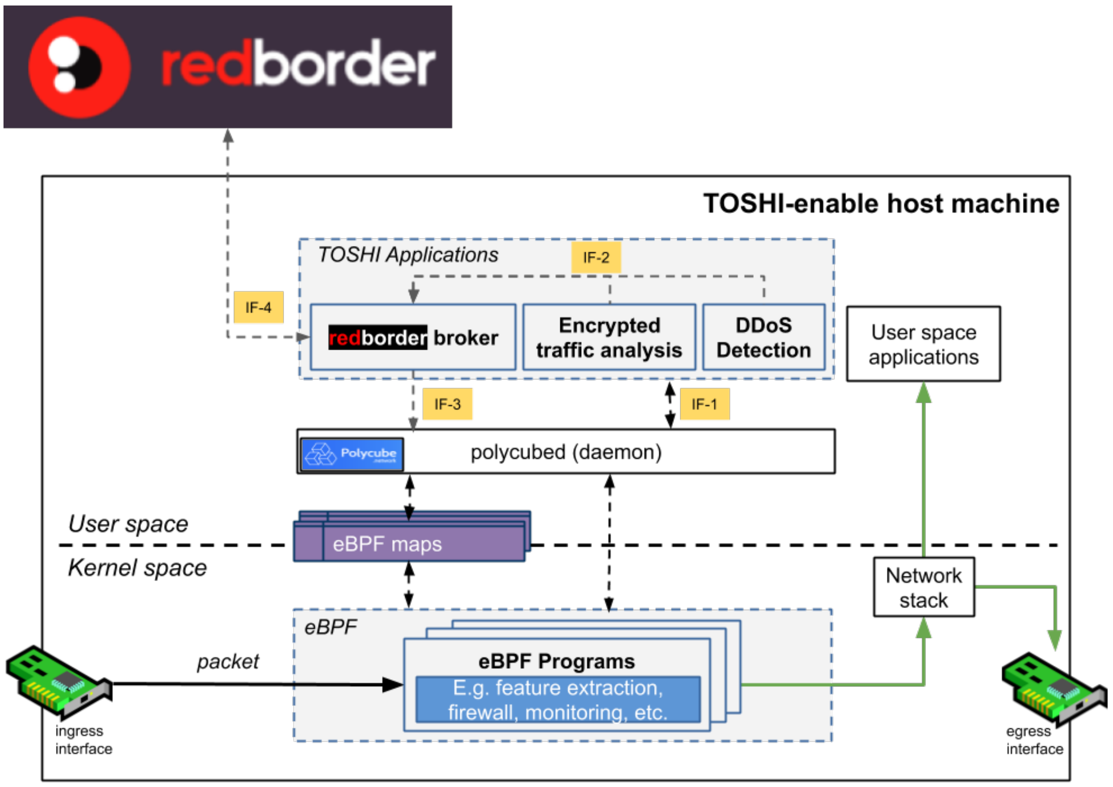


Figure 7.2: Toshi detailed architecture.

TOSHI applications interact with the network traffic through the interfaces (*IF-1* and *IF-3* in Figure 7.2) exposed by the Polycube daemon. Interface *IF-1* enables the applications to query the Polycube daemon for gathering the traffic information so far collected by the eBPF programs in the kernel space and saved in the eBPF maps.

Security policies produced by the TOSHI applications (e.g., traffic filtering, traffic re-routing), which are produced after the analysis of the network traffic, are injected into the kernel via interface *IF-3*. Specifically, these rules are injected into the Polycube Firewall service, which stores them into eBPF maps to be used in the Data Plane. One or more eBPF programs match the incoming packets against these rules and take specific actions if the match is positive. For instance, traffic filtering rules can be defined to drop the packets matching one of the source IP addresses listed in the map.

As described in the following sections of this document, TOSHI applications comprise a DDoS detection module, an application for Crypto-mining attacks detection, and one for DoS over HTTPS attack detection. Once retrieved traffic information from Polycube, these modules produce a set of traffic policies that are necessary to mitigate the specific type of network threat they are implemented for. Before being deployed in the kernel of the machine, these policies are sent to the RedBorder using the apposite broker (communications between ML modules and the broker via *IF-2*), for analysis, approval and aggregation, through the *IF-4*. In response, the broker might decide whether to ignore the policies because unnecessary, conflicting with other policies or already installed on the TOSHI-enable host. On the contrary, RedBorder might aggregate the policies coming from multiple TOSHI-enable hosts and send back (to all the hosts or a subset of them) a set of combined rules to be inserted in the apposite eBPF maps.

The role of the broker is to provide a user-space standardized communication channel to ease connectivity and communication between the different security applications, Polycube and RedBorder.

To sum up, the main qualities and features of TOSHI are:

1. automatic incident response;
2. holistic protection (devices can be computers, switches, routers, etc.);
3. autonomous and detached operations;
4. resource efficient;
5. high networking speed management.

7.3 Artificial Intelligence support

As previously described, the security modules are user space applications that, given the gathered metrics as input, produce policies to be possibly applied to the firewall service. To enhance the analysis, these modules exploits many AI/ML algorithm and structures, like neural networks or heuristic algorithms.

Despite being widely used in many fields, a machine learning model can significantly improve cybersecurity threats detection, combining powerful algorithms able to classify input data. In particular, a neural network, which is a very complex multi-layer structure similar to a human brain neuron, can be trained with a pre-defined dataset containing, for instance, the traffic captured during a DDoS attack (thus labelled as malicious), in order to train the entire structure to automatically analyse and decide the label for all the following inputs using correlations with the training dataset.

An interesting application of machine learning in cybersecurity is provided by LUCID [21], which is the technology used for DDoS attack detection also in the TOSHI project. The Figure 7.3 depicts its architecture.

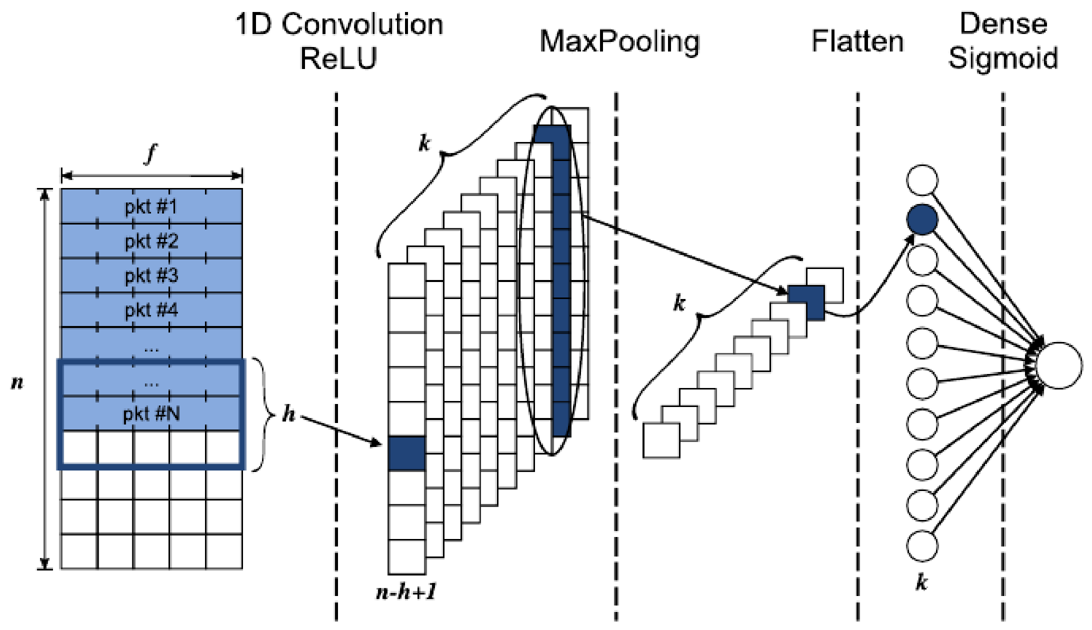


Figure 7.3: LUCID neural network architecture.

LUCID is a lightweight *Convolutional neural network (CNN)*, which shares the same parameters of CNNs used in Natural Language Processing, despite having different weights adjusted to the cybersecurity scenario. There are two main benefits of including a CNN based architecture. Firstly, it allows the model to benefit from efficiency gains compared to standard neural networks, since the weights in each filter are reused across the whole input. Secondly, during the training phase, the CNN automatically learns the weights and biases of each filter such that the learning of salient characteristics and features is encapsulated inside the resulting model during training.

The Crypto-mining attack detection module exploits several ML models, using Random Forest and Fully Connected Neural Networks to analyse the traffic. Once the models have been trained with apposite malicious traffic, they are able, given an array-like representation of flow states as input, return a value between 0 and 1. This value represents the probability P of a given input to be part of a cyberattack. Once decided the best value for the threshold T, all the inputs with probabilities P $\geq$ T will be considered malicious.

7.4 Dynmon probes for TOSHI

As already described, the eBPF programs are managed by the Polycube framework, in particular the Dynmon service. This service provides a network monitor able to handle dynamically injected user configuration to adapt network monitoring, filtering and forwarding.

In the TOSHI scenario, Dynmon turns out to be the perfect candidate to manage multiple probes, as it automatically adapts the Data Plane to the user needs. In fact, two different programs has been developed, one for each scenario presented in the MVP (the DoS attacks over DoH channels has not been presented yet).

The two program aims at extracting pre-defined features from the traffic, according to the specific use case. Moreover, each use case needs to take into account only certain protocols (e.g., UDP and TCP, but not ICMP), to improve the detection mechanism by pruning the unneeded one.

7.4.1 DDoS attack detection

The probe created for the DDoS attack use case filters all the IPv4 traffic carrying TCP, UDP or ICMP protocols. For each of these packets, the probe extracts the values that identify the specific connection:

- source IP address;
- destination IP address;
- source port (for ICMP is 0);
- destination port (for ICMP is 0);
- protocol type.

Once retrieved the connection identifier, the probe inserts or update a specific eBPF map where all the connections so far analysed are held. Then, if the number of packets captured for that specific session has not reached a threshold value, a few features of the current packet are extracted and pushed into a shared queue, ready to be extracted from the Control Plane program.

Despite this approach is innovative and effective, the performance could be constrained by two main factors. Firstly, sharing eBPF maps both between Ingress/Egress programs and between all the current available CPUs significantly slows down the networking capacity of the device, since all the cores computing incoming and outgoing packets have to wait for the map to be accessible (locking mechanism). Unfortunately, many constraints like the maximum packets analysed per-session do not allow using PER-CPU maps, as each core will not have entirely the current status of the capture (e.g., it will not be able to know if the current packet must be ignored or not). Secondly, the shared queue is a new data structure that I personally helped to improve, but it does not support PER-CPU usage yet. Thus, even though a queue is really performant, it cannot support high network traffic yet (e.g., 40 Gbit/s). However, the proposed probe has been optimized, in order to run monitoring code without harming the system. In fact, once the maximum analysed packet is reached, the program basically does not perform heavy instructions, since the shared queue cannot accept more packet. As a result, these parameters introduced ensure the efficiency of the monitoring program also with high speed, as it is reported in the experimental validation provided in Section 8.

Furthermore, it is worth considering that even all the others network traffic filtering software do not support high networking speed. On the contrary, they turn out to be even slower (e.g., Wireshark or NetFlow) as they rely on libraries that copy the entire matched packet in user-space, which of course it is way more expensive in terms of CPU time rather than taking few values and inserting them in kernel map.

The features extracted from each packet are displayed in Listing 7.1.

Listing 7.1: DDoS attack scenario packets features.

```
/*Features to be exported*/
struct features {
    struct session_key id;          //Session identifier
    uint64_t timestamp;             //Packet timestamp
    uint16_t length;                //IP length value
    uint16_t ipFlagsFrag;           //IP flags
    uint16_t tcpLen;                //TCP payload length
    uint32_t tcpAck;                //TCP ack number
    uint8_t tcpFlags;               //TCP flags
    uint16_t tcpWin;                //TCP window value
    uint8_t udpSize;                //UDP payload length
    uint8_t icmpType;               //ICMP operation type
} __attribute__((packed));
```

The Listing 7.1 provides a snippet of the main logical section of this probe. It is reported only the TCP analysis, but both the other two protocol have similar instructions, except the `struct session_key key = {...}` which strongly depends on the protocol ports.

Listing 7.2: TCP packet analysis for DDoS.

```
switch (ip->protocol) {
    case IPPROTO_TCP: {
      struct tcphdr *tcp = data + sizeof(struct eth_hdr) + ip_header_len;
      if ((void *) tcp + sizeof(*tcp) > data_end) {
        return RX_OK;
      }
      struct session_key key = {.saddr=ip->saddr, .daddr=ip->daddr,
            .sport=tcp->source, .dport=tcp->dest, .proto=ip->protocol};
      if(check_or_try_add_session(&key, curr_time) != 0) {
        return RX_OK;
      }
      uint16_t len = bpf_ntohs(ip->tot_len);
      struct features new_features = {.id=key, .length=len, .timestamp=curr_time,
        .ipFlagsFrag=bpf_ntohs(ip->frag_off),
        .tcpAck=tcp->ack_seq, .tcpWin=bpf_ntohs(tcp->window),
        .tcpLen=(uint16_t)(len - ip_header_len - sizeof(*tcp)),
        .tcpFlags=(tcp->cwr << 7) | (tcp->ece << 6) | (tcp->urg << 5) |
            (tcp->ack << 4) | (tcp->psh << 3)| (tcp->rst << 2) |
            (tcp->syn << 1) | tcp->fin};
      PACKET_BUFFER_DDOS.push(&new_features, 0);
      break;
    }
    case IPPROTO_ICMP: {
        ...
    }
    case IPPROTO_UDP: {
        ...
    }
    default : {
      return RX_OK;
    }
  }
```

7.4.2 Crypto-mining attack detection

The probe created for the Crypto-mining scenario, differently than the previous one, exploits the use of the most efficient data structures available in eBPF. That is possible as the neural network created for detecting such attacks is based on statistics and aggregated features, and not on the entire list of features for each packet. As a result, this probe is able to handle higher traffic speed than the DDoS attack detection one.

To start with, the program identifies the session identifier as in the previous case, by extracting IPs, ports and the protocol type from the packet header. The only two protocols taken into account for the Crypto-mining attack are TCP and UDP, carried over IPv4. Once retrieved the identifier, the program checks if an already existent value for that specific key is already present in the used eBPF map. If the key is missing, then a new value starting from the current packet's features is created. Otherwise, it looks for old connections identified by the same key, meaning that the map already contained that key, but the value has not been updated for a certain period, defined as SESSION_DROP_AFTER_TIME. In case the already existent value is older, it is completely overwritten, otherwise it is just updated adding the current packet ones.

Using a PER-CPU eBPF map allows the program to be really efficient, since every CPU core which is handling a packet does not have to wait for the shared map lock to be released, and the map can also be shared among Ingress and Egress programs for the same reason: every CPU has an own map lock-free. However, using a PER-CPU map does not allow referring to a unique structure in order to lookup some common values that might be useful, like the timestamp of the last received packet for that specific session. Every CPU has to handle its own, but since the features include both timestamp and other useful values to understand whether they are old or recent, when the maps contents are retrieved from the Control Plane it is quite easy to parse it and understand which value can be aggregated or has to be discarded since too old. That said, the PER-CPU map ensure high performance, even with high amount of packets to be inspected, since the probe needs to inspect all of them with no restrictions. The results reported in Section 8 confirm the effectiveness of this approach.

A complete list of features is provided in Listing 7.3. Depending on the server_ip, the fields referring to the Ingress and Egress programs are converted into client/server in the Control Plane. Moreover, the duration of the connection is computed by simply subtracting the start_timestamp value to alive_timestamp.

Listing 7.3: Features gathered for Crypto-mining scenario.

```c
/*Features to be exported*/
struct features {
    uint64_t n_packets_ingress;                    // Number of Ingress packets
    uint64_t n_packets_egress;                     // Number of Egress packets
    uint64_t n_bits_ingress;                       // Total Ingress bits
    uint64_t n_bits_egress;                        // Total Egress bits
    uint64_t start_timestamp;                      // First timestamp
    uint64_t alive_timestamp;                      // Last timestamp
    uint32_t  method;                              // Heuristic method
    __be32 server_ip;                              // The server address
} __attribute__((packed));
```

Another important issue that needed to be addressed in this scenario was the server identification within the connection; both the Ingress and Egress programs, whenever receiving a new session or an already existing session that has to be overwritten since too old, try to guess which of the two IP addresses belongs to the server, as it is an important factor that the neural network takes into account. There are different scenarios that needs to be considered:

- the program intercepted the 1st packet of a session: in this case it is still not possible to confirm that the receiver corresponds to the server, as it could be possible that the server itself opened a new connection with an already existent client but on a different port;
- the program intercepted a packet in between of an already established session: unfortunately, there is no way to certainly detects which of the two addresses is the server, unless digging the payload, but it is not suggested in eBPF.

For each of these two scenarios there are few sub-scenarios, that worsen the problem. To address all these cases, a performant heuristic algorithm has been introduces in the eBPF program, as depicted in Listing 7.4.

Listing 7.4: Heuristic methods for server detection.

```c
/*Method to determine which member of the communication is the server*/
static __always_inline __be32 heuristic_server_tcp(struct iphdr *ip, struct tcphdr
     *tcp, uint32_t *method) {
  /*If Syn, then srcIp is the server*/
  if(tcp->syn) {/*If source port < 1024, then srcIp is the server*/
    *method = 1;
    return tcp->ack? ip->saddr : ip->daddr;
  }
  uint16_t dst_port = bpf_htons(tcp->dest);
  /*If destination port < 1024, then dstIp is the server*/
  if(dst_port < 1024) {
    *method = 2;
    return ip->daddr;
  }
  uint16_t src_port = bpf_htons(tcp->source);
  /*If source port < 1024, then srcIp is the server*/
  if(src_port < 1024) {
    *method = 2;
    return ip->saddr;
  }
  *method = 3;
  /*Otherwise, the lowest port is the server*/
  return dst_port <= src_port ? ip->daddr : ip->saddr;
}
```

Three different identification methods have been designed:

1. Method 1: it is used only for TCP traffic (UDP only has Method 2 and 3), and consists in looking for the SYN flag. If only the TCP SYN flag is set, then the destination IP address is the server, otherwise if also the ACK flag is set, then the source address is the server, which is answering to the 3-way handshake protocol.

2. Method 2: this method try to identify the server by looking at the source and destination ports. The IP corresponding to the port that has a value lower than 1024 (well-known ports) is recognized as the server.

3. Method 3: this is left as the last option, and defines the server as the IP address which is using the lowest port. Presumably, a client during a connection uses a random dynamic port (49152 to 65535) free of its system, and it is not that common that such ports are used by a server.

Finally, in the Listing 7.5 is reported the main extraction section of a TCP packet, where all the so far discussed method are applied.

Listing 7.5: TCP packet analysis for Crypto-mining.

```
switch (ip->protocol) {
    case IPPROTO_TCP: {
      struct tcphdr *tcp = data + sizeof(struct eth_hdr) + ip_header_len;
      if ((void *) tcp + sizeof(*tcp) > data_end)
        return RX_OK;
      struct session_key key = {.saddr=ip->saddr, .daddr= ip->daddr,
            .sport=tcp->source, .dport=tcp->dest, .proto=ip->protocol};
      uint64_t curr_time = pcn_get_time_epoch();
      uint16_t pkt_len = bpf_ntohs(ip->tot_len);
      struct features *value = SESSIONS_TRACKED_CRYPTO.lookup(&key);
      if (!value) {
```

```
      pcn_log(ctx, LOG_DEBUG, "INGRESS - TCP New session");
      uint8_t method;
      __be32 server = heuristic_server_tcp(ip, tcp, &method);
      insert_new_session(server, curr_time, pkt_len, server==ip->saddr,
                                                      &key, method);
      break;
    }
    if(curr_time - value->alive_timestamp > SESSION_DROP_AFTER_TIME) {
      pcn_log(ctx, LOG_DEBUG, "INGRESS - TCP Session overwritten");
      uint8_t method;
      __be32 server = heuristic_server_tcp(ip, tcp, &method);
      update_expired_session(server, curr_time, pkt_len, server==ip->saddr,
                                                      &key, method);
      break;
    }
    update_session(value, pkt_len, curr_time, value->server_ip==ip->saddr);
    pcn_log(ctx, LOG_DEBUG, "INGRESS - TCP Session updated");
    break;
  }
  case IPPROTO_UDP: {
      ...
  }
  default :
    return RX_OK;
}
```

Chapter 8

Experimental validation

This chapter aims at providing experimental results concerning the performance of the service both in general and in a more specific use case. In fact, other than measuring CPU time taken by some basic operations in a general and simplified scenario using the C++ *std::chrono* library (*high_resolution_clock*), it is also presented the outcome achieved in the TOSHI project, where the monitoring programs are way more complex, and they exploit some of the most advantage features provided by the service (e.g., map atomicity).

Finally, to give an idea of *Dynmon* compared to similar networking programs, this chapter provides a comparison between *NetFlow* and a specific probe, which has been developed to act exactly like the other tool.

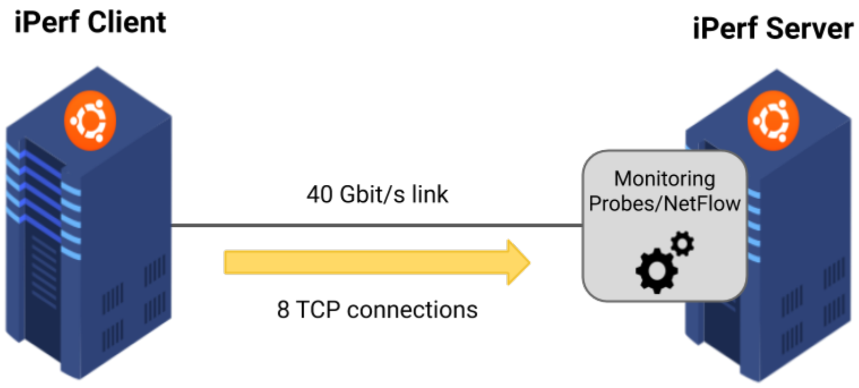


Figure 8.1: Client-Server tests set up.

The Figure 8.1 depicts the set up used for the tests. Even though the connection between the two machine supports 40 Gbit/s, the maximum bitrate measured in TCP connection between the two system under stable condition is 36.62 Gbit/s. The server runs an Ubuntu Server 18.04.3 LTS kernel 4.15.0-88-generic and 5.7.0-050700-generic x86_64 distribution, and it is equipped with an Intel(R) Xeon(R) CPU E3-1245 v5 3.50 GHz processor (4 cores and hyper-threading, 8 MB of L3 cache), 64 GB DDR4 RAM. Moreover, it runs the latest Polycube Docker, published September 16th and updated to the repository commit `770d0457`.

The tools used to create high-speed TCP connections and measure resources' availability are *iPerf2* [22] and *sar* [23]. While iPerf2 is used to create both TCP servers and clients communicating at the maximum speed reachable through the cable connection, sar is used in some tests to measure CPU consumption, in order to later compare data and comment the overhead added by some program.

8.1 Dynamic injection of monitoring code

To measure the time taken by the dynamic injection of user-defined monitoring code, many tests have been conducted, but only with the Ingress hook, since the same operations would be performed on the Egress, thus the resulting time taken would be twice. Therefore, for simplicity, only half of the configuration file has been injected, specifying different parameters in order to test as many cases as possible.

The first test aims at measuring the time taken by *Dynmon* to inject, once received, the Data Plane configuration, when declaring only one eBPF Array map with different entries, from 1 to 5000. Also, the same test has been repeated with the parameter “swap-on-read" enabled on the eBPF map, in order to trigger the class *CodeRewriter* and perform, alternatively, both the two rewrite types. This first experiment is expected to point out a constant injection and compilation time, since the number of entries should only affect the memory space assigned to the eBPF map, and not the entire injection process.

As depicted in Figure 8.2, it is quite clear that the number of entries in the eBPF map does not affect at all the Data Plane compilation and injection, independently by the optimizations introduced. In fact, on average this operation takes 0.195 s for the normal program injection, 0.207 s when optimizing with PROGRAM_RELOAD, and 0.389 s when using the PROGRAM_INDEX_SWAP feature. Changing the number of entries of the eBPF maps does not affect at all the compilation time,

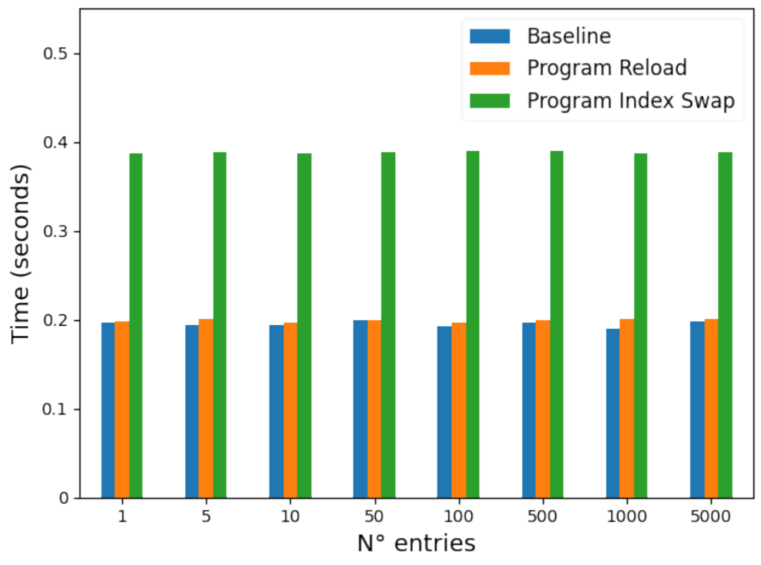


Figure 8.2: Data Plane injection with different eBPF map entries.

which is constant for both the three programs, even when using a 5000 entries map. However, if the PROGRAM_RELOAD rewrite leads to an injection time quite similar to the one measured when no optimization are performed, when using the PROGRAM_INDEX_SWAP feature takes twice the time to compile the program. This difference is due to the various controls, code modifications and the pivoting program injection, which of course slow down the entire process. Although, even though the injection process is not as fast as the other cases, using such optimization leads to a remarkable advantage every time the metrics have to be exported, as it will be displayed in the results provided in the following section.

On the other hand, while increasing the entries of a map has no effect on time, the second test presents the injection of different Data Planes, where only the number of eBPF Array maps vary, while the number of entry per-map is constant and equal to one. This time, the expected result is the opposite, since every map declaration requires multiple operations to be performed from the compiler, which has to instantiate and store information concerning all the structures.

The Figure 8.3, whose Y axes has been represented using a base 10 logarithmic scale in order to show all the values, clearly confirms the expected behaviour. As the number of maps grows, the required time increases exponentially, hitting a peak of 13.2 s when no rewrites are performed, 155 s for the PROGRAM_RELOAD

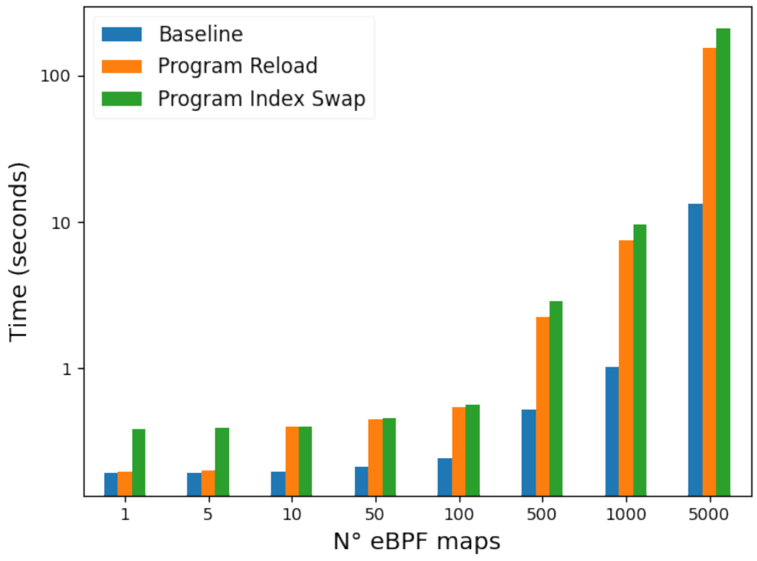


Figure 8.3: Data Plane injection with different eBPF maps.

rewrite, and 209 s when using the PROGRA_INDEX_SWAP enhancement, with 5000 eBPF maps. Clearly, this test has been performed only to measure the performance of the service, but it does not reflect a real life scenario. In fact, it is unlikely to have more than 10 eBPF maps in the same program, even the Polycube framework itself contains a lot of services which declare a dozen of maps at most. To summarize, even in this scenario the PROGRAM_INDEX_SWAP requires more time to compile the program, while the other two options presents a similar result for small eBPF maps. As soon as the number of maps increases (e.g., more than 10), the time taken by a normal program injection grows as well, but not as quick as the other two types, which turn out to require more than twice the time for the entire process.

8.2 Extraction of metrics

While the previous section presented tests concerning the Data Plane injection, this section covers the metrics extraction, which is way more important, since potentially is the most required operation that a user could perform, and consulting the maps may affect both the HTTP response time and the underlying injected eBPF program, which keeps executing its routine while the Control Plane needs to consult the same maps.

Before starting, it is quite important to remember that the extraction time is strongly related to the complexity of the data to be extracted, meaning that parsing a primitive value (e.g., int and unsigned) requires way less time than a structured one (e.g., struct and union). The following tests have been performed consulting an eBPF hash map with a variable number of entries, which are defined as:

- key: the 5-item tuple identifying a specific connection (source IP, destination IP, source port, destination port, protocol type);
- value: the entire TCP header (C struct containing 8 primitive fields), to cover a more complex scenario.

As most of the time is taken by the service to parse the raw value extracted from the map, these tests highlight the total execution time of the HTTP request, and the mere key-value lookup from the Control Plane.

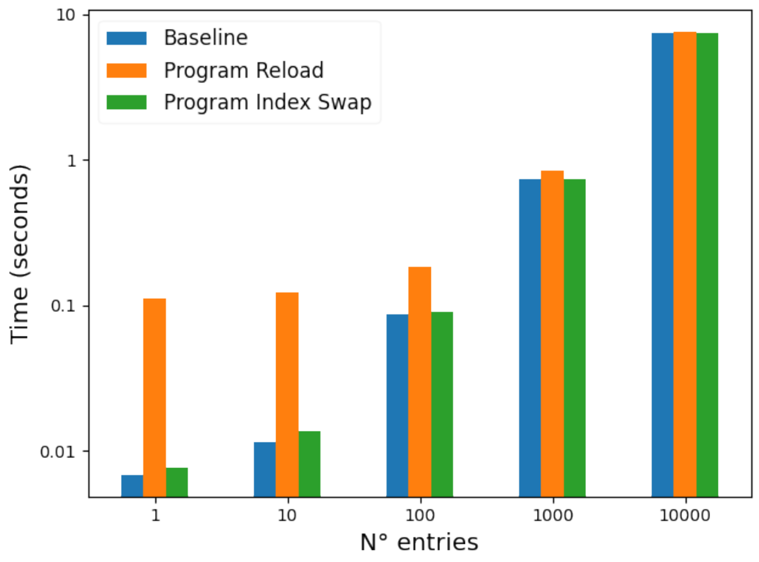


Figure 8.4: Extraction time of different TCP headers and program enhancements.

The Figure 8.4 shows the HTTP GET request total execution time, where all the variable keys-values belonging to the map are required. The program under test has been compiled and run both with no optimization, PROGRAM_RELOAD and PROGRAM_INDEX_SWAP rewrites, to mainly verify the efficiency of the enhanced method over the other. The advantage of using the latter feature when extracting metrics is relevant, compare to the other rewrite performed. In fact, this mode introduces only almost 4 ms of overhead with respect to the normal use case, where no map are declared as swap. On the other hand, the PROGRAM_RELOAD functionality takes 10 times more to extract the metrics, since every time they are requested the respective copy of the program containing the fictitious maps has to be re-inserted in the system. However, as soon as the number of entries in the eBPF map increases, the difference between all the 3 methods is negligible, as most of the time is taken by the C++ Control Plane to convert the data from raw pointers to JSON.

Considering both the complexity of keys and values (TCP headers have a lot of fields), the service is really performant up to a certain number of entries to extract, while the effects of having a generic method to parse all the possible eBPF maps are evident when using huge maps. Interestingly, for all those configurations which define only primitives and simple metrics to be extracted (e.g., the number of packet captured), the overhead introduced by the conversion from raw pointer to C++ data type and then JSON can be overlooked, and the charts would depict a linear dependency between the number of entries and the execution time.

The following test aims at underlying the clear difference between the mere map lookup performed by the Control Plane while using standard system calls and enhanced ones, later introduced into the Linux Kernel as “batch" operations. The main difference between these two operation is that the latter recalls only one system call, which encloses all the individual operations in order to perform the CPU context switch only once, saving precious time.

In fact, as represented in Figure 8.5, the batch operations introduces a significant advantage, speeding up the entire process of more than x10 factor, independently by the number of entries to lookup. The chart represents both the axes using a base 10 logarithmic scale, and the result is a directly proportional linear dependency between the number of entries and the required lookup time. Unfortunately, this advantage becomes negligible if the entire extraction and parse time is considered, as the previous chart (Figure 8.4) reports, since the HTTP response generated by the service takes almost 100 times more to be produced, due to the *Dynmon* recursive and generic parse methods.

However, as explained in Section 9.1, once the map key-value parse method is optimized using different techniques (BTF or parse method caching), the introduction of the batch operation will not be vain, but, on the contrary, will assume a lot

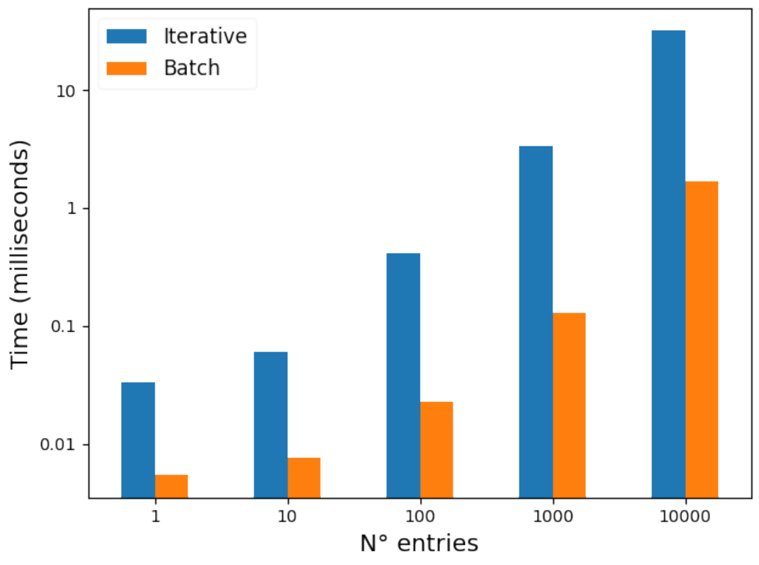


Figure 8.5: Iterative and batch extraction time of TCP headers.

of importance.

8.3 TOSHI performance

Dynmon is the service used within the TOSHI project, in order to extract metrics from packets and forward them to the user-space neural network, ready to analyse the traffic and decide whether some flow is malicious or not. For that purpose, two different probes has been injected in the system, one for each scenario taken into account (DDoS and Crypto-mining attacks). These probes have been inserted in sequence, and the final architecture is depicted in Figure 8.6.

The Crypto-mining probe has been injected as the last eBPF program executed at every packet, since it gathers statistics and it is extremely useful to check, during the experiment, if the entire system running the probes was able to handle all incoming and outgoing packets at high speed. The server is running 4 instances of iPerf2, in order to distribute connections among all the available cores. From a different machine in the same network, 4 iPerf2 clients each one with two parallel connections send TCP traffic to the server through the interface *eth0*. The link between client and server and their respective interfaces support 40 Gbit/s network speed.

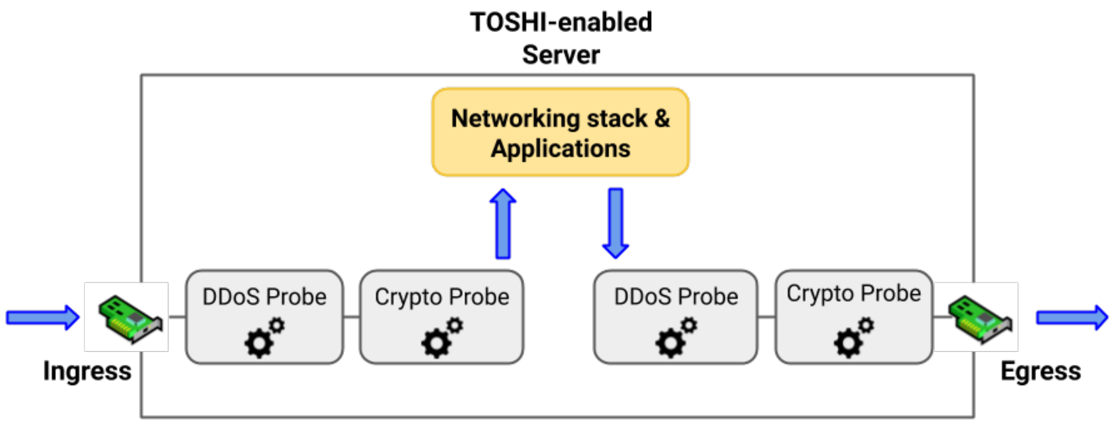


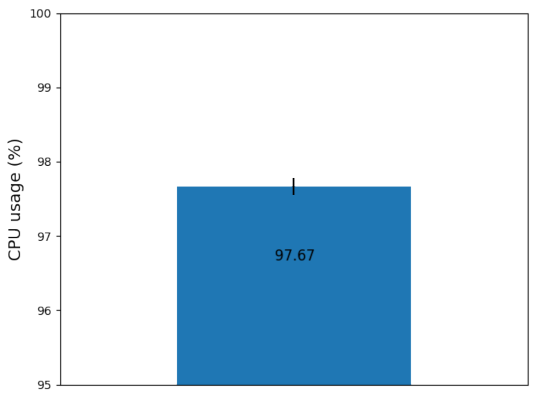
Figure 8.6: Architecture of the TOSHI experiment.

For the entire 60 seconds of connections between iPerf2 clients and server, the server has also been monitored using *sar*, storing results into a specific file later consulted. The collected statistics both from the tool and from the various iPerf2 servers are:

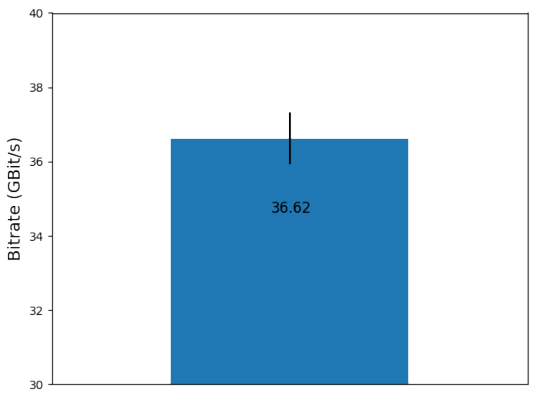
- Bitrate: 36.62 Gbit/s
- Total bytes transmitted: 274.7 GB
- Average CPU consumption: 97.7%

These results are really encouraging, as the system was able to handle high speed network traffic, even though almost all the CPU has always been busy computing packets, but at least there is no packet loss. In fact, the TOSHI probes turned out to be extremely efficient, due to their advanced monitoring intelligence, which ensures that the Crypto-mining probe captures all the traffic efficiently using PER-CPU maps, while the DDoS probe handle only a limited amount of packets as the neural network has been appositely trained to take decisions also with lower traffic information. In Figure 8.7 are illustrated these results with their respective confidence intervals, to better understand the efficiency of the probes.

In order to check whether the probes have correctly gathered all this amount of traffic, the two respective Python scripts have been used, in order to extract metrics from the two probes, post-processing data (e.g., convert timestamp into human-readable date, IP from integer to string, and many other) and store them.



(a) Average CPU consumption.



(b) Average Bitrate.

Figure 8.7: TOSHI results.

Listing 8.1: TOSHI probes metrics extraction output.

```
Crypto-mining - Got something!
        Time to retrieve metric: 0.02627873420715332 (s)
        Time to parse: 0.0004508495330810547 (s)
        Metric parsed: 8

DDoS - Got something!
        Time to retrieve metrics: 0.3696458339691162 (s)
        Time to parse: 0.0031197071075439453 (s)
        Packet parsed: 800
```

The output provided in Listing 8.1, suggests that the probes have correctly detected all the 8 connections (4 clients with 2 connections each) created. Moreover, the DDoS probe has successfully extracted the first 800 packets it detected, ignoring all the following ones, since the metrics has been required only once at the end of the experiment, thus there is only one time-window of size 60 seconds. While for simplicity the 800 hundreds packets output is not reported, the Table 8.1 reports the most important data gathered for each detected connection, ignoring some derived value that is still computed and passed to the neural network (e.g., number of client packets over the server ones). Finally, the last row in the table reports the sum of all these values. To improve the readability of the data, the following legend has been used:

- C: client

- S: server
- PKTS: packets
- Gbits: total Gigabits transmitted from the entity

C IP	S IP	C Port	S Port	Proto	C PKTS	S PKTS	C Gbits	S Gbits
192.168.254.4	192.168.254.2	34754	5203	TCP	1583575	1299728	271.3	0.52
192.168.254.4	192.168.254.2	49248	5202	TCP	1622573	1303211	285.89	0.53
192.168.254.4	192.168.254.2	34756	5203	TCP	1421394	1205893	218.31	0.51
192.168.254.4	192.168.254.2	49244	5202	TCP	1423466	1205932	218.54	0.50
192.168.254.4	192.168.254.2	51616	5201	TCP	1540845	1166858	299.4	0.51
192.168.254.4	192.168.254.2	51620	5201	TCP	1422428	1207217	218.02	0.53
192.168.254.4	192.168.254.2	43740	5204	TCP	1664996	1293129	320.9	0.49
192.168.254.4	192.168.254.2	43742	5204	TCP	1728679	1296447	361.06	0.50
8 Connections					**22386371**		**2197.6**	

Table 8.1: Crypto-Mining gathered traffic statistics.

Interestingly, the Crypto-mining probe not only correctly detected all the connections distinguishing clients and servers, but it also computed all the traffic transmitted, as the total amount of client-server traffic is 2,197.6 Gbit, which corresponds to 274.7 GB previously indicated in the output of the iPerf2 servers. These results are impressive, and clearly indicate that the two eBPF programs developed for TOSHI do not affect at all the system performance, monitoring traffic and exporting data at high speed.

8.4 NetFlow comparison

The last scenario taken into account provides a pure comparison between *Dynmon* and *NetFlow*, a wide-used protocol developed by Cisco introduced in many networking tools like *ntopng* (which uses *nprobe*). *NetFlow* has always been one of the most rated paradigms to extract metrics and statistics from traffic, and collect in a centralized component all these values to be easy accessible to users. Although, to make a comparison as real and effective as possible, an eBPF program has been created, in order to replicate the same behaviour provided by *NetFlow*, but within the Polycube framework.

While *NetFlow*, and in particular *nprobe*, exploits the usage of the On-Demand Kernel Bypass with PF_RING technique, allowing user-defined Device Drivers to handle the raw peripheral, the *Dynmon* probe is injected in the system following the architecture depicted in Figure 8.8. There is a probe attached both to the

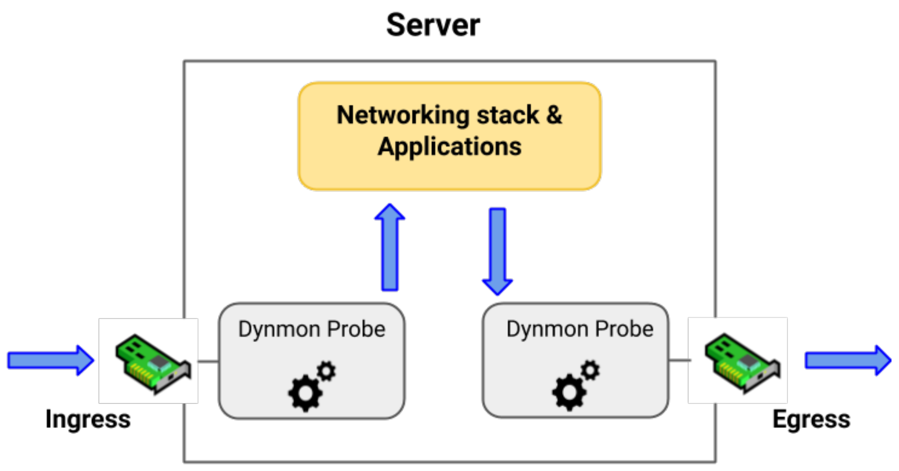


Figure 8.8: Dynmon probe architecture for *NetFlow* comparison.

Ingress and Egress hook, monitoring all incoming and outgoing packets. According to the information gathered and exported by the *NetFlow* protocol, the probe gathers the following features for each flow:

- 1^{st} packet timestamp
- duration
- protocol type
- source IP address
- source port (if any)
- destination IP address
- destination port (if any)
- IP protocol flags (if any)
- type of service
- total packets
- total bytes
- number of flows (if the same addresses, ports and protocol type are being reused)

More importantly, the *Dynmon* probe also supports IPv6 like *NetFlow*, but for simplicity this experiment carries only IPv4 data, in particular TCP connection between iPerf2 servers and clients, exactly like in the previous TOSHI scenario. These connections are being tracked for 60 seconds, and parameters like the average CPU available for other processing tasks and the bitrate reached by the system are computed and stored. Both the two instruments have been used to track only traffic passing through the *eth0* interface.

	Cpu Consumption (%)			**Bitrate (Gbits/sec)**	**Total GBytes**
	20s	**40s**	**60s**		
NetFlow	98.45	98.11	98.23	25.35	190.1
Dynmon	96.51	96.61	96.60	28.98	217.35
Dynmon+*	96.59	96.66	96.45	36.62	274.0

* The difference between Dynmon and Dynmon+ is the used eBPF map, which is a normal hash in the former case, and a PERCPU one in the latter.

Table 8.2: NetFlow and Dynmon resource and bitrate comparison.

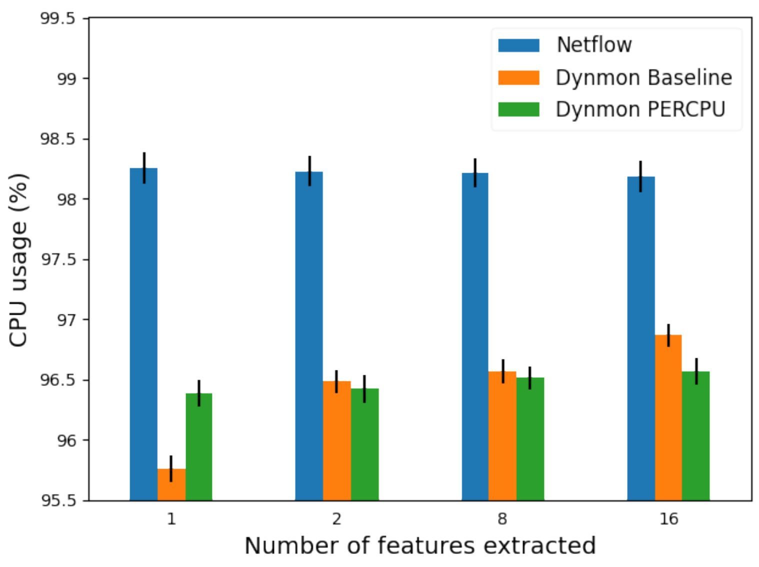


Figure 8.9: NetFlow and Dynmon probes CPU consumption comparison.

The Table 8.2 reports the surprising results of this test. Clearly, the average CPU available is quite constant for the entire test both for the three programs, but what really changes is the average bitrate measured. While running the *nprobe* program, the system was able to handle traffic at 25.35 Gbit/s, which is

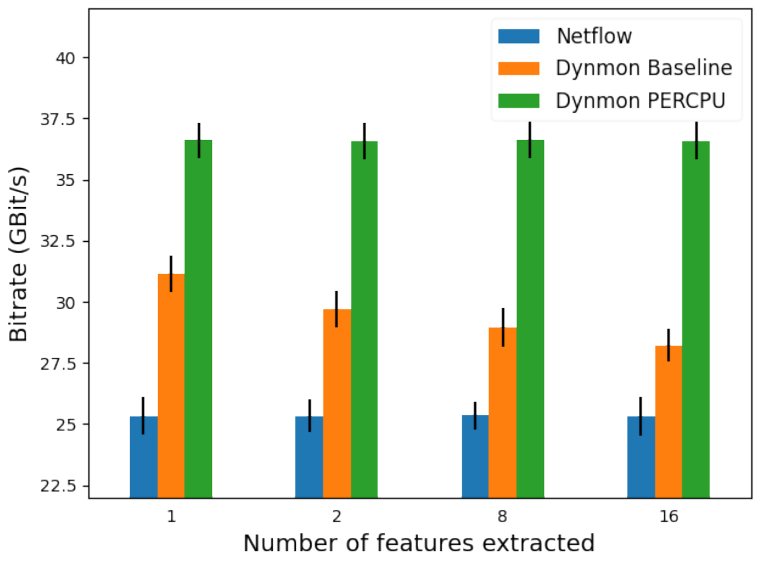


Figure 8.10: NetFlow and Dynmon probes bitrate comparison.

a good value considering that the full capacity (40 Gbit/s) of the connection is probably never reached, and that every packet has to be analysed. On the other hand, unexpectedly, the system was able to manage traffic at 36.62 Gbit/s while running the *Dynmon* PERCPU eBPF probe, which is significantly more that the previous one, almost exploiting the real wire speed as if no program would delay the processing of the packets.

The charts in Figure 8.10 and Figure 8.9 depict the measured differences between both *nprobe*, the baseline program developed in *Dynmon* using an eBPF hash map, and the respective enhanced version with the PERCPU maps. Surprisingly, even the baseline version is able to handle network traffic at very high speed, with a peak of 31.53 Gbit/s, which, considering that the map is shared between Ingress and Egress hooks, is a great result. However, the enhanced version allows to both handle more traffic (36.62 Gbit/s) and to save some CPU time, which can be used by other process other than the networking sub-system. The probe provided by *nprobe* definitively requires more resources than the eBPF programs, and it also decreases the throughput to 25.4 Gbit/s. Furthermore, as these programs have been tested by extracting a different set of features, while *nprobe* constantly requires the same resources, the eBPF programs can slightly save more CPU time by extracting less information from packets. On the other hand, when the number of features grows, the baseline version of the program needs more resources in order to parse the packets, while the PERCPU version clearly shows an almost constant behaviour, concerning both resources and bitrate.

In addition, to make sure that these results were coherent and real, the eBPF metrics have been extracted using an apposite Python extraction script, that, after retrieving the values from the Polycube daemon, perform some post-processing operations (e.g., changes IP from integer to string) in order to achieve the result reported in Table 8.3 (some parameter like duration, type of service and flows have been omitted to simplify the output, as they are known a-priori and not relevant at all in this test).

First seen	**Proto**	**Src IP:Port**	**Dst IP:Port**	**Flags**	**Packets**	**GBytes**
2020-09-23 16:58:46	TCP	192.168.254.4:43822	192.168.254.2:5204	...AP.SF	25590259	33.71
2020-09-23 16:58:46	TCP	192.168.254.4:49326	192.168.254.2:5202	...AP.SF	25200321	33.42
2020-09-23 16:58:46	TCP	192.168.254.4:34836	192.168.254.2:5203	...AP.SF	25278497	33.43
2020-09-23 16:58:46	TCP	192.168.254.4:51698	192.168.254.2:5201	...AP.SF	25417700	33.55
2020-09-23 16:58:46	TCP	192.168.254.4:34834	192.168.254.2:5203	...AP.SF	25160514	32.55
2020-09-23 16:58:46	TCP	192.168.254.4:43820	192.168.254.2:5204	...AP.SF	25444234	33.58
2020-09-23 16:58:46	TCP	192.168.254.4:51696	192.168.254.2:5201	...AP.SF	25663311	34.32
2020-09-23 16:58:46	TCP	192.168.254.4:49328	192.168.254.2:5202	...AP.SF	26549718	35.43
8 Connections					**204304554**	**274.0**

Table 8.3: Dynmon probe simplified output.

The output suggests that the *Dynmon* probe has successfully recorded all the traffic previously specified by the iPerf servers (274.0 GB), monitoring all the 8 TCP connections with no problems. Interestingly, the huge difference between the two programs designates new possibilities in the network monitoring field, which not only turned out to be efficient, but they also proved to be even better than older techniques, like in this last comparison.

Chapter 9

Conclusions

The proposed solution has shown promising results which go beyond the TOSHI project use case, and it has opened new horizons in the network monitoring. Many companies and projects are already trying to implement eBPF-based solution, like Cloudfare, both for fast packet analysis and statistics, proving that these technologies have a lot of potential that, from a research point of view, can be exploited and tried out. Moreover, this new method allows implementing a runtime adaptive packet inspection, which is extremely useful to tune the monitoring infrastructure according to both the system workload and the severity of the incoming traffic.

Interestingly, apart all the completely innovative way of adaptively monitoring traffic, this new technique has proven to be performant at least, but in some cases even better, than all the other traditional solutions, like *NetFlow*. This great achievement suggests that, in the worst case scenario, *Dynmon* performs exactly like all the other analysed solution, but in addition it allows modifying at runtime both the monitoring code and the metrics to be exported, while this is not possible to do with other tools.

The introduction of an adaptive and efficient monitoring logic, which allows users to both run their programs on-demand and change the quantity and granularity of information exported, should be clearly taken into account, because as far as this thesis has analysed the newly available technologies, it seems that the few drawbacks are strongly worth all the advantages individualized. Not to mention that the proposed solution addresses only one of the few limitations of the current state-of-the-art techniques. In fact, there are still open problems that could be overcome, but this thesis has pointed out that using eBPF to perform both networking functions and an adaptive analysis is an encouraging starting point for all the future developments.

9.1 Possible Dynmon improvements

As a matter of fact, the proposed solution can always be improved, both within the Polycube framework and outside as a stand-alone application.

Despite having introduced a lot of advanced features like the atomicity of maps thanks to the swap technique, a great enhancement that could improve the performance of the service could be the use of BTF (BPF type format), a metadata format that encodes debug information related to the BPF program and relative maps. This method is already used by *bpftool*, a tool which allows analysing information concerning both the eBPF programs and maps injected in the kernel using a command line interface. As a result, the use of BTF would significantly simplify the extraction logic of all the kinds of maps, which right now, despite being efficiently read exploiting the batch operations, takes a lot of time inspecting the type of the entries.

An alternative to BTF, would be introducing a “cache" to store the parsing method used for each eBPF map. This way, only the 1st entry has to invoke all the recursive method, while all the following ones would be extracted directly calling the function of interest. The advantage of such solution would be significant especially with huge eBPF maps, as the tests conducted pointed out some limitations of the extraction method when dealing with a lot of complex values to be extracted.

Furthermore, another great improvement to the service can be to enlarge the support of eBPF maps, allowing the extraction of all the types available so far. Unfortunately, many types are not so used, or they have never been used during these months, so the service offers support for all those that have actually been used, like arrays, queues, hash and PER-CPU. However, many of the most recent types of maps are still being developed, thus things like batch operations or PER-CPU map of those types are not available. It would be interesting to introduce directly in the Linux kernel these enhancements, in order to improve both the performance of the eBPF programs injected in *Dynmon*, and the supported maps in the service.

9.2 Possible eBPF programs improvements

Despite eBPF has proved to offer an interesting traffic monitoring support, even when using the *Dynmon* service realized, users have to manually write their own programs to be injected in the probe. The idea of providing templates of network monitoring programs, which offers the possibility to activate/deactivate certain network functions, like inspecting certain layers of the packets rather than others, would definitively ease the entire development process. As a result, users would have to only provide the portion of code to collect the data, worrying only to correctly invoke the specific map function. This way both the code reusability and scalability would improve, since the template must be coded only once and introduced in the framework, while the user can specify at runtime, using the Data Plane configuration file, which network functions he/she desires to use.

Furthermore, another advantage introduced by the introduction of templates, would also be the flexible granularity of the monitoring program, which can adapt itself, according to the user-defined parameters, in order to dig deeper or less some aspects of packets, lighten the system overhead in certain conditions where controls are not required at all.

Lastly, the potential of network monitoring programs introduced by eBPF could lead to a more sophisticated threats' detection engines, which could stop malicious traffic belonging to cyberattacks from the NIC itself, before it even enters the system. A notable example is provided in this [24] paper, where researches were able to detect and tag traffic belonging to a Skype conversation by only looking to packets headers correlations. Being able to categorize traffic at low level just by looking at some features would be revolutionary for the entire networking and cybersecurity world, which currently mostly relies on AI/ML algorithms running in user-space to retrospectively analyse packets, like *Cloudfare* [25].

Appendix A

Additional Open-Source Contributions

This section of the Appendix presents all the other important contributions to open-source projects. They were essential both for improving the Polycube framework and Dynmon itself, and to extend known and widely used frameworks.

A.1 BCC IOVisor improvement

The major contribution was to the BCC IOVisor project. BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. It makes use of extended BPF (Berkeley Packet Filters), generally known as eBPF, a new feature that was first added to Linux 3.15.

My contribution consists in adding support for new eBPF map types added in the latest Linux kernel versions, in order to allow users to use them within BCC, both for C++ and Python languages. Moreover, I revised other PR inherent to that extension, including the addition of the entire documentation for those new maps. These changes enable and ease the use of eBPF Queue/Stack maps within the framework, ensuring that all the interaction with the underlying data structure recall the specific system call.

A.2 Polycube Docker enhancement

Polycube provides a Docker image in order to deploy the entire framework without building the source code locally. Although, the Docker image was not optimized, since it contained all the needed dependencies both to build and execute the software.

My contribution consists in building a multi-layer Docker in order to install the needed dependencies and build the framework on a temporary image. From a fresh Ubuntu image, the final Docker extracts all and only the executable files (e.g., Polycube executable, shared libraries and system libraries) to make Polycube run correctly. A few tests have been run before publishing this change.

This choice led to a slimmer Docker image, from 2.7 GB of the previous image installed in the system (754 MB compressed) to 265 MB (81 MB compresses). This is really useful both to save space in the system, and to speed up the image download, especially in case Polycube has to be deployed in a cluster (e.g., the Kubernetes dedicated image).

A.3 Polycube Firewall upgrade

The Firewall service inside Polycube allows inserting Ingress and Egress rules in order to filter traffic as desired, like a real Firewall. Although, in order to insert rules the user has to perform one HTTP request at a time, even though the firewall presents a “transactional" mode where, before compiling the rules, it waits for the specific user request to mark the transaction as finished.

My contribution consists in modifying this mode, changing the “transactional" mode into a “batch" mode. Using newly created endpoints for the batch operations, users can specify a list of rules and the action that he/she wants to be performed on that rule (e.g., insert, modify or remove) that will be analysed and inserted in the eBPF programs at once. This mode allows to save a lot of time, which is essential especially in scenarios where interaction with Polycube are performed outside the device hosting the framework, meaning that these request have to pass through the Internet or a possibly congested interface (e.g., in TOSHI during a DDoS attack it is quite unlikely that all the single HTTP requests will be handled by the firewall).

A.4 Polycube Linux support

Polycube has always been tested with the version 4.15 of the Linux Kernel on Ubuntu. Despite it cannot be deployed on every Linux distribution for the needs of specific XDP kernel modules available for Ubuntu, I extended the kernel version support until the 5.7 version, enabling as a result all the newest Linux kernel features integrated (BPF maps, helpers, etc.).

The installation process has been enhanced, to detect the correct OS distribution and kernel version, in order to correctly download the needed dependencies for the compilation. In fact, since Ubuntu 20.04 a few packages like *GO* do not need to be manually installed from a dedicated repository any more, since they have been integrated on the Ubuntu official ones.

A.5 Polycube eBPF batch operations

Polycube defines an additional abstraction layer to interact with eBPF maps, in order to easily retrieve or modify data. This layer is encapsulated inside the *RawTable* class, deeply used by *Dynmon*, in order to extract data from a generic map.

Although, while general basic methods to retrieve entries were already present and used, I decided to implement also wrapper methods to use batch operations, which has proved to be 10x faster than normal ones. These batch operations are currently used only in *Dynmon*, but can be potentially called by any other service which interacts with its maps.

Bibliography

[1] *Wireshark*. https://www.wireshark.org/ (cit. on p. 5).

[2] *Netflow*. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html (cit. on p. 9).

[3] *ntop*. https://www.ntop.org/ (cit. on p. 11).

[4] *Snort*. https://www.snort.org/ (cit. on p. 12).

[5] Open Information Security Foundation. *Suricata*. https://suricata-ids.org/ (cit. on p. 12).

[6] *Netify*. https://www.netlify.com/ (cit. on p. 12).

[7] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese. «Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case». In: *IEEE Access* 7 (June 2019), pp. 107161–107170. DOI: 10.1109/ACCESS.2019.2933491 (cit. on p. 13).

[8] J. Levin. «ViperProbe: Using eBPF Metrics to Improve Microservice Observability». https://cs.brown.edu/research/pubs/theses/ugrad/2020/levin.joshua.pdf. MA thesis. Brown University, May 2020 (cit. on p. 13).

[9] *Cilium*. https://cilium.io/ (cit. on p. 13).

[10] Matt Fleming. *A thorough introduction to eBPF*. https://lwn.net/Articles/740157/ (cit. on p. 14).

[11] *LLVM*. https://llvm.org/ (cit. on p. 14).

[12] IOVisor. *XDP eXpress Data Path*. https://www.iovisor.org/technology/xdp (cit. on p. 16).

[13] *Polycube*. https://github.com/polycube-network/polycube (cit. on p. 16).

[14] Simone Magnani. *The Dynmon service.* `https://github.com/polycube-network/polycube/tree/master/src/services/pcn-dynmon` (cit. on pp. 19, 32).

[15] *YANG data modelling.* `http://www.yang-central.org/twiki/bin/view/Main/WebHome` (cit. on p. 24).

[16] M. Bjorklund, J. Schoenwaelder, P. Shafer, K. Watsen, and E. Wilton. *RESTCONF Extensions for the NMDA (RFC 8527).* `https://tools.ietf.org/html/rfc8527`. Mar. 2019 (cit. on p. 24).

[17] *OpenMetrics.* `https://openmetrics.io/` (cit. on p. 25).

[18] SoundCloud. *Prometheus.* `https://prometheus.io/` (cit. on p. 27).

[19] IOVisor. *BPF Compiler Collection (BCC).* `https://github.com/iovisor/bcc` (cit. on p. 40).

[20] *libbpf.* `https://github.com/libBPF/libBPF` (cit. on p. 40).

[21] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martínez-del-Rincón, and D. Siracusa. «Lucid: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection». In: *IEEE Transactions on Network and Service Management* 17.2 (June 2020), pp. 876–889. DOI: `10.1109/TNSM.2020.2971776` (cit. on p. 54).

[22] *iPerf2.* `https://sourceforge.net/projects/iperf2/` (cit. on p. 63).

[23] *Sar.* `https://linux.die.net/man/1/sar` (cit. on p. 63).

[24] D. Bonfiglio, M. Meo, M. Mellia, and D. Rossi. «Detailed Analysis of Skype Traffic». In: *IEEE Transactions on Multimedia* 11.1 (Feb. 2009), pp. 117–127. DOI: `10.1109/TMM.2008.2008927` (cit. on p. 78).

[25] *Cloudfare.* `https://www.cloudflare.com/` (cit. on p. 78).

Acknowledgements

With immense joy and a pinch of melancholy, my acknowledgements go to:

Prof. Fulvio Risso, for taking me under his wing, teaching me how to deal with problems, broaden my thought and for treating me like a son;

PostDocs. Sebastiano Miano, Alex Palesandro, and PhD. Marco Iorio, for their kindness, their professionalism and for having pushed me over the limits to overtake all difficulties;

Riccardo, Francesco, Enrico, Ilaria, Giulia and Daniele, my essential Turin friends, for walking side by side with me this two years, filling them with joy, love and immeasurable laughs which eased off stressful periods and the lack of home;

Edoardo, Fabrizio, Pietro, Carlo and Andrea, my cybersecurity teammates in Cesena, for their loyalty, their passion and knowledge shared also during these difficult years apart;

Rosanna, Fabrizio and Francesco, my beautiful family, and my grandma Paola, for their infinite love and for always believing in me and supporting my decisions, even though it means being far miles away from them;

all those who have shown me back love and sympathy during these years, especially my numerous UniTo friends, because it made me feel grateful and honoured, increasing my positivity and energy;

all those who turned out to be fake, resentful and opportunistic, because thanks to them I grew, I learned how to take care of myself and I developed extraordinary judgemental skills.

Infinite gratitude to you all.

"The road is now calling, and I must away"
Billy Boyd, The Last Goodbye