

POLITECNICO DI TORINO

Master of Science in Computer Engineering



Master's Degree Thesis

Dynamic Network Traffic Monitoring

Supervisor

Prof. Fulvio RISSO

Candidate

Michel SCIORTINO

April 2020

Table of Contents

1	Introduction	5
2	Background	7
2.1	Network Monitoring	7
2.2	The ASTRID Project	8
2.3	Thesis objective	9
3	Exploited Technologies	10
3.1	BPF	10
3.2	eBPF	13
3.3	Polycube	17
3.3.1	Polycube VNFs characteristics	19
3.4	Prometheus	20
3.4.1	Features	21
3.4.2	Architecture	21
3.4.3	Datamodel	21
3.5	OpenMetrics	23
4	Related works	24
4.1	Traffic monitoring tools	24
4.1.1	Tcpflow	24
4.1.2	NetFlow	26
4.2	Tracing with eBPF	27
4.2.1	Bpftrace	28
4.2.2	Falco	28

5	Architecture	30
5.1	Intelligent Remote Control Plane	30
5.2	Generic Self-adapting Local Control plane	32
5.3	Detailed VNF architecture	33
5.4	VNF Workflow	36
5.5	YANG model description	38
5.6	Input data plane configuration	40
5.7	Metrics	42
6	Implementation	44
6.1	Used languages	44
6.2	Implementation workflow	45
6.3	Main classes	46
6.4	Data plane configuration injection	48
6.5	Data extraction	49
6.6	Metrics export	53
6.7	Dynmon injector tool	54
7	Results	57
7.1	Injection of monitoring code	57
7.2	Extraction of collected metrics	59
8	Conclusions	63
8.1	Possible improvements	63

Chapter 1

Introduction

The growing adoption of virtualization technologies and micro-services architectures is re-shaping the traditional paradigms for running software appliances. Services are now designed as graphs of simple applications deployed over a virtualized set of computing resources linked by virtual links. While it's the best possible solution to avoiding single points of failure in a world full of security and performance threats (both malicious and accidental), this also complicates the way that these systems must be monitored and maintained making it much more expensive to implement and collect data; the more the services are disaggregated, the more the number of data to be collected increases. Moreover, although de-coupling software from the underlying infrastructure brings immediate benefits in terms of elasticity, portability, automation and resiliency, the intermediate hypervisor tier also raises new security concerns about the mutual trustworthiness between those two layers and the potential threats in the virtualization substrate; therefore, the need arises to activate precise monitoring able to inspect the traffic that reaches the hosted services. As the bandwidth capacity of modern networks increases, however, traditional traffic analysis mechanisms become increasingly inefficient and can lead to significant degradation of systems performance. Given the need for precise traffic monitoring and the unwillingness to bear its costs, this thesis proposes a new method of traffic analysis that exploits the capabilities of the eBPF technology to bring the traffic analysis as close as possible to its entry point into

the systems to reduce as much as possible the time necessary for the processing of individuals packets.

Chapter 2

Background

This chapter aims to discuss and explain the bases on which the thesis took place. Network monitoring is an essential component of IT security but it has his downsides; in modern distributed service architectures, its negative aspects are accentuated and this creates the need for new solutions for security management. This chapter will discuss about the problems related to network monitoring and explain comprehensively the objective of the thesis.

2.1 Network Monitoring

Network monitoring is a tool that allows you to infer the conditions of a network and of the devices that are part of it by tracking problems caused by non-functioning devices or overloaded resources (servers, network connections or other devices). Network monitoring is carried out by means of diagnostic software tools or specific hardware appliances which are connected to the network and are capable of analyzing the network traffic and the operation of the network devices.

Through network monitoring it is possible to generate alarms for the automatic warning of support staff in order to carry out the appropriate checks when problems are detected by the network monitoring system. Another feature of the monitoring systems is to generate reports on the state of the network where all the problems

detected are highlighted.

For security reasons, the ability to analyze traffic is extremely important because it allows you to find out when there are cyber attacks that exploit network vulnerabilities and how they act on the affected infrastructure.

2.2 The ASTRID Project

The ASTRID Project [1] proposes a new approach to assess situational awareness of virtualized services and effectively support quick remediation actions, beyond mere integration of security appliances in service graphs. The main concept is the disaggregation of cyber-security appliances into business logic and data plane, mediated by orchestration logic and proper security models.

The proposed framework expect the use of multiple programmable hooks attached to the virtualized containers (in the OS kernel, in system libraries, and in the micro-service code) in order to inspect and report what happens inside.

These programmable hooks include logging and event reporting capabilities developed by programmers into their software, as well as monitoring frameworks built in the kernel and system libraries that inspect network traffic and system calls (Figure 2.1).

Since monitoring operations and events reporting may introduce a significant overhead on the performances of the monitored hosts, the ASTRID framework foresee to selectively and locally adjust the monitoring deepness, the reporting type and rate in order to retrieve the exact amount of knowledge needed, without overwhelming the whole system with unnecessary information.

The purpose is to get more details for critical or vulnerable components when anomalies are detected that may indicate an attack, or when a warning is issued by cyber-security teams about new threats and vulnerabilities just discovered.

In the framework architecture, a Security Orchestrator enriches the deployed services graph with the programmable hooks required to monitor and inspect traffic and system calls. An analytics engine continuously monitors the statistics exported by the deployed hooks and requires them to dynamically change their

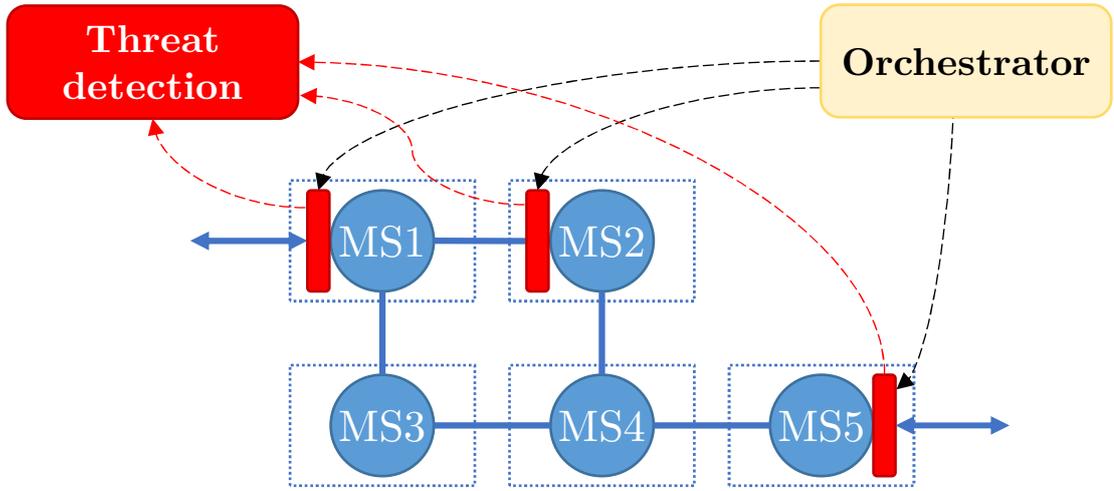


Figure 2.1: Embedded service-centric security framework

behavior, in order export different statistics over time to feed the actual analysis needs.

2.3 Thesis objective

This thesis focuses on the creation of a new programmable hook able to inspect the network traffic and export dynamically defined network statistics meeting the requests of the ASTRID project.

The objective is to create a lightweight and adaptive traffic monitoring service able to change its inspection deepness during its operation. This has been achieved by using Polycube [2], an open source framework developed by the Computer Networks Group of Politecnico di Torino, which allows the creation of Virtual Network Functions capable of efficiently inspecting and manipulating the network traffic by exploiting the eBPF technology.

Chapter 3

Exploited Technologies

This chapter introduces the main technologies that have been used to create the architectures and implementations proposed in this thesis, explained in detail in chapters 5 and 6. The informations about tools and technologies presented in this chapter are inspired on the documentations present in the corresponding websites.

3.1 BPF

The Berkeley Packet Filter (BPF) provides on some Unix-like OSes a raw interface to data link layers in a protocol-independent fashion, and the potential to operate with custom code on the intercepted packets. All packets on the network, even those intended for other hosts, are accessible through this mechanism, provided that the network driver support promiscuous mode. BPF roughly offers a service similar to raw sockets, but it provides packet access through a file interface rather than a network interface (the packet filter appears as a character special device. The BPF was designed as a common agent to allow network monitoring by multiple applications running in user space, with the specific purpose of minimizing packets getting copied between user and kernel space, which is known to lead to large performance degradation. Thus, BPF is specifically designed with efficient and effective packet filtering in mind, in order to discards unwanted packets as early as possible within the OS's stack. Associated with each open instance of a BPF

file is a user-settable packet filter. Whenever a packet is received by an interface, all file descriptors listening on that interface apply their filter. Each descriptor that accepts the packet receives its own copy. Reads from these files return the next group of packets that have matched the filter. Consequently, any modification performed on captured packets do not influence the actual data, as in BPF packets are always a copy of the original traffic.

BPF has two main components: the network tap and the packet filter. The network tap collects copies of packets from the network device drivers and delivers them to listening applications. The filter decides if a packet should be accepted and, if so, how much of it to copy to the listening application. Figure 3.1 illustrates BPF's interface with the rest of the system.

Because network monitors often only want a small subset of network traffic, a dramatic performance gain is realized by filtering out unwanted packets in interrupt context. To minimize memory traffic, the major bottleneck in most modern workstations, the packet should be filtered 'in place' rather than copied to some other kernel buffer before filtering. Thus, if the packet is not accepted, only those bytes that were needed by the filtering process are referenced by the host. BPF uses a re-designed, register-based 'filter machine' that can be implemented efficiently on today's register-based CPUs. Further, BPF uses a simple, non-shared buffer model made possible by today's larger address spaces. The model is very efficient for the 'usual cases' of packet capture.

The design of the BPF was guided by the following constraints:

- it must be protocol independent: the kernel should not have to be modified to add new protocol support;
- it must be general: the instruction set should be rich enough to handle unforeseen uses;
- packet data references should be minimized;

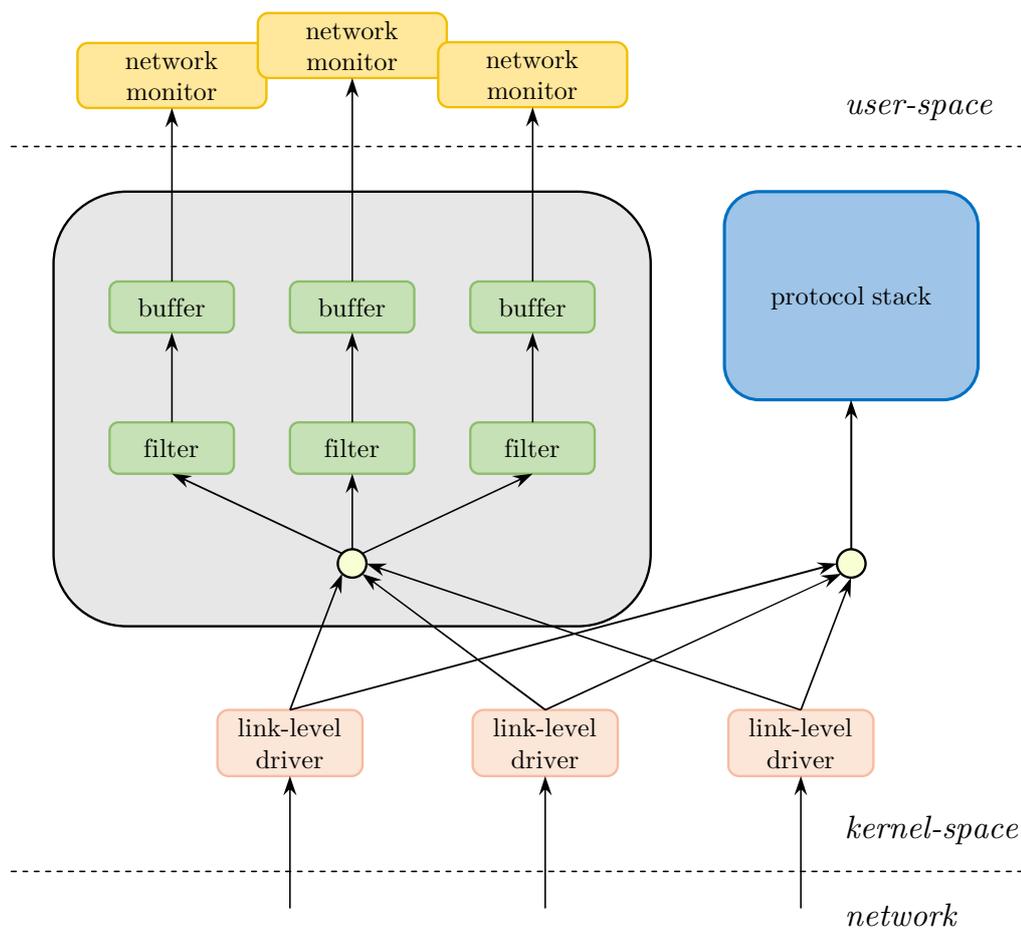


Figure 3.1: Overview of BPF architecture

- decoding an instruction should consist of a single C switch statement;
- the abstract machine registers should reside in physical registers.

The BPF abstract machine consists of load, store, ALU, branch, return and miscellaneous instructions that are used to define low-level comparison operations on packet headers; on some platforms, these instructions are converted with just-in-time compilation into native code to further avoid overhead.

BPF often refers to the filtering mechanism, rather than the entire interface. With this meaning, it is sometimes implemented in OS's kernels for raw data link layer socket filters (e.g., in Linux).

3.2 eBPF

Initially proposed by Alexei Staravotov in 2013, eBPF is the next version of BPF, which includes both modifications to the underlying virtual CPU (64-bit registers, additional instructions) and to the possible usages of BPF in software products. Packets are no longer a copy of the original data: eBPF program can operate and modify the packet content, hence enabling a new breed of applications such as bridging, routing, NATting, and more. The “Classic” BPF is not used anymore, and legacy applications are adapted from the BPF bytecode to the eBPF.

An overview of the runtime architecture of eBPF is shown in Figure 3.2. The following subsections will explain some of the relevant parts of the architecture and point out some of the main improvements in eBPF.

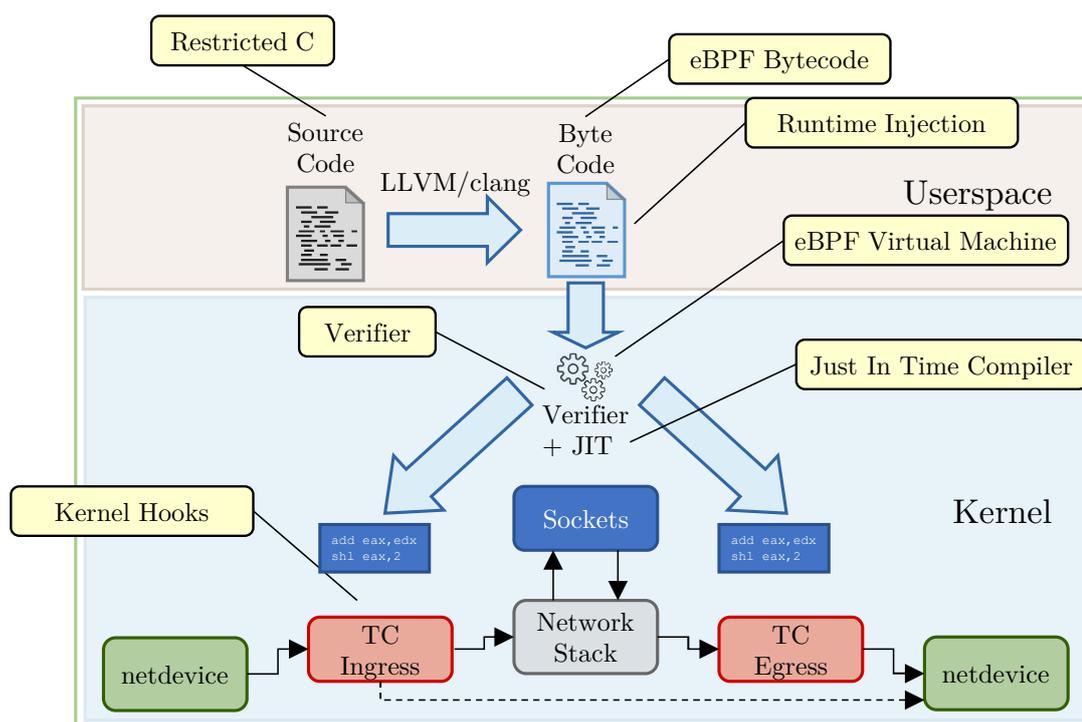


Figure 3.2: eBPF architecture

C-based programming

eBPF code can be written in (a restricted version of) C, which allows for easier program development and more powerful functionalities with respect to bare assembly.

Maps

An eBPF program is triggered by a packet received by the virtual CPU. To store the packet in order to process it, eBPF defines a volatile “packet memory”, which is valid only for the current packet: this means there is no way to store information needed across subsequent packets.

eBPF defines the concept of state with a set of memory areas, which are 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 that will be used to reference it.

An important feature of maps is that they can be shared between eBPF programs, and between eBPF and user-space programs. This is especially important for all those applications that need to perform operations that exceed the complexity allowed by the eBPF bytecode. In fact, maps allow to split complex processing in two layers (fast eBPF datapaths and slow user space control paths), keeping the state information shared and synced. Another important advantage of using maps is that their content is preserved across program executions.

Maps are never accessed directly: maps are read and written with predefined system calls. An important side effect of using maps is that the state of the program is decoupled from the code. Instructions are in the program, the data used by such instructions are in the maps.

Hooks

eBPF programs can react to generic kernel events, not only packet reception: they can react to any system call that exposes a hook.

Considering a network packet and recalling how the netfilter hooks work, with eBPF we can listen to any of the predefined hooks to trigger programs only at

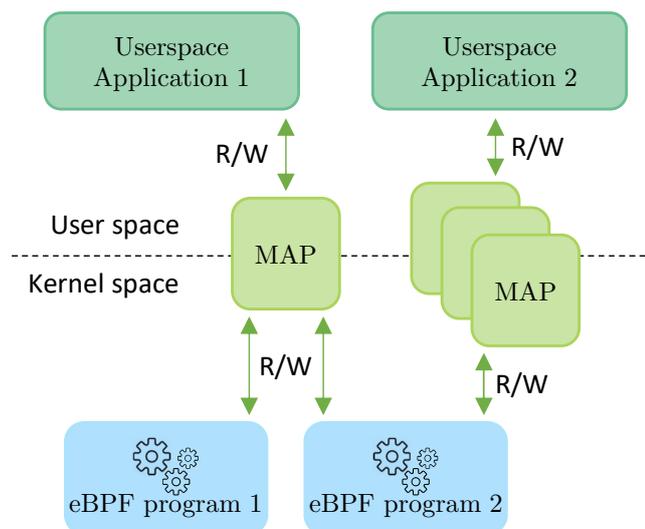


Figure 3.3: eBPF maps

certain steps during packet processing. Netfilter is a set of linked modules but it has no filtering concept: attaching to a hook means receiving all the packets. eBPF can attach to hooks and filter packets.

The TC Ingress hook intercepts all the packets that reach the network adapter from the outside. The TC Egress hook deals with the outgoing packets immediately before sending them to the network adapter.

XDP

XDP (eXpress Data Path) is a programmable, high performance packet processor in the Linux networking data path; it provides an additional hook to be used with eBPF programs to intercept packets in the driver space of the network adapter, before they are manipulated by the Linux kernel. The main advantage of this early processing is that it avoids the overhead and the memory consumption added by the kernel to create the socket buffer (*skb* data structure) which wraps the packet for standard Linux processing in TC mode. XDP runs in the lowest layer of the packet processing stack, as soon as the NIC driver realizes a packet has arrived. However, packets here are not delivered to userspace, but to the injected eBPF program executed in kernel space. One of the main use cases is pre-stack processing

for filtering or DDOS mitigation.

Service chains

BPF did not quite have the concept of multiple cooperating programs: each parallel program receives a copy of the packet and processes it; instead, eBPF can link multiple programs to build service chains, such as in Figure 3.4.

Service chains can be created exploiting direct virtual links between two eBPF programs or tail calls. Tail calls can be thought as function calls: the eBPF programs are separated, but the first one triggers the execution of the second by calling it. This allows developers to overcome the program size limitation in the JIT compiler: starting from one big program, this can be split in multiple modules, perhaps functionally distinct such as header parsing, ingress filtering, forwarding, and analytics.

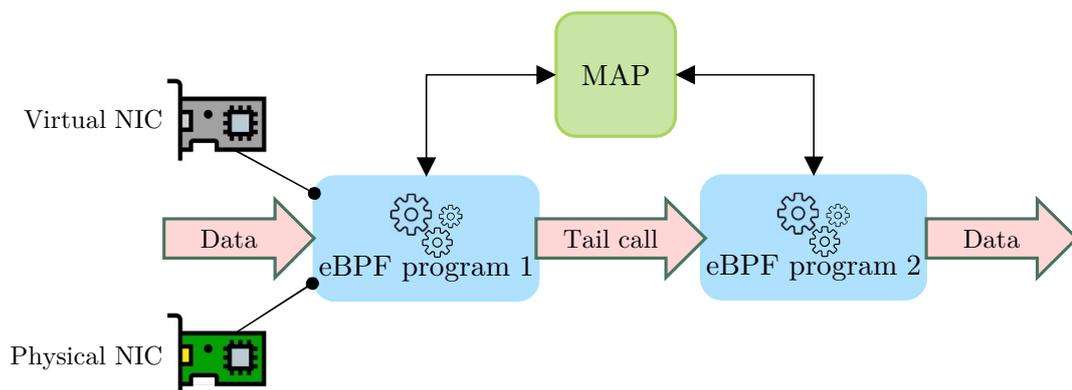


Figure 3.4: eBPF service chain example

Helpers

Helpers are sets of functions pre-compiled and ready to be used inside the Linux kernel. eBPF programs can call such functions, which are outside the virtual CPU (e.g. function to get the current timestamp). Helpers delegate complex tasks to the

operating system, overcoming the complexity restrictions in the eBPF validator and allowing developers to exploit advanced OS functionalities.

System-wide monitoring capabilities

In addition to filtering packets, eBPF programs can be attached and run anywhere in the Linux Kernel through *kprobes*. This allows inspecting the execution of the kernel and userspace programs and opens the door to high-performance Linux monitoring without running extra kernel modules. Potential kernel crashes caused by eBPF are prevented through static analysis. The kernel runs a verifier on every eBPF program before allowing its execution: it checks for any loops within the code (which could lead to possible infinite loops, thus hanging the kernel) and any unsafe memory accesses.

Using eBPF for dynamic tracing is very powerful and easier, as eBPF programs can now be coded in C thanks to an LLVM [3] backend. The BPF Compiler Collection [4] simplifies development even further, although it comes with additional runtime dependencies (Python, kernel headers and LLVM). Thanks to this technology, any system call in the kernel can be monitored, and its calling/return parameters can be inspected, paving the way for a holistic view of any kernel activity in real-time.

3.3 Polycube

Polycube is an open source framework developed by the Computer Networks Group of Politecnico di Torino, which allows the creation of Virtual Network Functions capable of efficiently inspecting and manipulating the network traffic by exploiting the eBPF technology.

All Polycube network functions feature a unified point of control, which enables the configuration of high-level directives such as the desired service topology. Polycube supports this model through a single, service agnostic, user space daemon, called `polycubed`, which is in charge of interacting with the different network function instances.

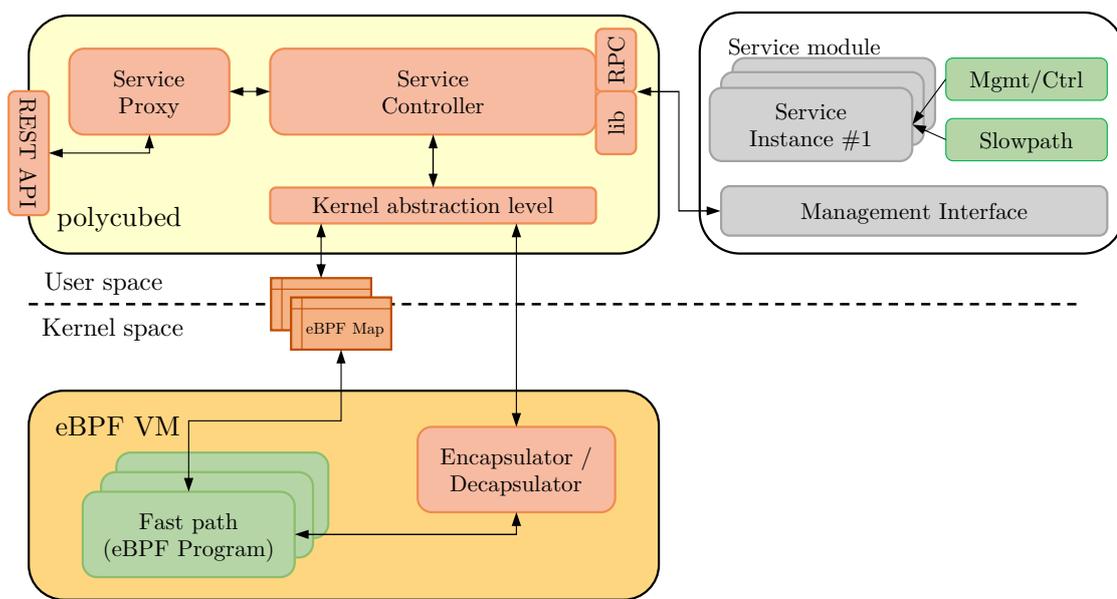


Figure 3.5: Polycube high-level architecture

Each different type of virtual function is called *Cube*, which are similar to plugins that can be installed and launched at runtime. A new Cube can be easily added to the framework by a specific registration phase, in which the service sends the information required for its identification within the framework. When the service is registered, different instances of it can be created by contacting `polycubed`, which acts mainly as a proxy; it receives a request from a northbound REST interface and forwards it to the proper service instance, returning back the answer to the user.

3.3.1 Polycube VNFs characteristics

Each Polycube service is made up of a *control plane* and a *data plane*. The data plane is responsible for per-packet processing and forwarding, while the control and management plane is in charge of service configuration and non-dataplane tasks (e.g., routing protocols). Although this separation between the control and data plane is common in many network functions architectures, Polycube provides a clear separation between these components; each service is composed of a set of standard parts that make it easier for the programmers to implement the desired behavior, while Polycube takes care of creating all the surrounding glue logic, handling all the interactions and communications between the different components.

Data plane

The data plane design of a Polycube service is characterized of a fast path, namely the eBPF code that is injected into the kernel, and a slow path, which handles packets that cannot be fully processed in the kernel or that would require additional operations, slowing down the processing of the other packets.

The data plane portion of a network service is executed per packet, with the consequent necessity to keep its cost as small as possible. When fired, the fast path retrieves the packet and its associated meta-data from the receive queues, then it executes the injected eBPF instructions. Typical operations are usually very fast, such as packet parsing, lookups in memory (e.g., to classify the packet), and map updates, such as storing data in memory (e.g., statistics), for further processing. When those operations are carried out, the fast path returns a forwarding decision for that particular packet or send it to the slow path for further processing.

Although eBPF offers the possibility to perform some complex and arbitrary actions on packets, it suffers from some well-known limitations due to his restricted virtual machine, which however are necessary to guarantee the integrity of the system. Those limitations may impair the flexibility of the network function, which may not be able to perform complex actions directly in the eBPF fast path or could slow down its execution, adding more instructions in the fast path to handle exceptional cases. To overcome those limitations, Polycube introduces an additional

data plane component that is no longer limited by the eBPF virtual machine and it can hence execute arbitrary code. The slow path module is executed in user space and interacts with the eBPF fast path using a set of components provided by the framework. The eBPF fast path program can redirect packets (with custom meta-data) to the slow path, similar to PacketIn messages in OpenFlow. Similarly, the slow path can send packets back to the fast path; in this case, Polycube provides the possibility to inject the packet into the ingress queue of the network function port, simulating the reception of a new packet from the network, or into the egress queue, hence pushing the packet out of the network function.

Control and management plane

The control plane of a virtual network function is the place where out-of-band tasks, needed to control the data plane and to react to possible complex events (e.g., Routing Protocols, Spanning Tree) are implemented. It is the point of entry for external players (e.g. service orchestrator, user CLI) that need to access service's resources, modify (e.g., for configuration) or read service parameters (e.g., reading statistics) and receive notifications from the service fast path or slow path.

Polycube defines a specific control and management module that performs the previously described functions. It exposes a set of REST APIs used to perform the typical CRUD (create-read-update-delete) operations on the service itself; these APIs are automatically generated by the framework starting from the service description, removing this additional implementation overhead to the programmer. To interact with the service, an external player has to contact `polycubed`, which checks the service to which the request is directed to and dispatches it to the corresponding service control path, which in turn serves the request modifying its internal state or reflecting the changes to the service data path instance.

3.4 Prometheus

Prometheus [5] is an open-source systems monitoring and alerting toolkit originally built at SoundCloud by ex-Googlers that wanted to monitor a highly dynamical

container environment. Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes [6].

3.4.1 Features

Prometheus's main features are:

1. a multi-dimensional data model with time series data identified by metric name and key/value pairs
2. PromQL, a flexible query language to leverage this dimensionality
3. no reliance on distributed storage; single server nodes are autonomous
4. time series collection happens via a pull model over HTTP
5. pushing time series is supported via an intermediary gateway
6. targets are discovered via service discovery or static configuration
7. multiple modes of graphing and dashboarding support

3.4.2 Architecture

The diagram in figure 3.6 depicts the architecture of Prometheus and some of its ecosystem components:

Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana [7] or other API consumers can be used to visualize the collected data.

3.4.3 Datamodel

Prometheus fundamentally stores all data as time series: streams of timestamped values belonging to the same metric and the same set of labeled dimensions. Every time series is uniquely identified by its metric name and optional key-value pairs called labels.

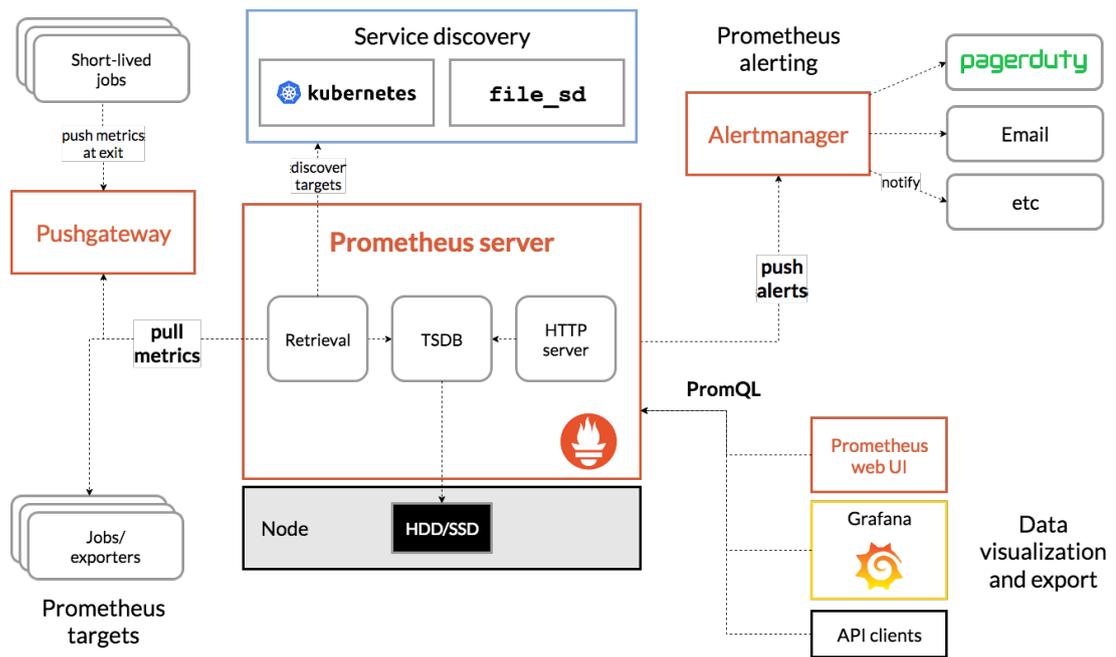


Figure 3.6: Prometheus architecture

The metric name specifies the general feature of a system that is measured. Labels enable Prometheus’s dimensional data model: any given combination of labels for the same metric name identifies a particular dimensional instantiation of that metric. The query language allows filtering and aggregation based on these dimensions.

Metrics in Prometheus have a type to indicate their meaning and usage. Prometheus currently supports four metric types:

1. **counter** is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart;
2. **gauge** is a metric that represents a single numerical value that can arbitrarily go up and down;
3. **histogram** samples observations (usually things like request durations or response sizes) and counts them in configurable buckets and provides a sum of all observed values;
4. **summary** is similar to a histogram but it calculates configurable quantiles over a sliding time window.

3.5 OpenMetrics

OpenMetrics [8] is an open source initiative, focused on creating a neutral metrics exposition format that is an evolution of the widely-adopted Prometheus exposition format. OpenMetrics brings together the maturity and adoption of Prometheus along with the experience and needs of a variety of projects, vendors, and end-users, aiming to move away from the hierarchical way of monitoring to enable users to transmit metrics at scale. The OpenMetrics format is a text-based metrics exposition format; this format has the advantages to be human-readable, easy to assemble and readable line by line as each metric is expressed in one text line (excepting the comments which may surround the metric).

The OpenMetrics format supports the four types of metrics supported by Prometheus (counter, gauge, histogram, summary) plus a generic type called untyped.

A metric is composed by several fields:

1. metric name
2. any number of labels represented as a key-value array
3. current metric value
4. optional metric timestamp

Metric output is typically preceded with `#HELP` and `#TYPE` metadata lines. The `HELP` string identifies the metric name and a brief description of it. The `TYPE` string identifies the type of metric. If there's no `TYPE` before a metric, the metric is set to `untyped`. Everything else that starts with a `#` is parsed as a comment.

Listing 3.1: Metric example

```
1 # HELP metric_name Counts the total number of POST http request failed with status 400
2 # TYPE metric_name counter
3 # Comment that's not parsed by Prometheus
4 http_requests_total { method="post", code="400" } 3 1395066363000
```

Since the OpenMetrics exposition format is going to be the de facto model for cloud native metric monitoring, it has been chosen to use it for the metrics exported by the proposed service within this thesis.

Chapter 4

Related works

This section will shortly provide a description of these projects to highlight their limits, as they were thoroughly considered during the development of the Dynmon service.

4.1 Traffic monitoring tools

This section presents two tools used today to monitor the network traffic and to collect information in order to detect anomalies.

4.1.1 Tcpflow

Tcpflow is an open source tool created in 1999 which is capable of capture data transmitted as part of TCP connections (flows). Tcpflow reconstructs the actual data streams and stores each flow in a separate file for later analysis. Tcpflow understands TCP sequence numbers so it is able to correctly reconstruct data streams regardless of retransmissions or out-of-order delivery. The main field of use of Tcpflow is the forensic analysis of network traffic. For this reason, it has been designed to collect as much information as possible with the awareness that it is often not possible to catch the same traffic again. A key feature of Tcpflow is the possibility to specify filtering expressions in order to filter the packets to be captured. Since Tcpflow uses the the libpcap library, tcpflow has the same powerful

filtering language available as programs such as tcpdump. If the HTTP or SMTP application protocol are encapsulated in the captured traffic, Tcpflow is also able to extract and save to disk the messages exchanged in each section.

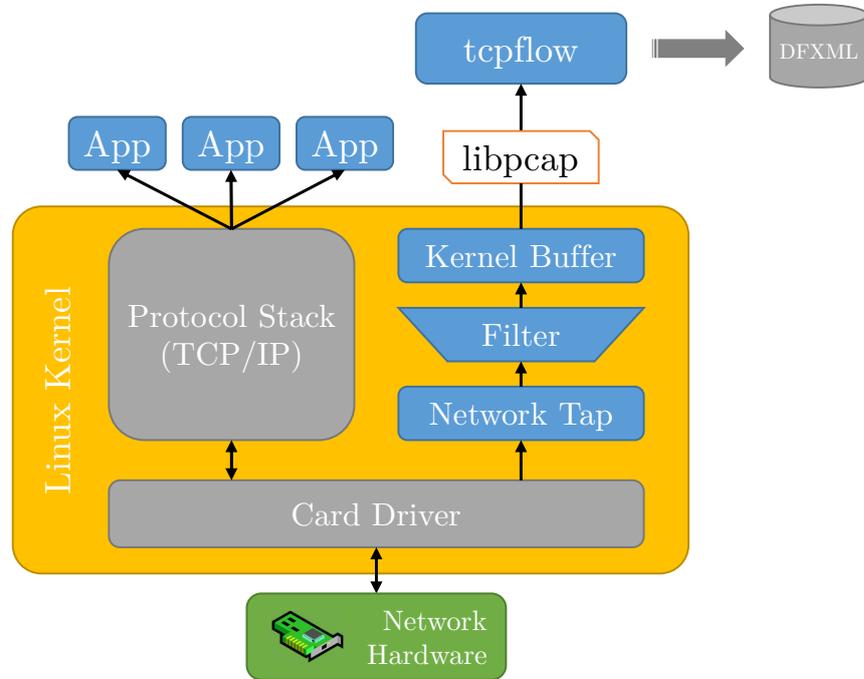


Figure 4.1: Tcpflow pipeline

The output produced by the Tcpflow tool is a DFXML (Digital Forensics XML) file in which are stored various information including the following statistics on the analyzed TCP flows:

1. timestamp of the beginning of the flow;
2. end of flow timestamp;
3. source IP address;
4. destination IP address;
5. source MAC address;
6. destination MAC address;
7. source port;

8. destination port;
9. number of transmitted packets;
10. number of transmitted bytes.

Additionally, Tcflow is also able to export a a Portable Document Format (PDF) file in which there are collected some overall statistics of the capture such as a series of bar graphs showing the data received over time, the IP addresses and the source ports that sent the most traffic and the IP addresses and destination ports they have received more traffic.

Although this tool is capable of collecting information on network traffic, it is not capable of monitoring any type of network traffic and extracting from it only the information deemed interesting.

4.1.2 NetFlow

NetFlow is a network protocol that was introduced on Cisco routers around 1996 that provides the ability to collect IP network traffic as it enters or exits an interface. The flow data is then analyzed to create a picture of network traffic flow and volume. The NetFlow protocol is used as a network traffic analyzer to determine the source and destination of the traffic, volume and paths on the network, class of service and the causes of network congestion. Netflow is able to provide detailed insight on the network bandwidth usage. Before NetFlow, network engineers and administrators used Simple Network Management Protocol (SNMP) for network traffic analysis and monitoring.

Cisco standard NetFlow defines a flow as a unidirectional sequence of packets that all share the following 7 values:

1. ingress interface (SNMP ifIndex);
2. source IP address;
3. destination IP address;
4. IP protocol;
5. UDP and TCP source port;

6. UDP or TCP destination port, type and code for ICMP;
7. IP type of service.

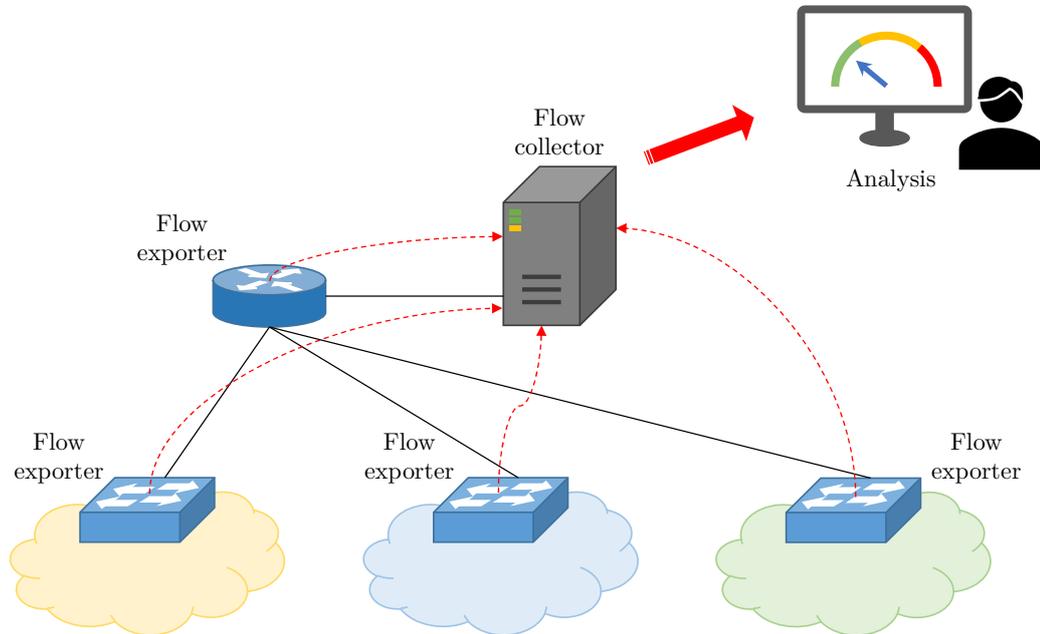


Figure 4.2: NetFlow environment

A typical NetFlow flow monitoring setup consists of three main components:

1. *flow exporters*: routers and switches which collect IP traffic statistics from their interfaces, aggregating packets into flows and exporting the statistics as flow records;
2. *flow connectors*: devices which collect, store and pre-process flow data received from the flow exporters;
3. *analysis application*: analyzes received flow data in the context of intrusion detection or traffic profiling.

Even if this tool is able to collect statistics on the network traffic, it is not flexible enough as each monitoring node has to support the NetFlow protocol.

4.2 Tracing with eBPF

The use of the eBPF technology has been around for many years, and in the Linux community it has been exploited many times in order to trace the performance of

the systems and for network monitoring. This section presents two tools which make use of this technology.

4.2.1 Bpftrace

Bpftrace [9] is a high-level open-source tracing tool developed by Alastair Robertson which allows to analyze systems in custom ways. Bpftrace involves the injection of custom tracing scripts inside the Linux kernel to allow the analysis and observation of the system from the inside by exploiting the eBPF capabilities. Bpftrace uses LLVM as a backend to compile scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system, as well as existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints.

Even if powerful, bpftrace is not enough; it lacks of an ecosystem that can allow it to cooperate with a dynamic monitoring system able to change its behavior at runtime.

4.2.2 Falco

Falco [10] is an open-source cloud-native runtime security project originally created by Sysdig and now part of the CNCF. Falco is a behavioral activity monitoring agent that comes with native support for containers. Falco lets to define highly granular rules to check for activities involving file and network activity, process execution, IPC, and much more, using a flexible syntax. Falco will notify you when these rules are violated.

Falco offers the following capabilities:

1. file integrity monitoring: Falco watches for any OS activity that is writing to a file of interest, and generates alertes in real-time;
2. network monitoring: Falco can see I/O from the inside of VMs and containers, and can correlate network traffic with applications activities;
3. detection capabilities: Falco is far simpler to understand and configure than

Linux security modules like SELinux and AppArmor, but it offers only the possibility to detect the problems, it cannot enforce measures to solve them.

Falco is a long-running server agent. In containerized environments, it can install as a container which monitors the host itself and all containers running on it. In regular environments it can also be installed as a regular host package. Once activated, Falco taps into the stream of system call events, checking each event against the list of rules in its configuration file. Specifically, it uses the sysdig kernel module for syscall interception and sysdig user libraries for state tracking and event decoding.

Behaviors and activities of interest are expressed as rules using a simple filter language; each rule has an associated output template specifying the message to be output if a matching event occurs.

Due to its nature, Falco cannot be used to do collection, alerting, reporting or remediation when anomalies are detected; moreover Falco's performance are an open issue because in a busy hosts or with large rule sets Falco has a high CPU usage.

Chapter 5

Architecture

This chapter illustrates the architecture defined for the proposed solution. In its initial part, this work focused on the design of an architecture that could combine the strengths of Polycube VNFs with the requests of the ASTRID project.

Two possible approaches were conceived to create a Polycube service that can handle generic monitoring data planes and expose the corresponding metrics:

1. Intelligent Remote Control Plane
2. Generic, Self-adapting Local Control plane

The two architectures are explained in the following sections. Considering the current capabilities of the framework, the different complexity between the two solutions and the time needed for prototyping, also given the need for the developers of the ASTRID project to use our results, the second architecture was chosen.

5.1 Intelligent Remote Control Plane

This first approach is meant to detach the implementation of the data plane and the control plane of the network function giving the possibility to instantiate the two components on different physical machines.

This requires to define a third component in charge of transforming monitoring policies into an instance of the virtual network function through the use of a code

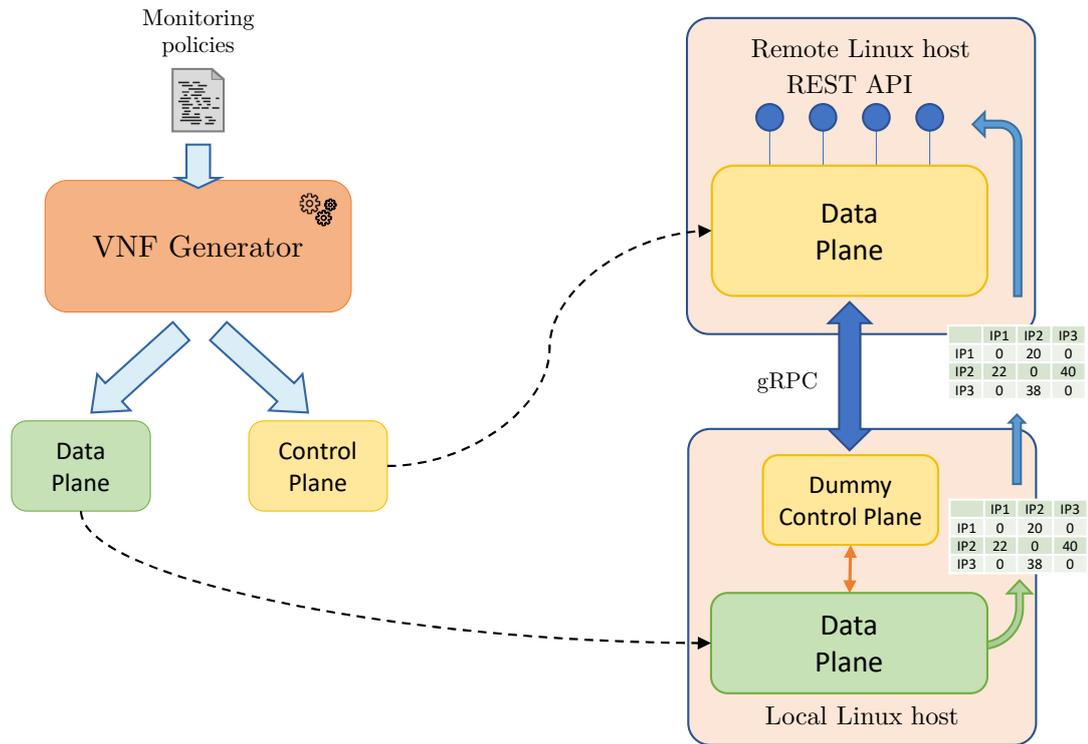


Figure 5.1: first approach architecture

generation pipeline and a general purpose compiler.

Once the two components have been generated, they can be instantiated on two different machines: the custom control plane is instantiated on a remote machine (for example, an SDN controller), while the dataplane is injected at runtime as a new service in Polycube on the machine to be monitored. The instanced components can then communicate with each other through a gRPC channel to allow the transport of data between the two. At this point, the two components are completely independent and it is possible to modify both independently. For example, it is possible to modify the logic of data extraction from the network traffic, simply by replacing the data plane component, keeping the communication logic with the remote control plane unchanged.

This first approach has the following advantages:

1. the ability to perform complex data processing in the control plane (e.g.,

aggregating data intelligently);

2. the possibility of having a flexible remote control plane that is completely disconnected and independent of Polycube and therefore can be written with languages other than C++ (the language used in Polycube).

The need to have a generation, compilation and building chain, however, makes this architecture too complex. In addition to this we need to consider the missing implementation of the gRPC communication in Polycube, a feature that has been abandoned. Moreover, this architecture raises security concerns: the control plane is executed locally and natively on the remote host (not in the sandbox such as the data plane), hence can pose non-negligible security problems as it has access to all the informations the dataplane exposes.

5.2 Generic Self-adapting Local Control plane

This second approach tries to simplify the architecture described above, eliminating the remotization of the control plane, enabling the local control plane to export monitoring information.

The basic idea is the creation of a Virtual Network Function which, once running, receives the configuration of its data plane through its REST interface, injects the monitoring code at runtime and exports the metrics it collects through specific endpoints.

The control plane has to be as generic as possible in order to be able to run any data plane code, allowing the extraction of the data it wants to expose. For this to work, a mechanism to extract the data collected by the dataplane is required. Since in Polycube the control plane and the data plane of a VNF can communicate with each other through the use of eBPF maps, it has been decided to exploit them to store the information extracted from the traffic (by the data plane), and expose their content as metrics through the control plane's REST interface.

The data structure received as input, which represents the new data plane, includes the inject-able eBPF code and a description of the metrics to be exported.

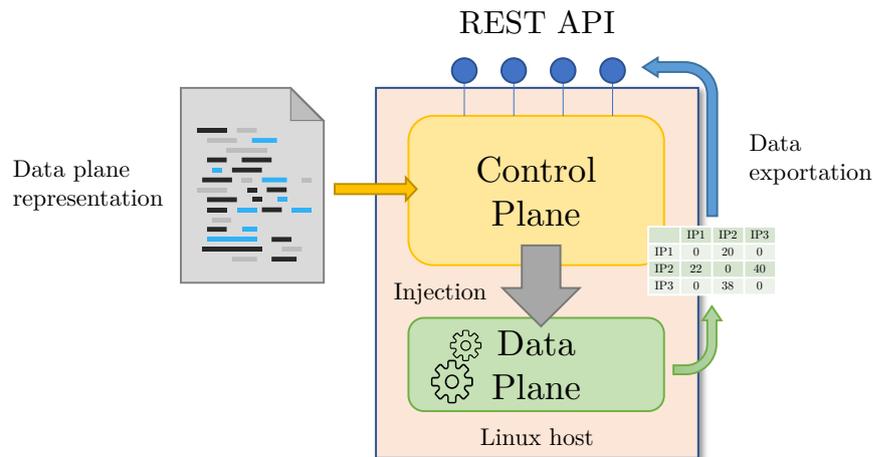


Figure 5.2: second approach architecture

This second approach has the following advantages:

1. dynamic and versatile: the ability to change the dataplane code at runtime allows the extraction of different informations over time, depending on the injected code;
2. continuous operation: thanks to the dynamic property of this architecture, there is no need to re-instantiate the VNF when the monitoring parameters change, so no packet is lost during the dataplane changes.
3. single point of control: as the VNF is instantiated in the monitored host, there is no need of connections to multiple machines, nor of a gRPC channel.

5.3 Detailed VNF architecture

As shown in Figure 5.4, the control plane and the data plane are the two main components of the conceived VNF structure.

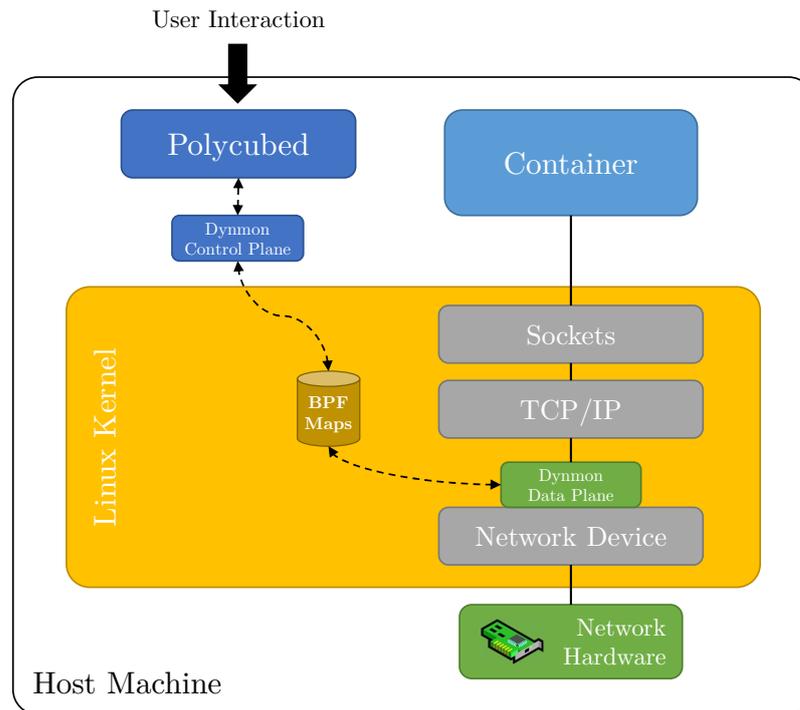


Figure 5.3: Overall architecture

Control plane

The control plane represents the control point of the virtual network function; it allows communication with the user and management of the data plane. The control plane contains the logic responsible of:

- receive the the HTTP requests coming through his REST interface;
- check the validity of the incoming data;
- inject the data plane code at runtime;
- store the metadata related to the injected data plane;
- interpret the structure of the eBPF maps;
- extract the content of the exportable maps;
- return the extracted data in form of metrics.

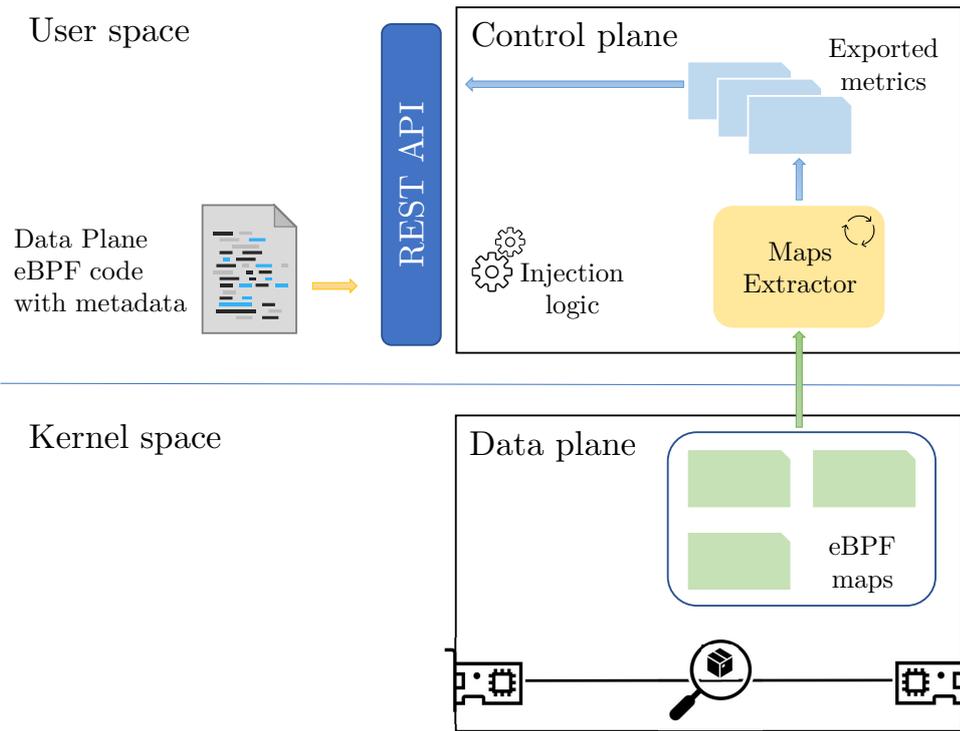


Figure 5.4: VNF high-level architecture

The Map Extractor is a component designed to interpret the structure of the allocated eBPF maps and extract their content by exploiting some internal metadata provided by the Polycube framework, which describes how the entries of a eBPF map are structured.

Data plane

The data plane is a low-level component responsible for processing network packets. The data plane contains the logic responsible of:

- create eBPF maps to collect data;
- analyze every single packet passing through the network interface attached to the VNF;
- filling up the eBPF maps.

5.4 VNF Workflow

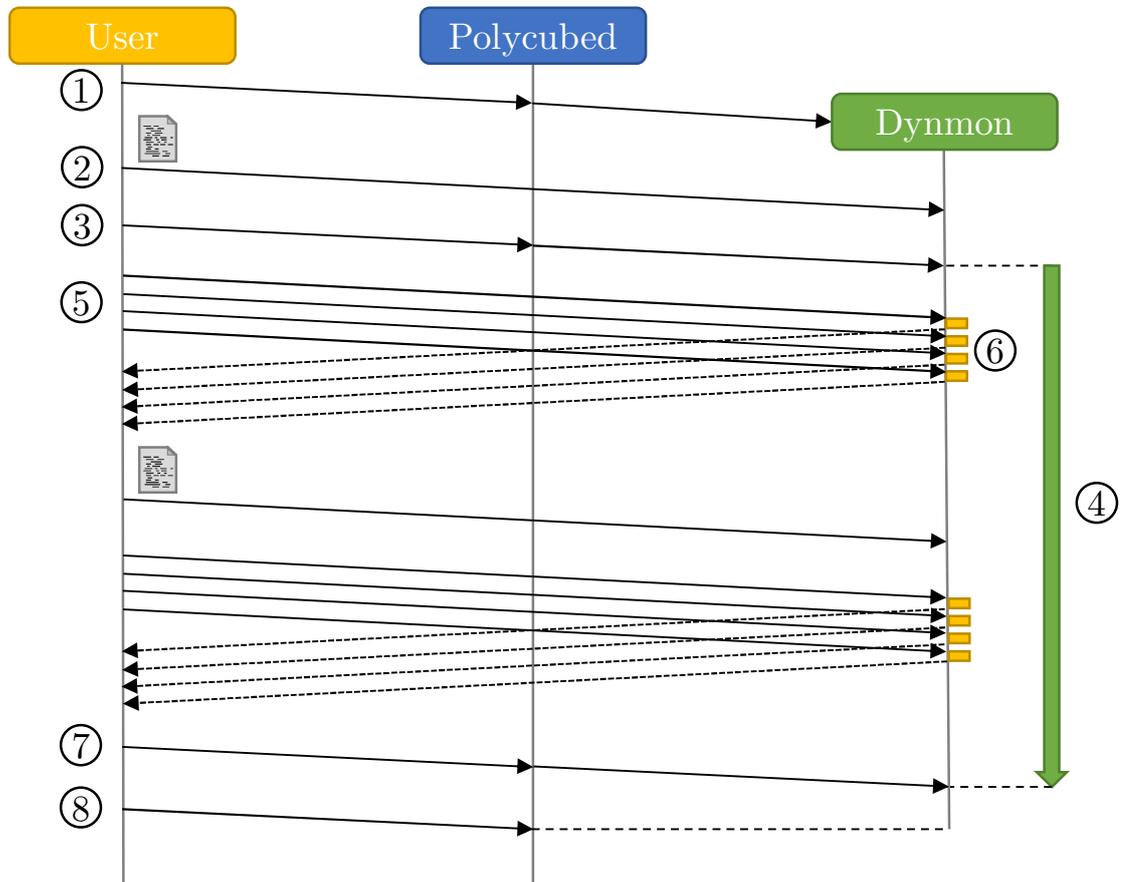


Figure 5.5: VNF workflow

Figure 5.5 depicts the workflow of the service:

1. *Service creation*, the service is instantiated by contacting the Polycube daemon; by default, it will create a basic version of the service that will not perform any analysis on incoming packets and that is not connected to any network interface. During the service creation it is possible to provide a configuration in order to create and set up the service and its data plane all at once.
2. *Data plane injection*, an external operator, such as the ASTRID Security Orchestrator, sends a representation of the data plane to be injected by

contacting the VNF through the REST interface. Once this object is received, the validity of the attached eBPF code is checked. If no error occurs during this operation, the eBPF code is injected replacing the previous one. The metadata contained in the input data plane are stored and available to be interpreted for the eBPF maps extraction. Each eBPF map declared in the eBPF code is here instantiated.

3. *Network interface attaching*, the service is connected to a network interface on the host machine, or to a port of any Polycube cubes instantiated in it, by contacting the Polycube daemon, specifying the name of the network interface. Once the VNF is connected to a network interface, its dataplane code is attached to a hook in the kernel and is ready to analyze the traffic passing through it.
4. *Data collection*, every time a packet goes through the hook, the injected eBPF code analyzed it and appropriately fills up the eBPF maps.
5. *Metric request*, the user can request the collected metrics by querying the metrics endpoint of the control plane through the REST interface.
6. *Data extraction*, the service control plane receives the metrics request: here the saved metadata are used by the Maps Extractor to get a reference to the eBPF and recursively parse their content in order to fill up the output structures. The output is then returned to the user.
7. *Network interface detaching*, the service is disconnected by the specified network interface by contacting the Polycube daemon that will take care to remove the previously injected hook. After this action, the VNF will not analyze any packet until it gets connected to a network interface again.
8. *Service deletion*, the service is destroyed by the Polycube daemon, which removes the injected hook and the eBPF maps created by the service.

5.5 YANG model description

Within the Polycube framework, YANG is the language used for defining services' structure. Each service is contained in precisely one module, where the service name is the module name.

In order to design a Polycube service, a YANG model must be created to describe the structure of the VNF; this permits Polycube VNFs to be compliant to the RESTCONF protocol.

Here we show the significant parts of the YANG model of the VNF, explaining the most important fields.

Dataplane container

The dataplane container represents the structure of the VNF dataplane; it contains:

1. *name*, a string used by the user to identify the service instance;
2. *code*, the eBPF code that runs in the service data plane;
3. *metrics*, a list of items which describe the metrics that the data plane is able to export. As an example, a possible metric in a probe that monitors the TCP traffic could be the number of TCP bytes/packets received by the probe. Each entry contains:

- *name*, the name of an exported metric;
- *map-name*, the name of the eBPF map which contains the value of the metric;
- *open-metrics-metadata*, a structure which contains the informations used to export the metric in the OpenMetrics format.

Listing 5.1: The dataplane container

```
1  container dataplane {
2      leaf name { type string; }
3      leaf code { type string; }
4      list metrics {
```

```
5     key "name";
6     leaf name { type string; }
7     leaf map-name { type string; }
8     container open-metrics-metadata {
9         leaf help { type string; }
10        leaf type {
11            type enumeration {
12                enum Counter;
13                enum Gauge;
14                enum Histogram;
15                enum Summary;
16                enum Untyped;
17            }
18        }
19        list labels {
20            key "name";
21            leaf name { type string; }
22            leaf value { type string; }
23        }
24    }
25 }
26 }
```

Metrics container

The metrics container represents the list of the exported metrics in the JSON format; it contains:

1. *name*, the name of the metric
2. *value*, an object that follows the structure and content of the eBPF map linked to the metric, expressed with the JSON format;
3. *timestamp*, a timestamp which indicates the time at which the metric has been read from the eBPF map.

Listing 5.2: The metrics container

```
1     list metrics {
2         config false;
3         key "name";
4         leaf name {
5             config false;
```

```
6         type string;
7     }
8     leaf value {
9         config false;
10        type string;
11    }
12    leaf timestamp {
13        config false;
14        type int64;
15    }
16 }
```

Open-metrics leaf

The open-metrics leaf represents the set of exported metrics in the OpenMetrics format. This resource is a string that encapsulates the open-metrics-metadata of each metric along with its name and its value.

Listing 5.3: The open-metrics leaf

```
1     leaf open-metrics {
2         config false;
3         type string;
4     }
```

5.6 Input data plane configuration

In order to configure the data plane of the VNF, a configuration has to be provided to the control plane; this configuration consist in a JSON file that follows the structure of the Dataplane generated from the YANG model's *dataplane container*.

A configuration file contains:

1. *name*, the name of the dataplane configuration
2. *code*, the eBPF code that must be injected in the Kernel
3. *metrics*, the list of metrics configurations

Listing 5.4: Input dataplane configuration example

```
1 {
2   "name": "NTP Amplification probe",
3   "code": "\r\n#include <uapi/linux/ip.h>\r\n [...]",
4   "metrics": [
5     {
6       "name": "ntp_packets_total",
7       "map-name": "NTP_PACKETS_COUNTER",
8       "open-metrics-metadata": {
9         "help": "This metric represents the number of NTP packets that has
10            traveled through this probe.",
11         "type": "counter",
12         "labels": [
13           {
14             "name": "IP_PROTO",
15             "value": "UDP"
16           },
17           {
18             "name": "L4",
19             "value": "NTP"
20           }
21         ]
22       }
23     ]
24 }
```

In listing 5.4 we can notice that for each metric configuration there are three properties:

1. *name*, the name of the exported metric
2. *map-name*, the name of the eBPF map which contains the value of the exported metric
3. *open-metrics-metadata*, a set of metadata used to convert the content of the eBPF map properly in the *OpenMetrics* format. This metadata include:
 - (a) *help*, a description which explains the meaning of the metric
 - (b) *type*, which indicates the type of the metric in the OpenMetrics format
 - (c) *labels*, custom key-value pairs attributes that will be attached to the metric in order to enable the use of Prometheus's dimensional data model.

The example presented in listing 5.4 corresponds to a dataplane configuration named "*NTP Amplification probe*"; this configuration contains an eBPF code and a metric configuration which describes a metric that the service is able to export. This metric named "ntp_packets_total" is linked with the eBPF map "NTP_PACKETS_COUNTER" which during the dataplane operation stores the value of the metric.

5.7 Metrics

Metrics are exported in two formats to make the service as flexible as possible:

1. JSON: is capable of completely reproducing the structure of the eBPF maps which are capable of containing complex structured data (C arrays, unions and structs, also nested) as well as primitive data types (integers, doubles, chars, etc ..);
2. OpenMetrics: a standard specifically designed for exposing metric data in the context of Observability, used by monitoring systems (e.g., Prometheus). This format is not able to represent complex data structures; hence only primitive data types and arrays are exported.

In order to return the metrics in the above depicted formats, the service exposes two main REST endpoints:

1. `/metrics`, which returns the collected metrics in the JSON format;
2. `/open-metrics`, which returns the collected metrics in the OpenMetric format.

Thanks to the Polycube architecture, some other endpoints are generated to allow the access to specific metrics and their fields:

1. `/metrics/{metric-name}`, which returns a single metric by providing its name;
2. `/metrics/{metric-name}/name`, which returns the name of a selected metric;
3. `/metrics/{metric-name}/value`, which returns the value of a metric;

For example, to retrieve the metric `ntp_packets_total` declared in listing 5.4, can be obtained by querying the control plane with a HTTP request like `GET .../metrics/ntp_packets_total` and an output like the following would be returned:

```
1 {
2   "name": "ntp_packets_total",
3   "value": "1860",
4   "timestamp": 1583893571617726
5 }
```

Chapter 6

Implementation

As already described in the previous chapter, the chosen architecture is the one which involves the use of a generic control plane capable of managing any monitoring eBPF code injected into the data plane. This chapter aims to provide implementation details for the aforementioned architecture and it is divided into sections that analyze the various elements that compose the solution. Finally, it will discuss the structure of a user-friendly tool created to facilitate the deployment of the service and the dynamic injection of monitoring eBPF code into it. The source code of the solution can be found on the main GitHub repository of the Polycube framework [11].

6.1 Used languages

For the implementation of the architecture, the main language used was C++, a choice forced by the fact that the entire Polycube framework is written using this language. A second language used was YANG, necessary for the creation of a service model for the automatic generation of the basic code structure. A third language used is Python, for the creation of the user-friendly tool aforementioned.

6.2 Implementation workflow

This section explains which steps were needed to implement the solution.

To create a new service in the Polycube framework it is first necessary to define a YANG model which declaratively describes each individual resource present in the service structure.

Since the structure of the YANG model has been explained in the previous chapter, this section is limited to explaining the role of the YANG model in the implementation of the service.

In the Polycube framework, the *polycube-codegen*[12] tool is used to generate the basic source code to speed up the implementation of the service.

This tool receives the YANG model of the service as input and generates all the main classes which corresponds to the resources present in the YANG tree as output.

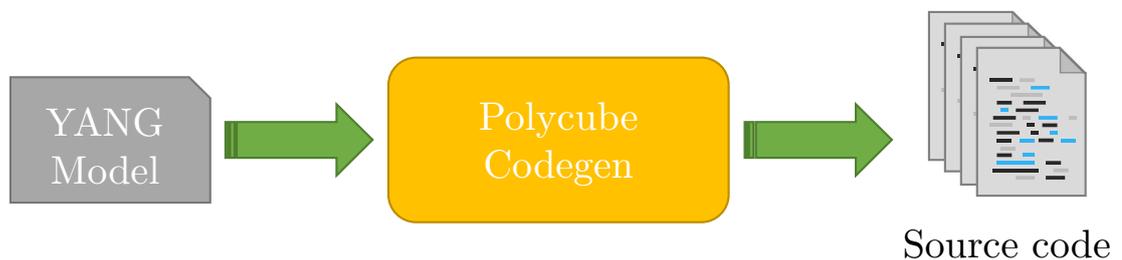


Figure 6.1: Code generation pipeline

Once the basic source code is generated, it can be modified and extended to implement the functionality of the service.

The generated source code is composed by the following classes:

1. Dataplane
2. DataplaneMetrics
3. DataplaneMetricsOpenMetricsMetadata

4. DataplaneMetricsOpenMetricsMetadataLabels
5. Dynmon
6. Metrics

Several other classes are generated to provide the service's REST interface APIs, the deserialization of the objects received from the interface and the serialization of objects sending responses in the JSON format.

In a second phase, it has been produced the source code that implements the functionalities of the Control Plane, such as:

1. the reception and the storing of the data plane configuration;
2. the eBPF code injection at runtime;
3. the extraction the information present in the eBPF maps;
4. the transformation of the extracted data into the two supported formats.

A third phase concerned the creation of eBPF monitoring code for real use cases to respond to the requests of the ASTRID project and to validate the solution.

The last phase was about the creation of a user-friendly tool named *Dynmon Injector* which allows the instantiation of a *dynmon* service on a target machine and the injection of a dataplane configuration without dealing with the Polycube command line interface.

6.3 Main classes

As explained in the previous section, the generated classes reflect the structure of the node tree present in the YANG model. Since the main structure is a tree, each class represents a node and contains all the properties of the node itself and references to its children nodes through the use of C++ shared pointers. Each generated class receives in its constructor a configuration object which may contain a value for any property of the class itself; this is mainly used by this service to be

instantiated with a pre-defined configuration which has to be applied to initialize the service.

Dynmon Class

The *Dynmon* class represents the entry point of the service: it actually contains the logic that represents the Control Plane of the service.

This class contains a shared pointer to a *Dataplane* object which represent the current dataplane of the VNF. This object is allocated when a data plane configuration is received from the REST interface or during the creation of the service itself if a configuration is provided.

The methods of the *Dynmon* class are linked to the service REST API and they mainly perform the functions necessary for receiving a configuration for the dataplane and for exporting the collected metrics.

Dataplane Class

The *Dataplane* class represents the configuration of the dataplane: it stores the eBPF code and the metrics configurations. The metrics configurations are used to identify the eBPF maps to be exported and to store the metadata needed for the translation of the metrics in the OpenMetrics format.

By default, when the service is instantiated without a provided configuration, a predefined configuration is generated where the eBPF code does no-inspection on the network traffic and no metrics are exported. As this class has been generated by the *codegen* tool [12], the constructor receives a *conf* object which may contain a non-default configuration. In this case, the configuration is retrieved by the *conf* object and stored. The *getMetricsList* method of this class returns the saved metrics configurations.

MapExtractor Class

The *MapExtractor* class is responsible for the extraction of the content of a eBPF map at runtime. This class offers the static method *extractFromMap* that receives a reference to the service whose dataplane has instantiated the map, the

name of the map and other parameters used to find the map in the set of eBPF injected chains. The output of this method is a JSON object which represent the content of the requested map.

More about how the maps content is extracted is explained in Section §6.5.

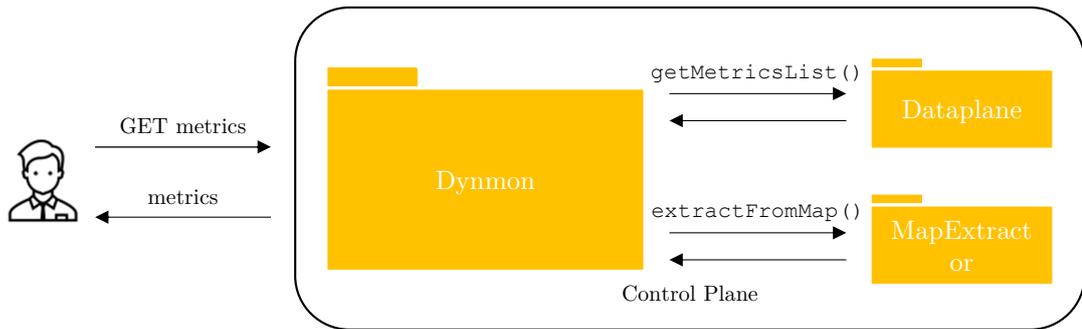


Figure 6.2: Main classes

6.4 Data plane configuration injection

As explained previously, the core aspect of this service is the capability to change the behavior of the data plane during operation by injecting different eBPF code in the Linux Kernel at runtime.

This capability exploits the use of the `reload` method exposed by the *BaseCube* class from which the *Dynmon* class derives; this method is responsible for replacing the eBPF code executed in the kernel hook to which the service is attached to.

When a dataplane configuration is sent to the service control plane, the eBPF code it contains is passed as a parameter to the `reload` function. The `reload` method tries to inject the new code in the kernel hook and, if the kernel's eBPF verifier considers it correct, the injection is successful and the eBPF code immediately starts running, analyzing the new packets which reach the service.

If the kernel’s eBPF verifier rejects the eBPF code, an exception is raised, the dataplane configuration is rejected and the previous configuration is restored.

Once the new dataplane Configuration is injected, the service is able to export the new configured metrics automatically without any other required action.

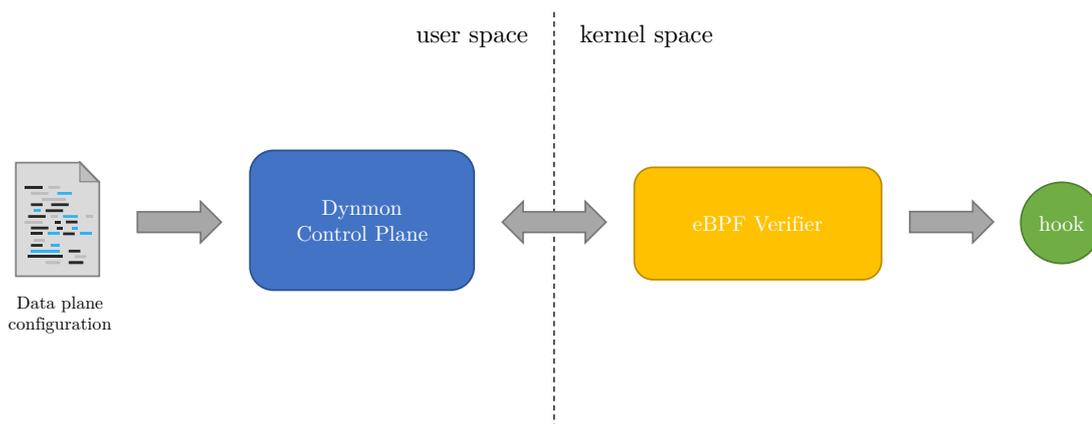


Figure 6.3: Data plane configuration injection

6.5 Data extraction

One of the main features of this service is the capability to export the content of eBPF maps in a lightweight data-interchange format: JSON.

The data extraction from the eBPF map is performed by a component called *MapExtractor*. This component exploits the use of the *TableDesc* class, a class provided by the BCC library which describes how a eBPF map is built.

The *TableDesc* object of a map is obtained by calling the `get_table_desc` method exposed by the *BaseCube* class from which the *Dynmon* class derives.

A *TableDesc* object contains a property called *leaf_desc*, a JSON object which describes the structure of the entries of the eBPF map. This JSON object has a

predefined tree structure to represent C structs, unions, arrays, enums and primitive types (such as `int`, `char`, `double`, `unsigned long long` and so on). The structure of this object is different for each one of the aforementioned C data types.

In order to extract the content of a eBPF map, the `leaf_desc` object is parsed recursively to cast the memory block of the map's entries accordingly to their structure. The parsing recursion of the `leaf_desc` JSON object first recognizes it as one of the aforementioned C data types, then calls the corresponding extraction method that will cast the memory block of a map entry to extract the contained value.

Since in the `leaf_desc` object structs and unions have nested data types for each one of their fields, the recursive method is called on each of them.

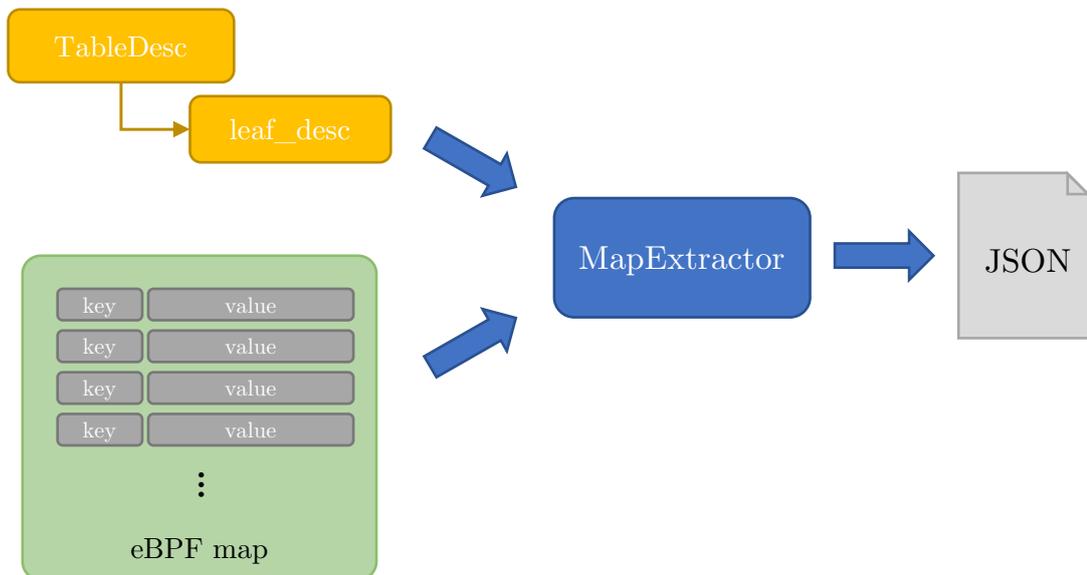


Figure 6.4: Map extraction pipeline

The static `extractFromMap` method of the `MapExtractor` class is the one used by the `Dynmon` class to get the content of an eBPF map. This method obtains the `TableDesc` object corresponding to a eBPF map and calls, for each map entry, the recursive extraction method `recExtract`. The obtained objects are grouped in a single JSON which is returned as output.

The static `recExtract` method of the *MapExtractor* class is the recursive method which identifies the type of a node obtained from a *TableDesc*'s *leaf_desc* tree and calls the corresponding extraction method.

The extraction methods used to parse the memory blocks of the maps entries are:

1. `valueFromStruct`: parses a memory block corresponding to a C struct and produces a JSON object which represents the struct as a set of key-value pairs where the keys are the names of the struct properties and the values are the results of the memory casting operation.
2. `valueFromUnion`: parses a memory block corresponding to a C union; The real type of the union is decided by the program which is using the union and cannot be known at runtime so the memory corresponding to the union value is here parsed as each possible type the union contains in order to let decide who uses the union how to handle its value. This method produces a JSON object which represents the union value as a set of key-value pairs where the keys are the names of the possible types and the values are the results of the memory casting operation.
3. `valueFromEnum`: parses a memory block corresponding to a C enum and produces a JSON object which represents the enum. The casting operation will look at the value stored in the memory block and will transform it into the name of the corresponding enum field as a string.
4. `valueFromPrimitiveType`: parses a memory block corresponding to a C primitive type (e.g., `int`, `uint`, `float`, `double`, `char`, etc..) and produces a JSON object which represents the value resulting from the memory casting operation. For each primitive type supported by eBPF, a corresponding string name is contained in the *leaf_desc* JSON object (e.g., `unsigned long long`, `signed char`, etc..). The casting operation will use this string to cast the memory block correctly and return the contained value.

Here we show some examples that explain how the various data types are represented in the `leaf_desc` object.

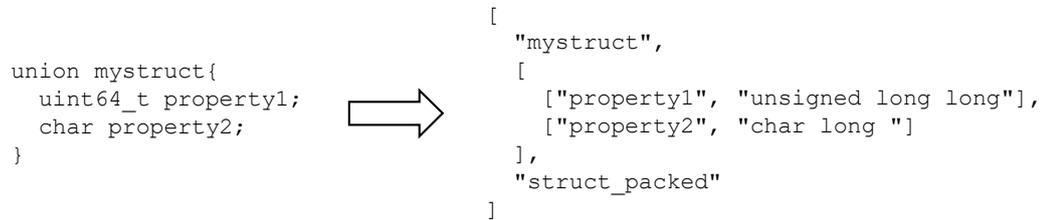


Figure 6.5: C struct to *leaf_desc* JSON

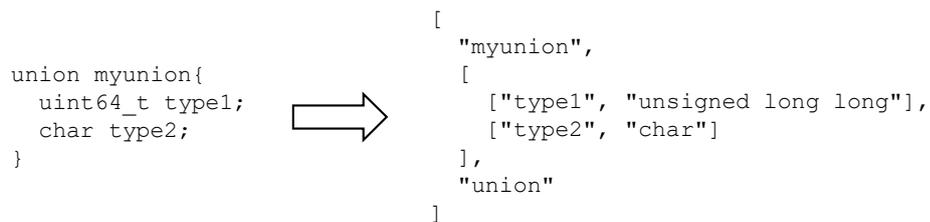


Figure 6.6: C union to *leaf_desc* JSON

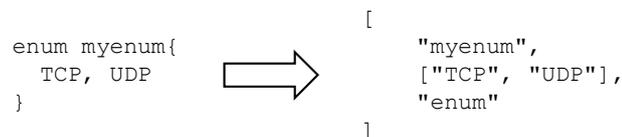


Figure 6.7: C enum to *leaf_desc* JSON

6.6 Metrics export

The collected metrics remain saved in the corresponding eBPF maps throughout the life cycle of the control plane. During the operation of the service, the control plane accesses the maps only when the REST endpoints that return their content are contacted.

When the `/metrics` endpoint is reached by a GET HTTP Request, the `getMetricsList` method of the *Dynmon* class is called; this method first gets the metrics configurations from the dataplane configuration, then, for each exportable metric, extracts the corresponding eBPF map content by calling the static `extractFromMap` method of the *MapExtractor* class. For each extracted metric a *Metric* object is created and the set of metrics is returned as a JSON in response to the HTTP Request.

Listing 6.1: JSON metrics output example

```
1 [
2   {
3     "name": "ntp_packets_total",
4     "value": "1860",
5     "timestamp": 1583893571617726
6   },
7   {
8     "name": "ntp_mode_private_packets_total",
9     "value": "3",
10    "timestamp": 1583893571617762
11  }
12 ]
```

Similarly, when the `/open-metrics` endpoint is reached by a GET HTTP Request, the `getOpenMetrics` of the *Dynmon* class is called and the same metrics extraction process is performed but, instead of producing a JSON object containing the set of metrics, the extracted metrics are transformed in the OpenMetrics format using the *OpenMetricsMetadata* data present in the metrics configurations stored in the dataplane configuration. The result of this operation will be a string which contains all the extracted metric. This string is then returned as Response to the HTTP Request.

Since the OpenMetrics format is able to represent only numeric values, only metrics with a numeric value type can be exported with this format. If an eBPF map corresponding to a exportable metric contains a set of values, the service generates a metric for each entry of the map, concatenating the entry index to the metric name.

Listing 6.2: OpenMetrics metrics output example

```
1 #HELP ntp_packets_total This metric represents the number of NTP packets
   that has traveled trough this probe.
2 #TYPE ntp_packets_total counter
3 ntp_packets_total{IP_PROTO="UDP", L4="NTP"} 1860 1583893571
4
5 #HELP ntp_mode_private_packets_total This metric represents the number of NTP
   packets with MODE = 7 (MODE_PRIVATE) that has traveled trough this probe.
6 #TYPE ntp_mode_private_packets_total counter
7 ntp_mode_private_packets_total{IP_PROTO="UDP", L4="NTP"} 3 1583893571
```

6.7 Dynmon injector tool

The *Dynmon Injector* tool is a Python REST client which communicates to a Polycube daemon running on a target machine to facilitate the creation of a Dynmon service instance and the injection of a dataplane configuration on a target machine without dealing with the Polycube command line interface. The tool has been developed mainly because the Polycube command line interface does not support the insertion of complex data input like the dataplane configuration.

The tool receives in input:

1. `cube_name`: name of the dynmon instance to handle;
2. `interface`: name of the network interface to which attach the service;
3. `path_to_dataplane`: path to a JSON file which represent a dataplane configuration which has to be injected on the target instance.

By default the tool tries to connect to a Polycube daemon running on `localhost` and listening on port 9000; a different address and port configuration can be provided by using two optional input parameters.

When the tool is launched, it checks if a dynmon instance with the provided name already exists: if it does, the tool checks if the cube is attached to a different network interface compared to the specified one; in case this condition is verified, it detaches the existing cube instance to the previous interface and attaches it to the new interface. If a service instance with the provided name does not exist, a new instance is created and attached to the specified network interface.

Finally the provided dataplane configuration is sent to the service instance control plane.

Figure 6.8 shows a flowchart which summarises the tool behavior.

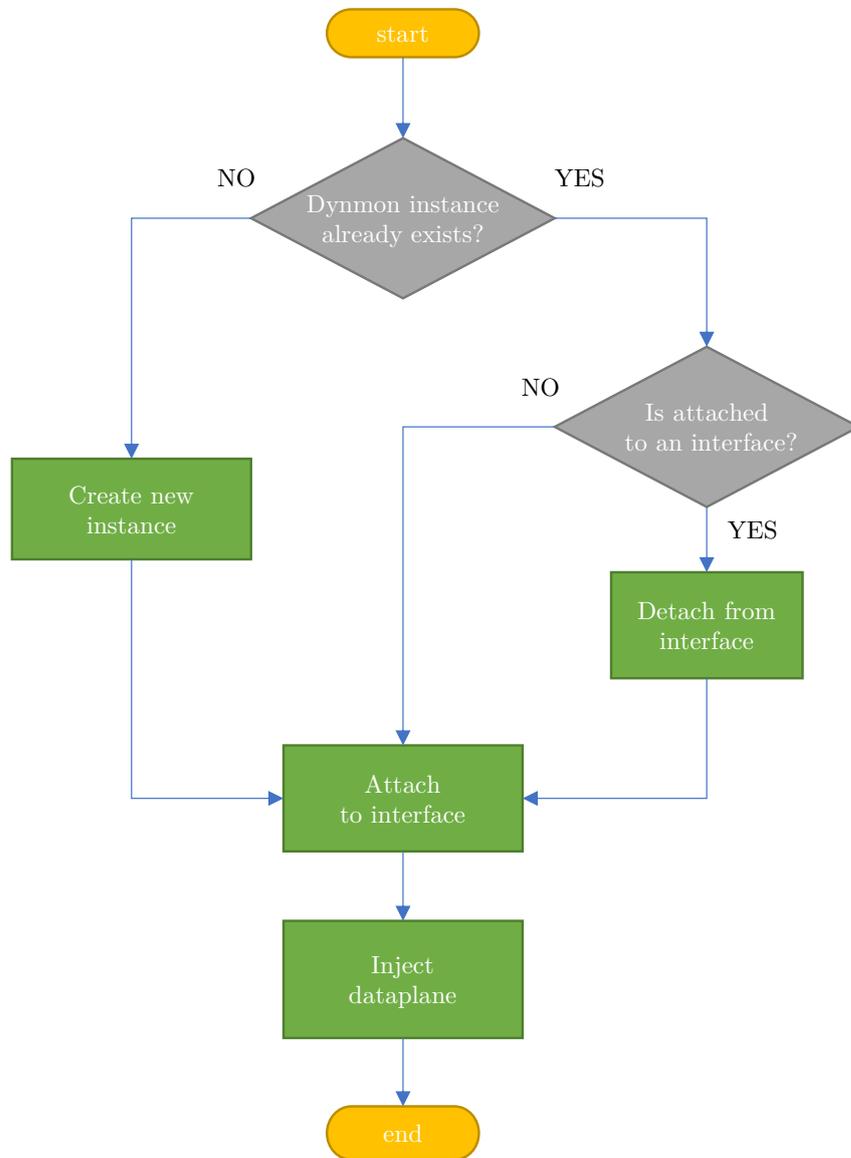


Figure 6.8: Dynmon Injector workflow

Chapter 7

Results

This chapter analyzes the performance of the proposed architecture and the goodness of the implemented solution.

Two types of validations were performed: the first concerning the prototype's performance in performing the functions for which it was designed; the second, based on the measurements of performance overhead on the monitored host in terms of network throughput degradation. The first validation is performed through the measurement of the times necessary for the injection of different dataplanes at runtime and those necessary for the extraction of the contents of the maps eBPF. The second validation is performed through the use of the *iperf3* tool [13].

All the tests have been run on an Ubuntu Server 18.04.3 LTS, kernel 4.15.0-88-generic x86_64 equipped with a Intel(R) Xeon(R) CPU E3-1245 v5 3.50GHz processor (four cores plus hyper-threading, 8MB of L3 cache), 32GB DDR4 RAM, with a installed Polycube distribution aligned to the Polycube repository master branch at commit `df3c16e`.

7.1 Injection of monitoring code

In order to test the performance of the service in terms of speed in the injection of a new dataplane at runtime, several measurements were made in which eBPF codes is provided which uses an increasing number of eBPF maps. The time required for

the injection of new code is in fact linked to the number of eBPF maps that must be created within the kernel.

In a first measurement the maps declared in the eBPF code are maps with a single entry of type `uint64_t`, in a number ranging from one to one hundred. A second measurement replaces the previous maps with maps of 10k entries of type `uint64_t`, still in a number ranging from one to one hundred. The size of the code injected in each test includes N lines of code dedicated to the declaration of maps (where N is the number of declared maps) and three lines of code for the `handle_rx` function which takes no action on the received packets. During the tests the service has not been attached to any network interface in order to have no interference on the access to the map by the dataplane.

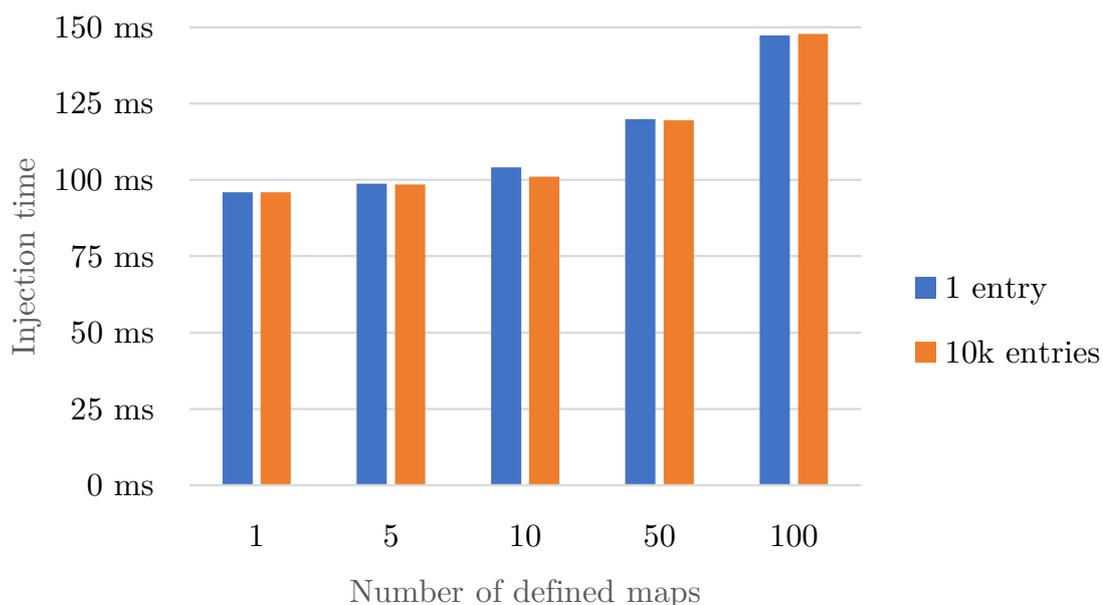


Figure 7.1: Dataplane injection time

From the graph in figure 7.1 it is possible to notice that the size of the eBPF maps that are created during the injection process does not affect the injection time while the number of maps that are created, does.

7.2 Extraction of collected metrics

In order to test the time spent by the service to extract the content of the eBPF maps it has been created a dataplane configuration in which multiple eBPF maps with different structures are defined. The structures of the eBPF maps used in this test follow the structure of Ethernet frames headers, IP headers, TCP headers and UDP headers, plus a simple `uint64_t` value type. For each one of the aforementioned structures maps of 1, 10, 100, 1k and 10k entries have been defined. The tests have been performed without attaching the service to a network interface to eliminate delays due to concurrent access to the eBPF maps by the dataplane.

The following chart depicts the result of this tests; the extraction time axis is in a base 10 logarithmic scale in order to facilitate the reading of the data and to better see how the extraction time grows proportionally to the number of map entries.

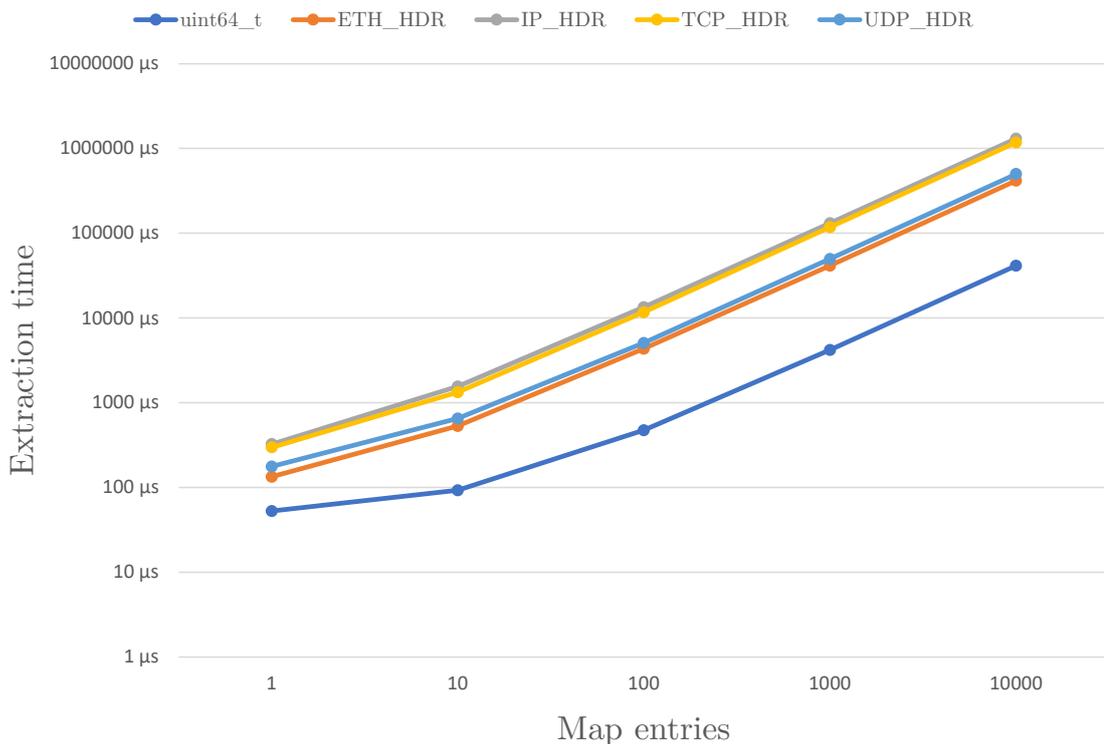


Figure 7.2: Maps extraction time

		Number of map entries				
		1	10	100	1000	10000
Map entry content	uint64_t	53 μ s	93 μ s	474 μ s	4203 μ s	41354 μ s
	ETH_HDR	134 μ s	532 μ s	4324 μ s	41439 μ s	417362 μ s
	IP_HDR	325 μ s	1549 μ s	13415 μ s	130870 μ s	1305696 μ s
	TCP_HDR	298 μ s	1325 μ s	11722 μ s	117415 μ s	1179823 μ s
	UDP_HDR	177 μ s	649 μ s	5067 μ s	49529 μ s	499384 μ s

Figure 7.3: Test results

Extraction time with traffic flow

This tests aim at evaluating the cost of the concurrent access to the map from both dataplane (which produces data) and the control plane (which reads it and deliver on connected clients). In fact, the kernel has the precedence over control plane, given that it runs at higher privilege. Therefore, particularly in presence of large traffic, we expect that the control plane access to the map is considerably delayed, with the consequent impact in reading the data. Finally, it is worth noticing that the access time to a single entry may have a huge impact in case of large tables, as the total amount of time for reading all the data can be in the order of seconds, which results in a possible non-coherence of the data read returned by the dataplane.

The tests have been performed by creating a dataplane configuration where the eBPF code saves in a eBPF maps a data structure for each TCP session it discovers. A map entry contains the source and destination IP address, the source and destination port and a counter of packets traveled through a TCP session.

The iperf3 has been configured to generated traffic using only one tcp session in order to maximize the concurrent access to the eBPF map by the dataplane and the control plane.

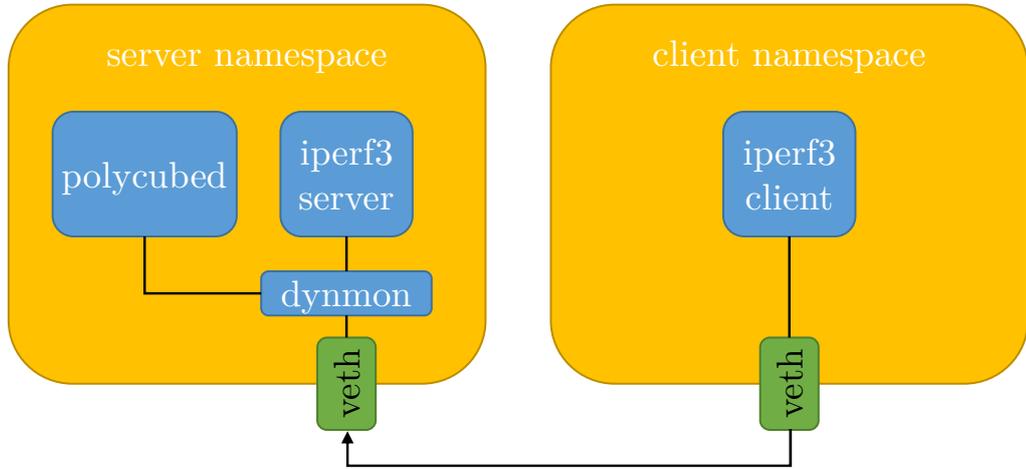


Figure 7.4: Testing setup

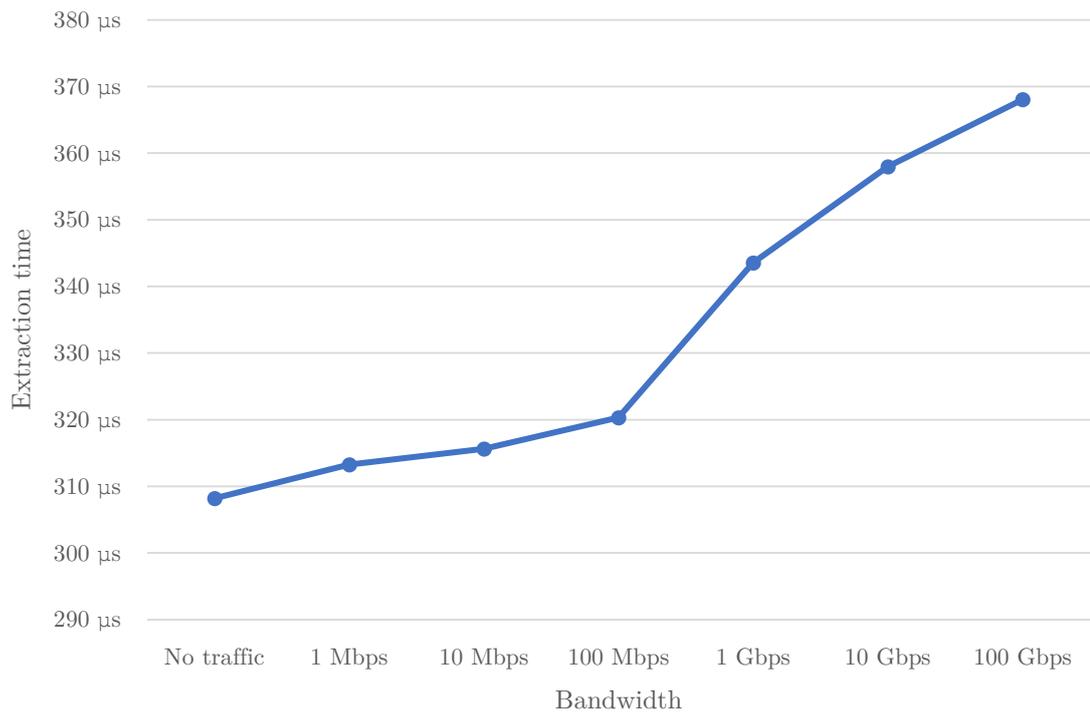
The iperf3 client and server have been instantiated in two different Linux namespaces, connected by a virtual link. The Polycube daemon has been instantiated in the same namespace of the iperf3 server and an instance of the Dynmon service has been attached to the virtual interface of this namespace.

In these tests the bandwidth parameter of the iperf3 client has been used to generate traffic from 1 Mbps to 100 Gbps. In order to measure the maximum achievable throughput by the physical machine on which these tests are run, a preliminary test has been done setting the iperf3 bandwidth cap to 100 Gbps without having the Dynmon service instantiated. A maximum value of 72.2 Gbps has been measured.

		Iperf3 bandwidth						
		0 Mbps	1 Mbps	10 Mbps	100 Mbps	1 Gbps	10 Gbps	100 Gbps
Extraction time		308 μ s	313 μ s	316 μ s	320 μ s	344 μ s	358 μ s	368 μ s

Figure 7.5: Extraction time results

Iperf3 bandwidth						
	1 Mbps	10 Mbps	100 Mbps	1 Gbps	10 Gbps	100 Gbps
Achieved throughput	983 Kbps	9.29 Mbps	91.4 Mbps	986 Mbps	9.68 Gbps	66.7 Gbps

Figure 7.6: Throughput results**Figure 7.7:** Maps extraction time with incremental throughput

These tests confirm what was expected: with lower network traffic the effects of the concurrent access on the eBPF maps by the dataplane and the control plane are reduced. Moreover, it is clearly proven that with higher network traffic the access to the eBPF maps of the control plane suffers a delay due to the lower access privileges.

Chapter 8

Conclusions

The proposed solution has shown promise and could have uses that go beyond integration with the ASTRID project, since at the moment there are not many alternative tools that can offer the possibility of injecting eBPF code directly into the kernel to inspect and manipulate the network traffic in a quick and easy way.

8.1 Possible improvements

At the end of the work, many possible improvements were thought of.

By delving deeper into the BPF ecosystem, an alternative method was found to dump the contents of the eBPF maps. This method consists of exploiting BTF (BPF type format), a metadata format that encodes debug information related to BPF program and maps. This method already used by *bpftool*[14], a tool which allows to analyze varied information on BPF programs and maps already injected into the kernel, through a simple command line interface. The use of BTF would greatly simplify the logic of extracting data from maps, significantly improving the performance of the service.

Another important improvement to the service can be the enabling of the injection of multiple codes of data planes in cascade which would allow the addition of monitoring code in a modular way without replacing the code already injected.

Furthermore, the dataplane of the virtual network function currently only acts on the incoming traffic of the interface to which it is connected, while the outgoing traffic is not considered.

The addition of a second pipeline for outgoing traffic would therefore allow you to monitor different parameters on incoming and outgoing packets, enabling more complete traffic analysis.

As for as the metrics exported in OpenMetrics format, not all possible metrics types are currently managed; a necessary improvement would be to give full support to the OpenMetrics standard, allowing to export metrics of the **Histogram** and **Summary** types.

In addition to the work already implemented and the possible improvements listed above, the theme of how to provide the service with monitoring codes ready to be injected starting from the monitoring policies to be applied remains open. A continuation of the work carried out could be the creation of a service that is capable of automatically generating the eBPF codes and the metadata that make up the structure of the dataplane to be injected, starting from an abstract description of the network parameters to be monitored.

Bibliography

- [1] *The Astrid Project*. URL: <https://www.astrid-project.eu/>.
- [2] *The Polycube Framework*. URL: <https://github.com/polycube-network/polycube>.
- [3] *LLVM*. URL: <https://llvm.org>.
- [4] *BCC*. URL: <https://github.com/iovisor/bcc>.
- [5] *Prometheus*. URL: <https://prometheus.io/>.
- [6] *Kubernetes*. URL: <https://kubernetes.io/>.
- [7] *Grafana*. URL: <https://grafana.com/>.
- [8] *OpenMetrics*. URL: <https://openmetrics.io/>.
- [9] *bpftrace*. URL: <https://github.com/iovisor/bpftrace>.
- [10] *Sysdig Falco*. URL: <https://sysdig.com/blog/sysdig-falco/>.
- [11] *The Dynmon service*. URL: <https://github.com/polycube-network/polycube/tree/master/src/services/pcn-dynmon>.
- [12] *Polycube-Codegen*. URL: <https://github.com/polycube-network/polycube-codegen>.
- [13] *iperf3*. URL: <https://github.com/esnet/iperf>.
- [14] *bpftool*. URL: <https://github.com/torvalds/linux/tree/master/tools/bpf/bpftool>.
- [15] *BPF in Linux Kernel source code*. URL: <https://github.com/torvalds/linux/tree/master/kernel/bpf>.

BIBLIOGRAPHY

- [16] *Cilium BPF reference guide*. URL: <https://cilium.readthedocs.io/en/latest/bpf/>.

Acknowledgments

My acknowledgements go to Prof. Fulvio Risso, who offered me the opportunity to work on an innovative project and to go beyond what I have learned in these years of study. Thanks to Alex Palesandro for the time we shared, the help received and all the things he taught me since the beginning of this journey. Thanks to Sebastiano Miano for the hints and the knowledge that he has handed down to me.

Last but not least, I thank my family for the support they have given me and all the friends who have shared with me these years of study, laughs and satisfactions.

Thank you all.

Michel