Alexandre Serras
Gonçalo Leal
Pedro Duarte
Vasco Regal

ORWELL
Monitorization Platform for a 5G Testbed

Keywords 5G, Monitoring, Testbed, Prometheus, Grafana, OSM, VNF.

Abstract 5G networks are fiercely developing and so are the vertical applications of this technology, such as Automotive, 4.0 Industry or Public Protection and Disaster Relief. To develop new 5G applications, researchers need to test them. However, the cost of 5G infrasctructure and the required expertise to assemble a fully functional testbed may be a barrier that some cannot overcome. This led to the creation of semi-public and public 5G testbeds with state-of-the-art technology which allow NetApp developers to have access to the required infrastructure to test their applications without having to worry about allthe deployment process. These tests require a controlled environment, thus all the infrastructure has to be monitored. This work discusses an approach that complies with 5GASP directives and centralizes all the information on a unique time-series database. Moreover, the proposed methodologies were implemented in a proof-of-concept product, demonstrating the potential of the proposed approach on a working testbed.

Contents

List of Figures

List of Tables

Glossary

5G	5th Generation Mobile Network
5GASP	5G Application & Services Experimentation and Certification Platform
API	Application Programming Interface
CI	Continuous Integration
CD	Continuous Delivery
CD	Continuous Deployment
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
CSV	Comma-separated Values
ETSI	European Telecommunications Standards Institute
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
ITAv	Instituto de Telecomunicações de Aveiro
JSON	JavaScript Object Notation
MAE	Mobile Automation Engine
MANO	Management and Orchestration
NetApp	Network Application
NFV	Network Function Virtualization
NSM	Network Security Monitoring
OS	Operating System
OSM	Open Source MANO
SD	Service Discovery
SNMP	Simple Network Management Protocol
RAM	Random Access Memory
RAN	Radio Access Network
REST	Representational State Transfer
TCP	Transmission Control Protocol
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
XML	Extensible Markup Language

CHAPTER 1

Introduction

1.1 5G testbed

Before embarking upon a full-scale project, researchers conduct pilot studies that evaluate feasibility, computational costs and potential problems regarding their Network Application. These tests are held in a controlled environment over state-of-the-art infrastructure called testbed. 5G testbeds offer complete 5G system functionalities such as a 5G Core and 5G Radio Access Network. Besides the network infrastructure and since 5G applications are deployed on virtual machines or containerized environments based on the cloud, 5G testbeds provide sets of servers that host several VNFs. Allied with all this infrastructure, a fully functional testbed also grants a trustworthy Continuous Integration (CI)/Continuous Deployment (CD) pipeline for Network Applications (NetApps) deployment on the test environment.

1.2 How ITAv's testbed works

As described above, 5G applications are deployed through Virtual Machines (VMs)s, which means ITAv's testbed infrasctructure is, pragmatically, a cluster allied with state-of-the-art 5G Systems provided by Huawei[1]. In order to maintain and manage this cluster ITAv uses OpenStack[2] as their Virtual Infrastructure Manager. Moving deeper into 5G requirements, the management and orchestration tool used is Open Source MANO and to control all the 5G related infrastructure ITAv uses Huawei's eSight[3] and MAE[4]. ITAv also provides a well-defined and maintained CI/CD pipeline that validates NetApps and delivers the results to the developers.

1.3 5GASP

5GASP is a European project that aims at creating a European 5G testbed to shorten the idea-to-market process. This project was born from an alliance between Public and Private entities. The main goal of this project is to create a fully automated and self-service testbed in order to allow the development and testing of new NetApps built using 5G Network Function Virtualization (NFV) based reference architecture. One of the main objectives of 5GASP is providing CI/CD pipelines for these innovative NetApps. Monitoring is an important piece of any CI/CD pipeline thus the necessity of developing a monitoring tool for this testbed emerged. This document will discuss a monitoring solution that could be integrated in 5GASP's CI/CD pipeline. Since this project brings together various entities, the need arose to create standards for all the tools developed and used. For monitoring tools the standards are Prometheus[5], for data storage and querying, and

[1] https://e.huawei.com/en/
[2] https://www.openstack.org/
[3] https://e.huawei.com/en/products/software/mgmt-sys/esight
[4] https://www.huawei.com/en/news/2018/11/auto-driving-mobile-network-solution-mae
[5] https://prometheus.io/

Grafana[6], for data visualization.

1.4 Motivation and Goals

The motivation behind the development of this project was to, through the use of existing tools, prove that we can develop a monitoring tool that relies on a toolset to monitor a 5G application testbed. Besides this, network's performance and security monitoring, system's security liabilities detection and 5G infrastructure monitoring tools should also be addressed.

In the end, this tool shoul follow some directives:

- **Be non intrusive** - There are already some solutions for monitoring testbeds. However, in order to make NetApp developers trust the testbed and test results, the less interference in the tests, the better.
- **Transparency** - Another key point mentioned is that, regardless of the toolkit used, the information must be available through an uniform access interface that offers all the information in a unique format.
- **Open Source** - A big part of this project relies on open source tools, so this project must give back to the community what the community has given it. Besides, 5GASP aims to speed up the research process offering tools to test new 5G applications, so this project must follow the same purpose.
- **Modularity** - Testbeds' infrastructures have big and complex architectures, sometimes with several system admins and more than one project on the same infrastructure. For that reason, this project must be as divisible as possible, so system admins can choose only what makes sense in their reality. Another advantage of having a modular and a micro-service oriented architecture is the possibility of adapting some modules without having to study all the code.

[6]https://grafana.com/

CHAPTER 2

State Of The Art

In order to understand the problem we wanted to solve and how to solve it, we searched for information on the topic and selected a few papers on the subject. In this section, we will present a summary of these papers along with some advantages or disadvantages of the solution proposed on them.

2.1 Monitoring and Controlling Research Experiments in Cloud Testbeds [1]

Sonar, is a synchronous system to simplify the monitoring of a cloud testbed infrastructure which was designed for cloud testbed monitoring. It log traces, times series data for long periods of time, distributes and automatically installs on destination servers. In addition to storing metric readings, Sonar has the capability of storing application logs.

Sonar offers a simple interface to access both live and historical monitoring data.

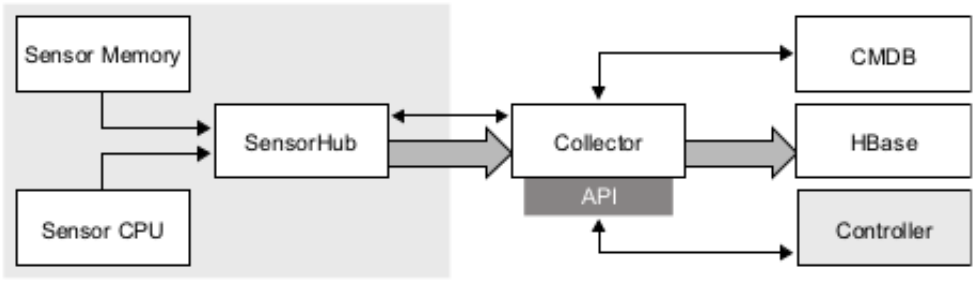

Figure 2.1: Sonar Architecture

2.1.1 Advantages of Sonar

- Allows for a flexible monitoring of timeseries and application log data without extensive configurations of the type of sensors and hosts that support cloud short-lived VMs
- There is no need to reduce the number of samples or continuously delete data.
- Basic access to data by processing CSV files.
- Large amount of data can be processed by MapReducers
- Flexibility and horizontal scalability

2.2 Monitoring of a virtual infrastructure Testbed [2]

This paper presents a distributed SNMP based monitoring tool called SBLOMARS that uses JAVA-sockets and XML files, capable of effectively monitor virtual testbeds. Its a very standard and generalized tool, with auto-configurable resource scheduling algorithms capable of adapting to network and resource constraints and offers easy data exporting functionalities for 3rd party systems. Each node in the system runs an instance of the tool. All nodes query a system agent responsible for finding and identifying the nodes in the network. The main advantages of this method are the scalibility properties.

2.3 A Testbed Evaluation for a Privacy-Aware Monitoring System in Smart Home [3]

This paper suggests a monitoring system for smart homes. The challenge is to collect data from different sensors in the network while trying to reduce energy consumption and the number of packages sent and keeping the privacy of those packages in mind.

2.3.1 Reduced network traffic

The implementation follows a threshold-oriented methodology - packages are not sent periodically, they are sent when the difference between the current value and the last sent exceeds a predefined value. This allowed more than 90% reduction in sent packages while still preserving an acceptable accuracy (95%). Another concern is data compression, which they implemented with a wavelet compression algorithm.

2.3.2 Security and privacy

Three encryption algorithms were tested - AES, RC5 and Skipjack. AES was by far the most energy-efficient algorithm, while RC5 had the least ROM/RAM usage. AES was also considered the most secure option and was chosen for the implementation.

2.4 NetGraf: A Collaborative Network Monitoring Stack for Network Experimental Testbeds [4]

NetGraf is a network monitoring stack that uses open-source tools to monitor, collect, aggregate and display metrics. The data is shown on a single Grafana dashboard. This solution has its focus on network metrics instead of VM or VNFs metrics, nevertheless it was one of our main monitoring system examples.

This solution is divided in three main modules: Network and Application, Data Collection and Aggregation and Visualization. These are easily identified by looking at the architecture on 2.2.

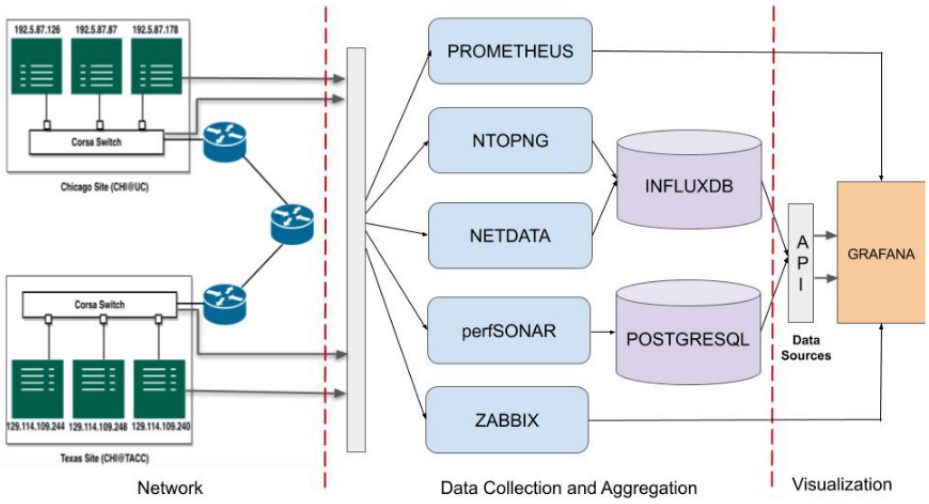

Figure 2.2: NetGraf's Architecture

2.4.1 Problems of NetGraf

The information stored by NetGraf is not centralized as it has 4 different databases and therefore we do not have a centralized API from where we can obtain metrics. Instead, we have some metrics available in Prometheus, others in the API and others available from Zabbix. In the future, the decision of having several data sources in different formats may be a liability if, for example, someone decides to change the data visualization tool.

CHAPTER 3

Conceptual Modelling

This chapter provides further information regarding the reasoning behind the development and the problem addressed by this project. The terminology used through this document and the project's requirements are also described in this section.

3.1 Problem

As 5G approaches a very high maturity level, testing and validation of 5G solutions has become of utmost importance, creating a need for well-structured testing environments where the CI/CD pipeline is one of the main pieces.

To guarantee the quality of the running tests we need to be constantly monitoring every aspect that can interfere with them, such as the network and the 5G infrastructure status, as well as monitoring the 5G application itself through the VNF's behavior since their variation could be caused by a malfunction on the system being tested. Given that the addressed testbed is open to anyone who wants to test their NetApp, some security measures should be considered to prevent attacks from the inside and to assure that no tests are affected by outside attacks.

3.2 Terminology

3.2.1 Orwell

Orwell is the name of the solution created. Throughout this document, the solution may be addressed as Orwell.

3.2.2 Intrusive

The process of directly accessing a VNF to collect metrics from it, is deemed intrusive in Orwell's context.

3.2.3 Non-Intrusive

In the context of Orwell, when we gather the metrics of a VNF without having to input them, the solution is called non-intrusive, and these are the ones we plan to adopt at work.

3.3 Actors

After a careful analysis of the scope of this project two actors were identified. These actors are enumerated and described below:

- NetApp Developer: Represents the person that wants to deploy the NetApp in the testbed. He wants to know if his NetApp is successfully deployed and how it is performing.
- ITAv's System Admin: Represents the person responsible for ITAv 's systems, which include the 5G testbed. He would like to know how the VNFs are performing. This can be done via our Grafana dashboard. He would also like to receive an alert when one of the VNFs starts to behave abnormally.

3.4 Use Cases

In this section we will take a closer look at the use cases for Orwell. A diagram is provided along with some notes on each use case.

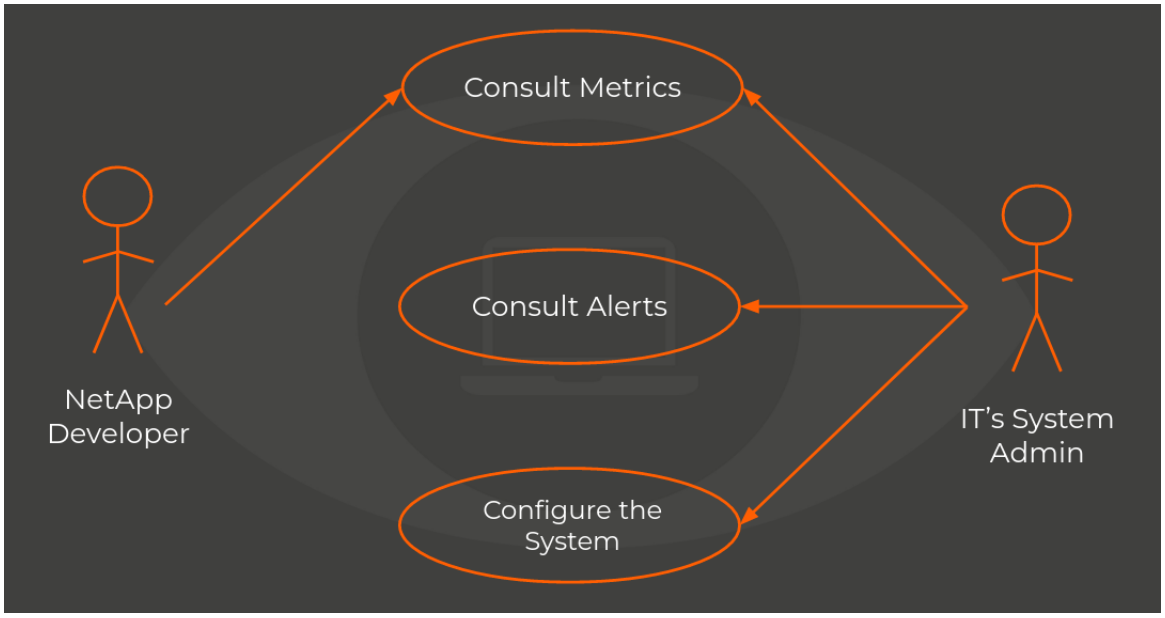

Figure 3.1: Use Cases Diagram

- Consult Metrics - both actors are able to perform this action. However, a NetApp developer is only able to consult metrics related to its VNF while the system admin can consult metrics associated with any VNF, with the network or the 5G infrastructure.
- Consult Alerts - the system has several alerts, being them related to a VNF's status, the network or security alerts. The system admin has access to all these alerts via Slack[1].
- Configure the System - the system admin can consult and change configurations through Orwell's dashboard.

3.5 Requirements

3.5.1 Functional

- Collect the metrics of the machines and the network, including 5G metrics
- Mandatory use of Prometheus and Grafana (5GASP directives)
- Send security related reports to the System Admin
- Send alerts to the System Admin when one of the VNFs starts to behave abnormally

[1] https://slack.com/

3.5.2 Non-Functional

- Scalability

  The system has to be easily scalable to deal with possible demand changes, in order to dynamically alter its performance.

- Modularity

  The system's architecture must be micro-service oriented, allowing to alter a module without having to alter the others.

- High availability

  A monitoring system cannot fail or the whole infrastructure may be compromised.

- Easy integration of new metrics collectors

  The main goal of the project is developing a toolkit, however this toolkit may not fit all purposes. Therefore, the integration of new tools must be straightforward.

CHAPTER 4

Procedure and Implementation

4.1 System Architecture

Looking at the system architecture figure 4.1, Orwell shows various components, each of which serves a distinct purpose in the final solution.

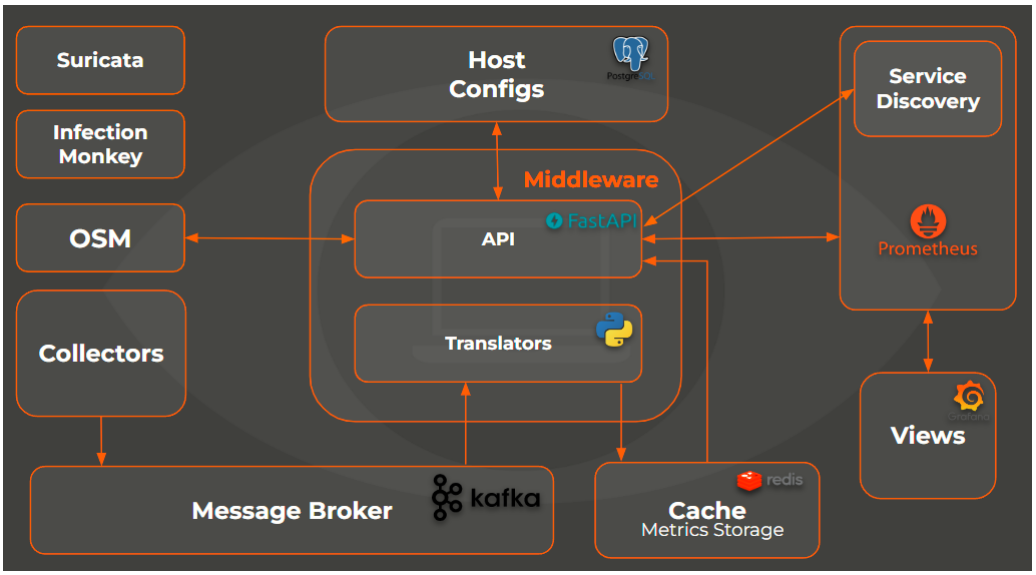

Figure 4.1: System Architecture

Our system has a micro-services architecture, based in several independent modules that work together. We wanted to create a system where the information was centralized in a unique format and stored in a unique database cluster.

4.1.1 OSM

Open Source MANO (OSM) is an open-source NFV management and orchestration tool. This Management and Orchestration (MANO) is aligned with the standards upheld by the European Telecommunications Standards Institute (ETSI). A MANO is the responsible for managing and orchestrating 5G network slices, in other words, it is responsible for dynamic management such as initiating, terminating and monitoring a network slice and its VNFs for the purpose of scaling or auto-healing in real-time.

It will be utilized by the solution to determine which VNFs are active and must be monitored.

4.1.2 Collectors

In this subsection some tools and mechanisms involved in the metrics collection process will be addressed.

Operating System Image

An Operating System (OS) image is a file that contains the OS executable and any data files that might be related to the programs. The OS imaging software tool starts by capturing the OS image, which is then deployed to systems as and when required.

The OS images created run Ubuntu 20.04 by default with one of our metrics exporters configured.

Prometheus' Node Exporter [1]

Prometheus' Node exporter is one of the exporters made available from the Prometheus' community, as well as the only used from that list. It gathers many useful metrics for monitorization of a regular machine, such as CPU, RAM or disk usage, network metrics and information about running processes.

It was built by modules, so it can be configured to only collect the metrics we are interested in.

Telegraf [2]

Telegaf is a time-series database. It was created to store time series data with a very flexible data model. Similar to Prometheus' Node Exporter, it allows the extraction of CPU, RAM, disk, and other metrics from a machine, but metrics gathering process is different from Node Exporter's method. This tool as out-of-the-box integration with Kafka, making sending metrics through it an easy process.

Gnocchi [3] Gnocchi is a time-series database which was created to store time series data with a very flexible data model, which is used by OpenStack as its default database, where it stores Virtual Machines' metrics.

In ITAv's OpenStack, Gnocchi always has data related to every machine (VMs and VNFs) running on the stack, but the metrics available are limited in quantity and quality, since the time interval reaches some hours between metrics sometimes. Nevertheless, when the OS image is unknown or does not have any collector associated with it, we use Gnocchi to be able to present some metrics related to every VNF.

PerfSonar [4]

PerfSonar is used for network testing. It can be installed in any machine, making it a measurement node and afterwards we can specify which tests we want to perform between the available nodes. Tests include generic ones such as speed or latency but there are also more specific options such as testing Amazon Web Services' S3 connectivity.

It comes with many different installation options, from which we selected PerfSonar Toolkit and PerfSonar Testpoint. The first one works as the main node, offering a scheduler API, which allows us to configure the tests dynamically, and an archival API, which allows us to read test results. The testpoint is a lighter version which we install in any other required machine, enabling it to be a target for the tests.

eSight [5]

eSight is Huawei's Operations and Maintenance system which ITAv uses, since all the hardware is provided by Huawei, to manage the 5G Core and Radio Access Network (RAN), consulting information about the state of the network devices like routers or switches and information about the RAN like what is going through each antenna.

[1] https://github.com/prometheus/node$_e$*xporter*

[2] https://www.influxdata.com/time-series-platform/telegraf/

[3] https://gnocchi.osci.io/index.html

[4] https://www.perfsonar.net/

[5] https://e.huawei.com/en/products/software/mgmt-sys/esight

However, to this date, this platform is not yet fully functional inside of ITAv and we ran into some problems, as features provided in the platform's graphical interface are not available through the API and vice versa. Huawei's documentation was also misleading, claiming to have features, such as direct access to network devices' metrics, that could not be accessed.

In the end, we were not able to collect any RAN metrics, but we did collect a few from the 5G core which allowed us to present a proof of concept.

4.1.3 Message Broker

A message broker is a software that allows services and applications to interact via messages. These messages should follow a specific protocol, so that the communicating ends understand each other, but the format of a message is not relevant for the broker. It has three basic and fundamental notions: Producers, Topics and Consumers.

In our context, a message broker is in charge of storing the metrics provided by a collector, arranged by topics depending on the associated exporter. A translator is then in charge of requesting messages from a specific topic.

Kafka [6]

Apache Kafka is a distributed publish-subscribe messaging system that handles data from many source systems and makes it available in real time to destination systems. Kafka facilitates the asynchronous data exchange between processes, applications and servers.

4.1.4 Middleware

Our middleware is the most important module of the system, because it makes everything communicate and work as it should. All metrics have to go through our middleware before being exported to Prometheus. This allows us to easily change our storage location and/or system.

API

Our API provides all the information needed to monitorize the system and acts as an orchestrator for all the components in the system. From all the packages used in the development of this API there are two that we would like to highlight:

- Fast API [7] - FastAPI is an open-source Python framework to quickly develop APIs. It is known for its fast performance allied with a very intuitive and quick flow of developemnt. This framework also follows the Open API Standard and offers great support for JavaScript Object Notation (JSON) Schemas and Typing, generating automatic endpoint documentation.
- SQL Alchemy [8] - SQL Alchemy is a Python toolkit to create Object Relational Mappings, a process where SQL Alchemy maps relational databases' entities to python classes, allowing for an object oriented manipulation of database records. In addition, SQL Alchemy offers an engine class, which creates an abstraction layer on all database operations. With this in mind, once the data model is created, records can quickly be manipulated simply using Python methods and objects.

Translators

They interact with the message broker to retrieve metrics from a certain topic, translate them into Prometheus format, and transmit the translated metrics to the system's cache.

[6] https://kafka.apache.org/
[7] https://fastapi.tiangolo.com/
[8] https://www.sqlalchemy.org/

4.1.5 Host Configs

This database will be used to record information about the VNFs that will be monitored by the system, such as the metrics collector installed in their OS Image.

PostgresSQL [9]

PostgreSQL is a relational database, also open-source, which allows the persistence of data in tables. It has a strong reputation due to performance and availability features.

4.1.6 Cache Metrics Storage

Metrics that have previously been translated are stored here until Prometheus consumes them, after which they are deleted.

Redis [10]

Redis is an open-source NoSQL in-memory database, providing a set of structures to store data. Redis offers high availability, fast reads and great horizontal scaling making it a great tool for a cache service.

4.1.7 Storage and Visualization

Prometheus

It is an open-source project which is able to store metrics in a time-series database and exposes querying interfaces for easy integration.

One of its most helpful characteristics is the extensive list of exporters that it provides for using without effort with Prometheus. Those include exporters for hardware metrics, message broker metrics, HTTP server metrics and many others.

This tool comes from the functional requirements and is responsible for storing and querying all the metrics of our project. It comes with many useful functionalities, such as service discovery of metric exporters and configuration of alerts via many popular options such as email or Slack. The service discovery functionality allows to dynamically determine which VNFs or networks should be monitored, instead of hard-coding the addresses of all the collectors in the configuration file.

Grafana

Grafana is also an open-source platform that allows users to visualize metrics from different sources, such as Prometheus, allowing the creation of custom monitoring dashboards. It offers a wide diversity of elements such as line and bar charts, gauges, or simple numbers and the possibility to create queries to common data sources so as to select the data that will be represented. More than its rich graphical interface, it also comes with an extensive API, making it easy to automate any action that can be done manually.

Through this tool, Orwell users are able to interact with the metrics collected as well as creating alerts, based on threshold values, for popular services such as email, Slack, Kafka and custom webhooks.

4.1.8 Security

Suricata [11]

Suricata is a Network Security Monitoring (NSM) application that examines and processes network traffic using sets of community-created and user-defined signatures (also known as rules).

Suricata runs as a passive Intrusion Detection System (IDS) by default, scanning for abnormal activity on a server or network. It will produce and log alarms to be investigated further. It may also be set up as an active Intrusion Prevention System Intrusion Prevention System (IPS), logging, alerting, and entirely blocking network traffic that fits particular rules which are classified into four types:

[9] https://www.postgresql.org/
[10] https://redis.io/
[11] https://suricata.io/

- alert - writes in the log file what happened and what package created a certain alert
- pass - specifies packages that do not need to be inspected by Suricata
- reject - resets the transmitted Transmission Control Protocol (TCP) packet and discards it (only in IPS mode)
- drop - immediately interrupts packet processing, generates an alarm and times out the TCP connection (only in IPS mode)

This tool will be used to monitor the network from a security point of view.

ElasticSearch [12]

Elasticsearch is a distributed, free and open search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured.

ElasticSearch is a server that handles JSON queries and returns JSON data objects, allows to save, search, and analyze enormous amounts of data in real time and provides results in milliseconds. It stores and searches data using document-based frameworks and Representational State Transfer (REST) APIs so documents are the fundamental unit.

Kibana [13]

Kibana is a tool for viewing ElasticSearch that provides various graphs in real time from Suricata's data.

Infection Monkey [14]

Infection Monkey is yet another secuirty tool, developed by Guadicore to simulate network exploits and report vulnerabilities. To use this tool, an Infection Monkey server must be setup in a node of the target network, with access to a MongoDB Database.

To interact with the tool, a Web application is used. Infection Monkey provides an highly customizable environment, allowing tests to be run with a wide range of different exploits, configurable dictionary attacks and even precise victim selection. Once running in "Island" mode, Infection Monkey will keep propagating from discovered machines (starting on the server where the instance is running) and will eventually generate a report of the results. There is also support for import and export of configurations.

Infection Monkey is used to find flaws and unsafe situations inside the testbed.

[12] https://www.elastic.co/
[13] https://www.elastic.co/kibana/
[14] https://www.guardicore.com/infectionmonkey/

To keep track of the status of the targets being monitored by the system, Orwell's middleware persists hosts, names of images and collectors data in a PostgreSQL database. This allows Orwell to make queries regarding which OS Images are being used by which hosts and which collectors are installed in said images. This database contains critical information for a wide range of scenarios in the REST API, like, for example, finding Hosts that do not have an installed exporter and therefore rely on Gnocchi Puller or to keep an updated persisted list of the targets found by OSM.

4.2.1 Domain Model

The relational schema follows the model presented in Figure 4.2.

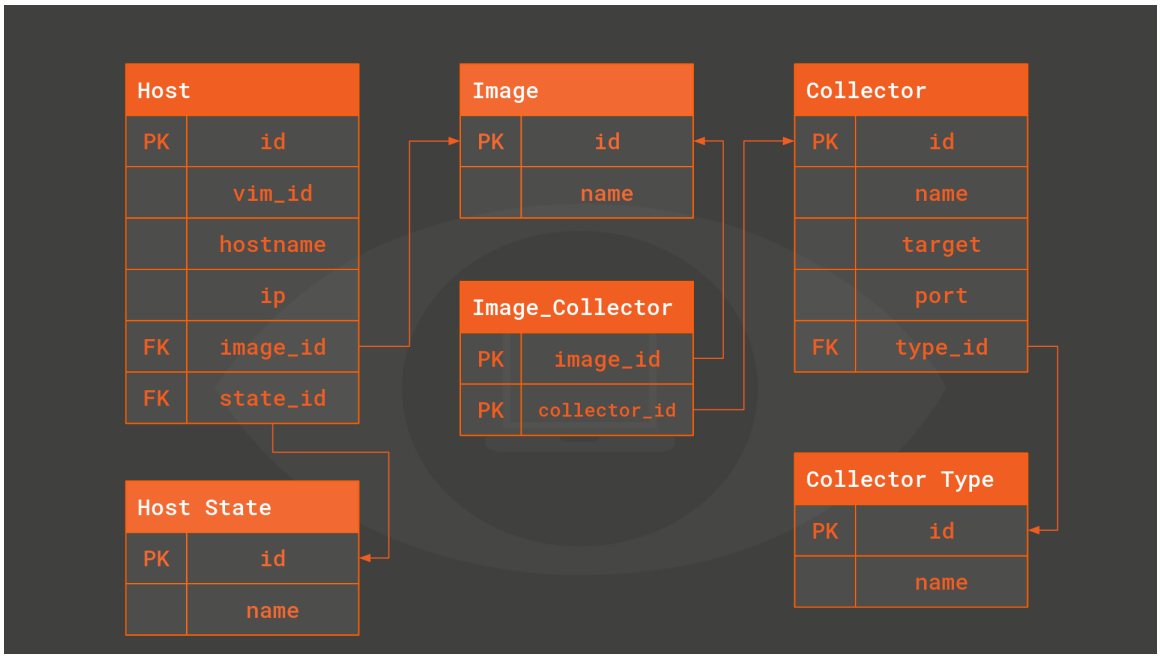

Figure 4.2: Relational Database Schema

For the hosts, which is the table containing information on the system's monitoring targets, the ip adresses and hostnames are stored, along with a field to keep track of the machine's state (active or not). Orwell's service discovery provides a list with the active ones, inactive hosts cannot be monitored.

Collectors have a target and port fields, which are used for pull collectors, so our system knows where to fetch the metrics from. For the context of this project no pull collectors were used, but the system could easily integrate this option.

As for images, only a name is assigned to better identify the image's contents.

With the assossiations schemed in the figure 4.2, it is assured that given an host, the system can identify which image the machine is running, and which collectors are available for metric harvesting.

Orwell integrates various types of metrics, therefore, different approaches will be required for each type of metric that is intended to be obtained. For metrics of a VNF, these will be obtained through OS images, and it was for these metrics that the initial phase of the Orwell project was directed. After being able to collect metrics using the described method, a backup plan had to be considered in case a OS image did not have any metrics exporter. The solution for this problem is based on collecting metrics at the hypervisor level which contains a Gnocchi service collecting metrics from every machine. After this collection was completed, we went on to collecting network metrics using the Perfsonar, and lastly, the metrics of the 5G network, which were collected via Huawei's eSight API.

4.3.1 How metrics are collected

OS Images

As stated in the purpose of our study, VNFs are designed to be seen as blackboxes and so should not be accessed to collect metrics from them.

One solution is to configure OS images that will be associated to the VNFs and, because we are working at the operating system level, we can acquire metrics from a VNF without having to know the machine's credentials to access it.

These linked OS images are actually Ubuntu 20.04, to which we just added the installation of a metrics gathering tool, which was set up as a service of that OS image and will always be running while that VNF is active.

Two OS images were then created and in the next subsections we will explain in detail how they are developed.

Prometheus' Node exporter

To collect metrics with Prometheus' node exporter, an OS image was created with the tool installed. When running, we can query the node exporter's API via HTTP to fetch the machine's metrics.

With a simple Python script, the /metrics resource of the API is consumed in intervals of 15 seconds. The metrics returned are then pushed to the Kafka broker, to be later handled by the other Orwell components.

To automate this worfklow and give it a one touch deployment behaviour, the image is provided with a file containing the address of the core's Kafka Service Discovery endpoint, which will be used to point our Python script to the Kafka bootstrap server. Finally, a service was created to setup the node exporter and run the Python script. This service runs when the image boots, allowing Orwell to begin monitoring the VNF with no necessary manual configurations.

Telegraf

Initially, Telegraf was installed on an OS image that only contained Ubuntu.

After installing Telegraf, we proceeded to the configuration section of the Telegraf.conf file, where we defined all of the metrics that we wanted to collect in the inputs, and then the output, which in this case will be sending to our Kafka the metrics that Telegraf will be able to obtain, for which we injected the location of Kafka in the config file that was passed as an environment variable.

Then, because we already had the metrics to send to Kafka, we needed to configure the Telegraf service in the image OS's systemctl, which was as simple as creating a .sh file that launched Telegraf and calling the .sh file from the Telegraf service established.

Because the service was already configured and ready to use, it was just necessary to activate it and set it to enable so that once the OS image rebooted, it was instantly performed in the boot options. As a result, we produced an OS image with Telegraf that allows us to retrieve metrics from any VNF without having to enter it, which was our goal.

Gnocchi and OpenStack

Gnocchi is a time-series database which was created to store time series data with a very flexible data model, which is used by OpenStack as its default database, where it stores Virtual Machines' metrics.

In ITAv's OpenStack, Gnocchi always has data related to every machine (VMs and VNFs) running on the stack, but the metrics available are limited in quantity and quality, since the time interval reaches some hours between metrics sometimes. Nevertheless, when the OS image is unknown or does not have any collector associated with it, we use Gnocchi to be able to present some metrics related to every VNF.

PerfSonar toolkit and testpoints

Metrics from PerfSonar tests can be stored in many different places, being the most common one the archival service that comes with PerfSonar, esmond. Our collector interacts with the API from that service so as to gather the latest metrics across all the tests running and publish them in our Kafka.

eSight

5G metrics are collected through an API developed by Huawei called eSight. This API provides information about the 5G network interfaces and its devices. However, since the necessary infrastructure is still not in place, some of the methods are not available.

To collect metrics from this API we need to gather a list of all the interfaces and devices we want to monitor. Then, we need to create tasks for each one of them. These tasks only perform some measurements, so, if, for instance, we want to know the receiving and the sending rate of all the interfaces we have to create two different tasks for each interface. After creating the tasks we have to wait until the system is ready to export metrics which takes up to 20 minutes. After getting the metrics, these are sent to Kafka, so the respective translator can deal with them.

We developed an exporter that runs as a cron job every day. As the gathering of these metrics may have an impact on NetApp test results we were not allowed to collect metrics continuously. However, the code is ready for this scenario and all it takes is the change of a line of code which is flagged on the code.

eSight Constraints

Considering that eSight is not yet totally functional we had a lot of challenges through the development of the exporter for this API.

- time between tests - we had to wait 20 minutes for the tasks setup every time we wanted to test our exporter which made the development process much more time consuming
- devices' metrics are not available through the API - devices' metrics are only available through the user interface which did not fit the purpose of this project. To bypass this restriction we collect metrics from the components of each device. Nonetheless, there are only seven tasks available for the components which limited the results
- 5G RAN infrastructure has no monitoring - 5G RAN enables metrics collection for each antenna individually. This is undoubtedly important to understand the system's behavior and the 5G infrastructure status. However, since these were not available we could only gather information related to 5G Core

4.4 Message broker

In this work, metric collectors are the broker's producers, sending metrics regularly from various sources, ranging from VNF to network metrics, which must be distinguished in the message broker, and that is where the topics come into play. The consumers are the translators, which are usually associated with a unique Kafka topic, a unique collector service.

4.5 Python Package

Our work was focused on a well-defined subset of tools that met our requirements but, because of the modularity of our solution, any message that can be exported to our message broker and afterwards translated to Prometheus format can be integrated into our system. Collecting and publishing metrics to Kafka is a process too much dependant of the interfaces that a specific tool provides for us to facilitate, but the translation of a message to Prometheus format, which could be considered a very error-prone step and lead to hours of debugging, could be made more transparent. For that, we created a Python module and published it to PyPI.

With the help of this module, a developer has the only responsibility of creating a single function that receives a message (according to what was sent to Kafka) and returns a list of Metric objects. Under the hood, the module deals with the Kafka connection, transforms the Metric objects into valid Prometheus format and publishes the result to Redis, making it available for the middleware to use.

Example of a very simple translator:

```
from orwell import Metric, Runner


# The translator function should be able to receive multiple lines
# of the desired service output and return a list of Metric objects

def translate(lines: str) -> list[Metric]:
  metrics = [ line.split(',') for line in lines.split('\n') ]
  return [ Metric(metric, value, { 'host': host }) for host, metric, value in metrics ]


translator = Runner(translate)
translator.run()
```

As previously said, translators are those who will consume messages from the message broker and whose primary function is to translate metrics.

As required for this project, all metrics must be in a single format, which is the Prometheus format, thus every metric that reaches a translator must be transformed so that we can centralize metrics' storage.

The translators are Python Kafka consumers able to translate metrics from one specific format to Prometheus format, which is easier when using the developed Python package. When the message broker receives new metrics, the translators translate them and forward them to the next component, which in this case is the system's Cache.

Since the Grafana dashboard will be the same for all VNFs' metrics provided by the different collectors, the translators for Node exporter, Telegraf, and Gnocchi are responsible for generating metrics in a uniform format. This is a very complex task, since different collectors generate different metrics in different ways which will have to be manipulated to meet this uniform format.

4.6.1 Prometheus' Node Exporter

This translator will be responsible for reading messages from the *prometheus* topic of Kafka, and because the node exporter is already in the prometheus format, the code required to conduct the translation will be minimal.

4.6.2 Telegraf

This translator will be responsible for reading messages from the *telegraf* topic of Kafka. As previously stated, in addition to translating the metrics into Prometheus format, it was also required going through all of the metrics to see if we already had a direct translation for the compatible name or if we had to create a new one.

4.6.3 Gnocchi

This translator will be responsible for reading messages from the *gnocchi* topic of Kafka. As the previous translator, metrics had to be manipulated to follow the format defined previously.

4.6.4 PerfSonar

This translator will be responsible for reading messages from the *perf* topic of Kafka. Since PerfSonar has its own Grafana dashboard, the translation process did not require any large transformation.

4.6.5 eSight

Interfaces

This translator will be responsible for reading messages from the *esight_interface* topic of Kafka. Since eSight interfaces has its own Grafana dashboard, the translation process did not require any name transformation.

Slots

This translator will be responsible for reading messages from the *esight_slot* topic of Kafka. Since eSight slots has its own Grafana dashboard, the translation process did not require any name transformation.

4.7 Orwell API

Orwell's middleware REST API offers a range of resources to both orchestrate and manipulate the system's modules. It can be broken down in categories providing different functionalities for the entities they interact with. In this section we present the structure of the API and the previously mentioned categories.

4.7.1 Structure

The API is structured in a set of folders, each containing code for different layers of the application, providing uncoupled micro Python modules. The current version's file tree is presented below:

```
.
|--- app
|      |--- cache
|      |      |--- __init__.py
|      |      |--- redis.py
|      |--- config
|      |      |--- cors.py
|      |      |--- docs.py
|      |      |--- __init__.py
|      |      |--- settings.py
|      |--- data
|      |      |--- db
|      |      |      |--- conn.py
|      |      |      |--- __init__.py
|      |      |--- __init__.py
|      |      |--- models
|      |      |      |--- associations.py
|      |      |      |--- collector_image.py
|      |      |      |--- collector.py
|      |      |      |--- collector_type.py
|      |      |      |--- host.py
|      |      |      |--- host_state.py
|      |      |      |--- image.py
|      |      |      |--- __init__.py
|      |      |--- postgres.py
|      |      |--- schemas
|      |      |          |--- associations.py
|      |      |          |--- collector.py
|      |      |          |--- collector_type.py
|      |      |          |--- host.py
|      |      |          |--- image.py
|      |      |          |--- __init__.py
|      |      |          |--- response.py
|      |--- gnocchi
|      |      |--- gnocchi_exceptions.py
|      |      |--- gnocchi_puller.py
|      |--- __init__.py
|      |--- main.py
|      |--- osm
|      |      |--- osm_adapter.py
|      |      |--- osm_exceptions.py
|      |--- routers
```

```
|           |--- collectors.py
|           |--- db_helper.py
|           |--- hosts.py
|           |--- images.py
|           |--- __init__.py
|           |--- metrics.py
|           |--- service_discovery.py
|--- Dockerfile
|--- requirements.txt
```

4.7.2 Discovering Services

/service_discovery/targets

To find the targets Orwell should be monitoring, the list of currently active hosts stored in the database (persisted from the parallel OSM's API calls) can be accessed in this resource. Mainly consumed by Prometheus to fetch its targets. The response given by this endpoint follows the Prometheus standard.

/service_discovery/kafka

The endpoint returns the address of Kafka's bootstrap server. This is useful to dynamically configure both our OS Images, pointing the collectors where to publish the metrics and our Translators, to consume and translate the messages.

4.7.3 Metric fetching

/metrics/id

Prometheus consumes directly from these endpoint to store in its database. The API fetches the already translated metrics from the Redis cache, populated by the Translators.

4.7.4 Management

/images/*

/collectors/*

/hosts/*

All CRUD functionalities are provided to images and collectors, allowing the manipulation of both entities. This is mainly consumed by a Web App where we can associate or modify collectors in the OS Images or list which collectors each Host is using.

4.7.5 Background tasks

In addition to providing these resources, Orwell's API is also responsible for both OSM service discovery and metric pulling from the Gnocchi collector. These tasks are implemented on parallel threads (launched on the API initialization) which periodically consume from external services.

4.8 Service Discovery

Orwell has two different service discovery services, one configured in Orwell's core and another in Prometheus. This section will describe both of them and their role on this solution.

4.8.1 Orwell SD

Orwell Service Discovery is a custom made service discovery mechanism that queries OSM targets every 30 seconds. OSM returns information about every active VNF such as the OS image running on the VNF. This information is stored in our hosts config database and is used to inform the Gnocchi Puller about the VNFs without a monitoring tool installed on the OS Image. The list of active VNFs returned by OSM is compared to the active VNFs in the database and the ones that do not belong to the OSM's list are marked as inactive. The list of active VNFs is available through Orwell API at /service_discovery/targets.

4.8.2 Prometheus SD

Prometheus allows the configuration of a service discovery service instead of hard-coding the addresses of all the collectors in the configuration file. There are many built-in integrations such as getting addresses from OpenStack, but for our use-case we required a more personalized option, integrating it with our own HTTP endpoint, which has the only requirement of following a well-defined structure. After starting, Prometheus will start reading the configured endpoint every minute, by default, and collecting the metrics in the specified addresses. In this case, all the addresses point to different endpoints of our middleware.

4.9 Orwell Management Platform

For components like collectors, images and hosts to be managed in a more accessible and efficient way, Orwell provides an Angular Web App.

The main functionalities consist in creating, updating and deleting components and their respective relations, providing a dynamic interface to execute operations such as defining collectors for an image or change an host's current image. The app consumes from the middleware's REST API both to fetch and persist data.

Although used mainly as a development utility, new features can easily be added to provide more functionalities, like information on active translators and tools to deploy new ones, monitor and orchestrate active components and statistics about the system.

4.10.1 Suricata

Suricata must be configured in a VM that should be the network entrypoint, all traffic has to travel through it both to enter and depart the network, so that packets can be evaluated. For our implementation we did not have authorization to make such a configuration inside of the ITAv's network and therefore we were not able to configure a fully working Suricata instance, but only monitoring the packages which were destined for it.

Said instance was configured in IPS mode to be able to block suspicious packets. When a packet arrives it is verified to which configured rules this packet belongs, and depending on that categorization there are four possible behaviours which are alert, drop, pass and reject. A packet can, technically, belong to more than one rule, in which case the drop operation has priority.

As seen in the listing 4.1, a rule always has a sid, which is the rule's id (highlighted in the figure). The first word in the rule identifies the type of rule (an alert), followed by the protocol it will operate on (ip) and the assessment location: source and destination (any for both, in this case). When a rule is violated, an entry in the log file will display the message set in the msg variable and an alarm will be triggered anytime a privilege check command is given. The violation of a rule implies that someone with root rights within the computer is utilizing these privileges in a wrong way and may be trying to exploit further vulnerabilities. The alarm is registered in the log file, as shown in the listing 4.2. To show the alert related to this event a grep by sid was performed just to make the screenshot easier.

Listing 4.1: Some Java code

```
alert ip any any -> any any (msg:"GPL␣ATTACK_RESPONSE␣id␣check␣returned␣root";
content:"uid=0|28|root|29|"; classtype:bad-unknown; sid:2100498; rev:7;
metadata:created_at 2010_09_23, updated_at 2010_09_23;)
```

Listing 4.2: Suricata example of Logs

```
grep 2100498 /var/log/suricata/fast.log
05/25/2022-12:47:22.660800  [**] [1:2100498:7] GPL ATTACK_RESPONSE id check
returned root [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP}
18.66.122.20:80 -> 10.0.12.84:38998
05/25/2022-13:43:41.924165  [**] [1:2100498:7] GPL ATTACK_RESPONSE id check
returned root [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP}
18.161.97.95:80 -> 10.0.12.84:45062
05/25/2022-18:09:03.890624  [**] [1:2100498:7] GPL ATTACK_RESPONSE id check
returned root [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP}
18.161.97.120:80 -> 10.0.12.84:33444
05/25/2022-18:22:46.154957  [**] [1:2100498:7] GPL ATTACK_RESPONSE id check
returned root [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP}
18.161.97.57:80 -> 10.0.12.84:53978
05/25/2022-18:23:57.773508  [**] [1:2100498:7] GPL ATTACK_RESPONSE id check
returned root [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP}
18.161.97.21:80 -> 10.0.12.84:42742
```

Because the same packet in one network might not be critical and in another might be crucial, the rules that were specified in Orwell are all alert only, and it is up to whomever uses the solution to update the rules file for their use case.

In our system, all suricata logs are passed to elasticsearch, which then sends them to Kibana. Kibana's dashboards can be accessed by searching for "type:dashboard suricata" in the search bar. There will be found two Suricata dashboards:

- Alert dashboard

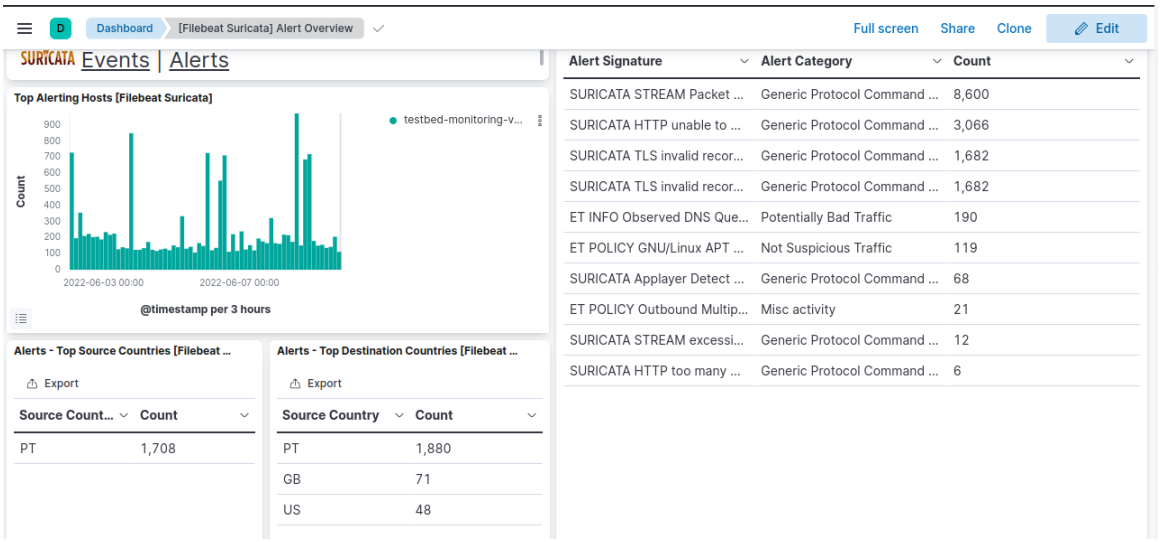

Figure 4.3: Suricata alert dashboard example

As shown in the figure 4.3, on this dashboard we can see how many times alerts were sent for each of the rules, the time of they day they occurred, and which countries were the source and destination of the packets that triggered those alerts.

- Event dashboard

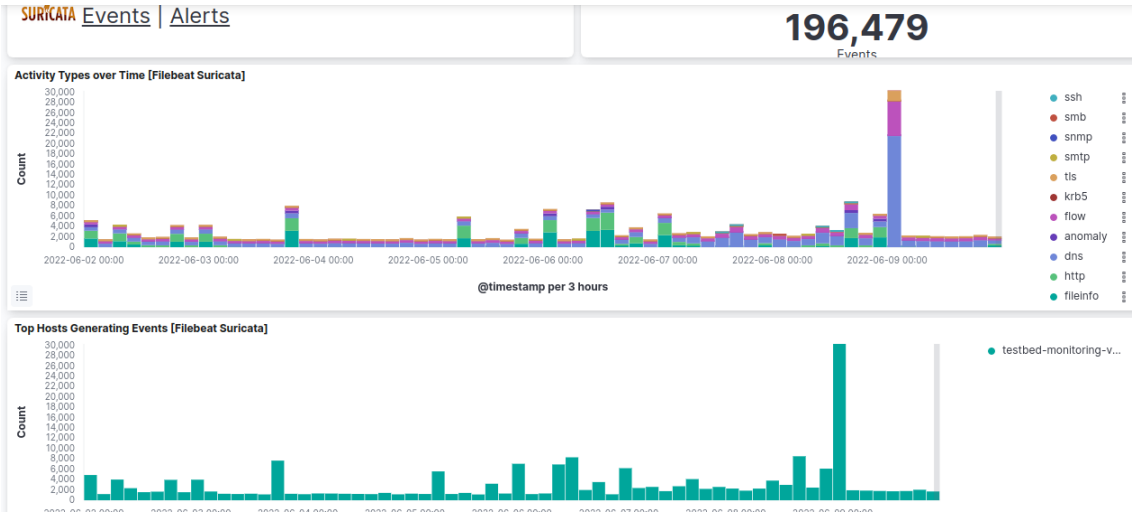

Figure 4.4: Suricata event dashboard example 1

As shown in the figures 4.4,4.5, this dashboard displays the events that occurred, with further insights on the associated activities and the hosts that created these events. It is also possible to see when each event occurred as well as which ip addresses, ports and transport layers were used.

The configured rules were associated with different priority values, which can range from 1 to 4. So that alerts could be better monitored we created a Slack workspace with four channels, one for each priority and configured alarmistic via webwooks.

The logs are then separated into the slack channels, 4.6, and logs will be transmitted to slack all day long, allowing the network manager to have a cleaner and more structured manner of monitoring everything that happens on the network.

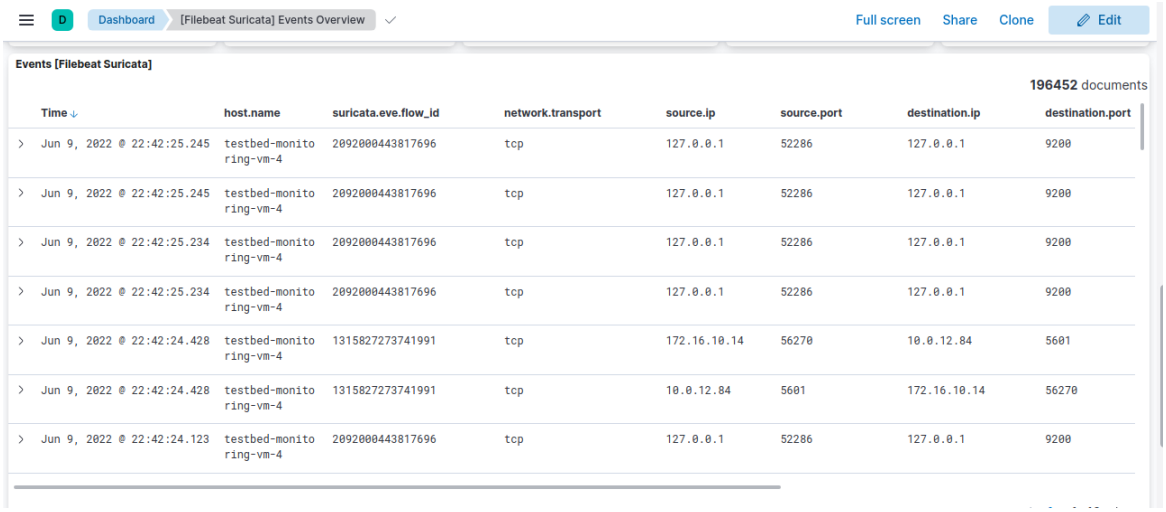

Figure 4.5: Suricata event dashboard example 2

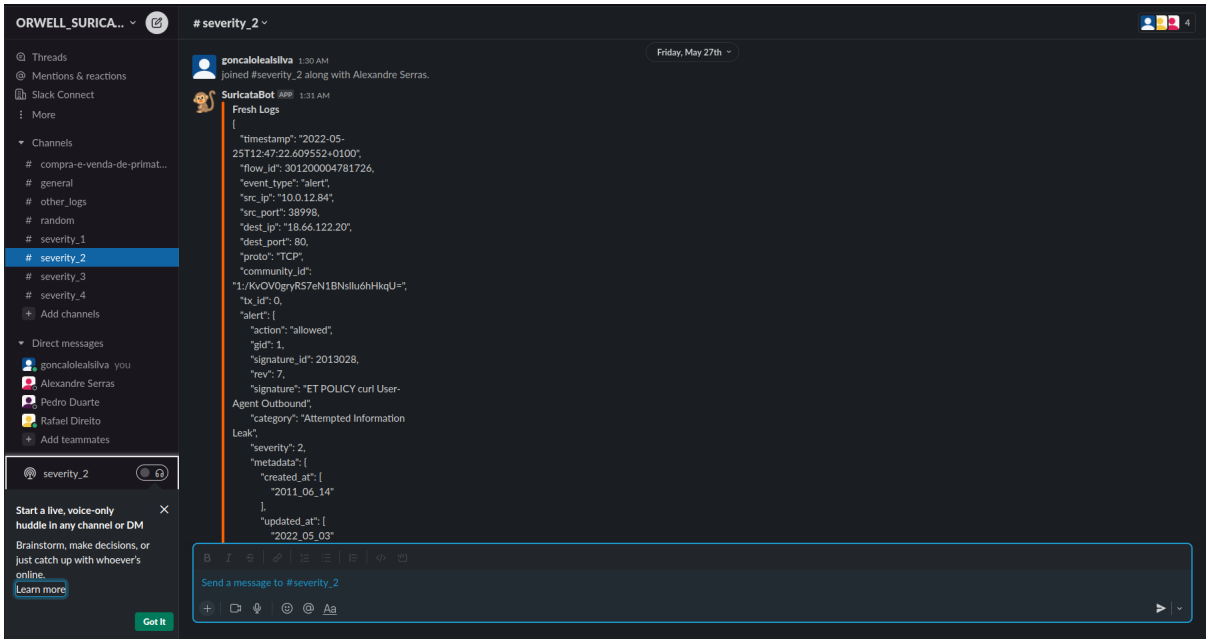

Figure 4.6: Suricata Slack channel

4.10.2 Infection Monkey

Infection Monkey is an open-source network breaching simulation tool which can be used to test and find flaws and exploits on defined targets.

Infection Monkey in Orwell

In Orwell's context, Infenction Monkey is able to detect vulnerabilities in services running on the VNFs or exploitable configurations, situations that compromise not only the testbed but the whole network. For testing purposes, an Infection Monkey server was setup on a closed subnet of the network with vulnerable nodes. The tool successfully identified and cracked the target nodes, proving it could be easily deployed on the testbed to generate reports for the administrator and/or netapp developers.

This service will be mainly used to give NetApp developers a report on the stability of his application.

REST API

Since no API is provided for this service, to make Infection Monkey's usage an automated process, an

API was developed using the FastAPI framework to interact with the web app via Selenium. In addition to providing easier access to the functionalities of the tool, since our API was conceived as a standalone project, it can be quickly integrated in different contexts with a package release on PyPI.

The API allows running simulations imported from config files, abort all current tests and list which configurations are available. In its current state, there is no authentication to access the instance, which should be a concern when used on untrusted networks.

As an open source project, it can easily be contributed to provide features to match many other use cases. Thanks to the modularity of the development, an open source contributor could easily just create a new method on our Selenium setup abstracted main class to meet their use case.

4.11 Data Storage and Visualization

This layer of our application is composed by Prometheus (storage) and Grafana (visualization).

4.11.1 Prometheus

Prometheus is initialized with a custom configuration file that enables service discovery with our middleware's address.

4.11.2 Grafana

Grafana does not start with any customization as the idea is that the user would never need to customize it as all the configuration is automated when required by our middleware. Taking advantage of its extensive API we are able to create accounts and dashboards, as well as dealing with authorization, abstracting all the customization features from the end-user, making it deeply connected with our system.

So as to facilitate metrics' monitorization, we've also configured Slack notifications with a few demonstrative examples, which are visible in Figure 4.7

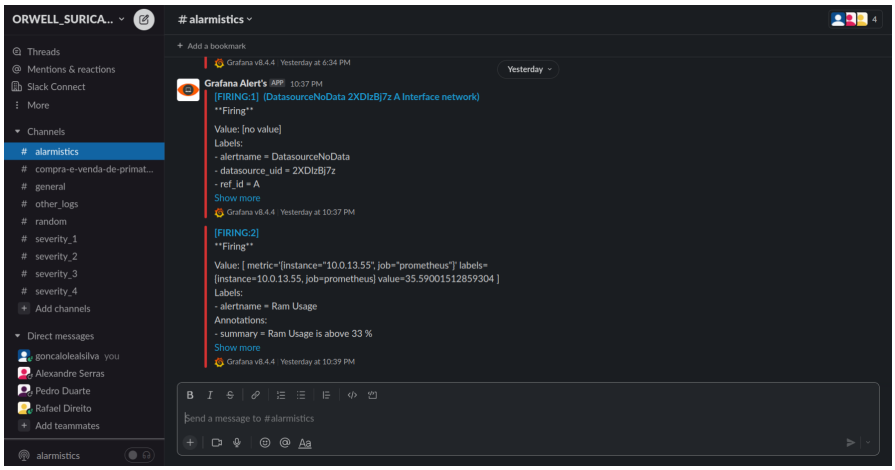

Figure 4.7: Grafana Slack channel

CHAPTER 5

Results and Discussion

5.1 VNF Metrics Exporters

Metrics from running VNFs were successfully collected and translated using the three different collectors - Gnocchi, Telegraf and Node Exporter. In fact, for a developer, it is completely transparent which technology is being used to monitor his environment as the metrics from the three tools appear in the exact same dashboard.

5.1.1 Gnocchi

Starting with our worst result, Gnocchi is the exporter with the fewer metrics available. It is still a very important part of our solution because that data comes directly from the hypervisor, allowing us to gather the minimum metrics whether the VNF's image has an exporter running or not.

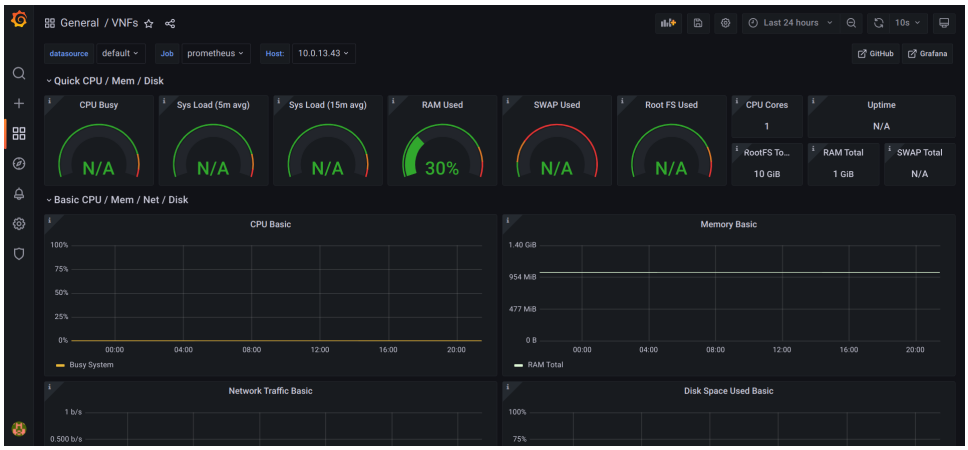

Figure 5.1: Gnocchi Dashboard

5.1.2 Prometheus Node Exporter

The dashboard used was adapted from a template for this exporter and, therefore, it is not a surprise that it behaved perfectly. We are able to flawlessly fill the whole dashboard with updated metrics across all the Grafana panels.

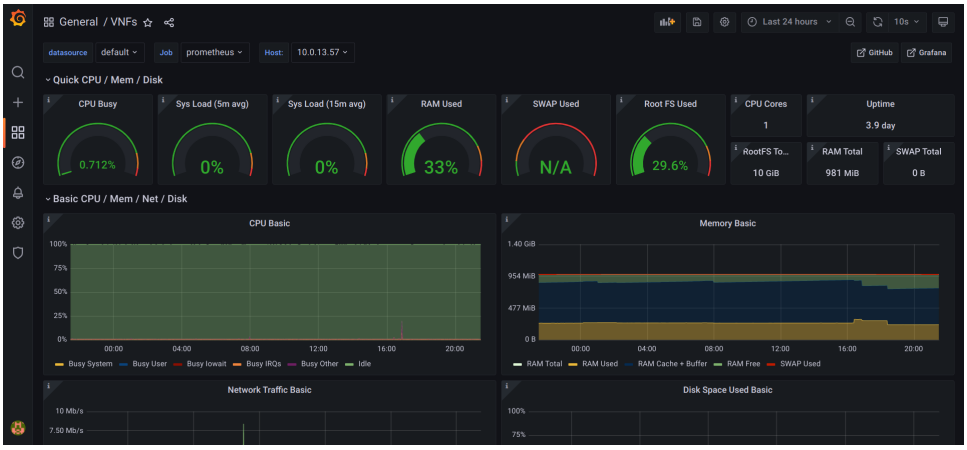

Figure 5.2: Node Exporter Dashboard

5.1.3 Telegraf

This exporter was probably the most challenging as, with the proper configuration, it allows us to collect all the required metrics and, therefore, we had an enormous amount of translations to make. We also succeeded in this case, presenting a complete dashboard with all the metrics that Node Exporter was also able to provide.

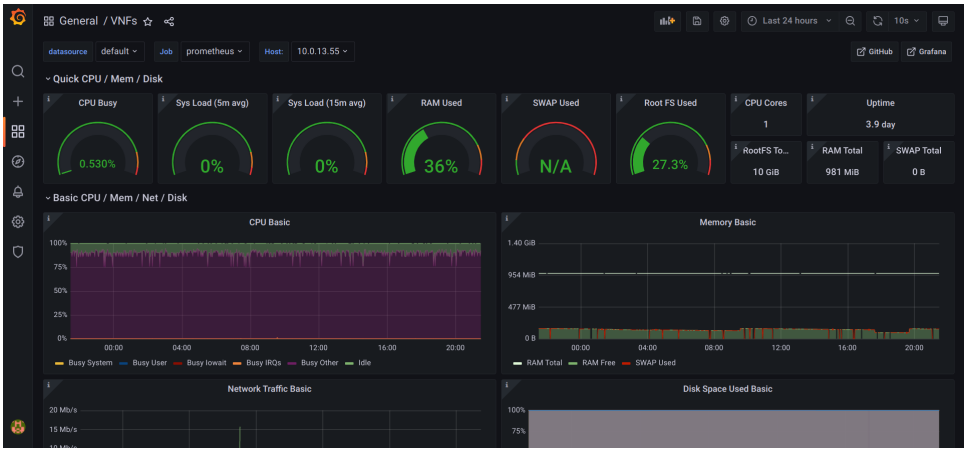

Figure 5.3: Telegraf Dashboard

5.2 Network Metrics

PerfSonar was integrated into our system in multiple layers. Starting from the middleware API, an endpoint was implement so as to automatically set up tests with a specified node, as well as configuring Grafana's dashboard so as to add given addresses to the available variables. It is then possible to consult the metrics resultant from those configured tests in a Grafana dashboard, thanks to an external service that reads PerfSonar's API every ten minutes and publishes them to Kafka, to a topic where the corespondent translator will be listening.

We also provided easy initialization scripts, one for Toolkit and other for Testpoint so as to achieve the expected one-touch deployment. Both start a docker container and run the required configuration commands and it is even provided a template for the NTP configuration, which is required to run the tests.

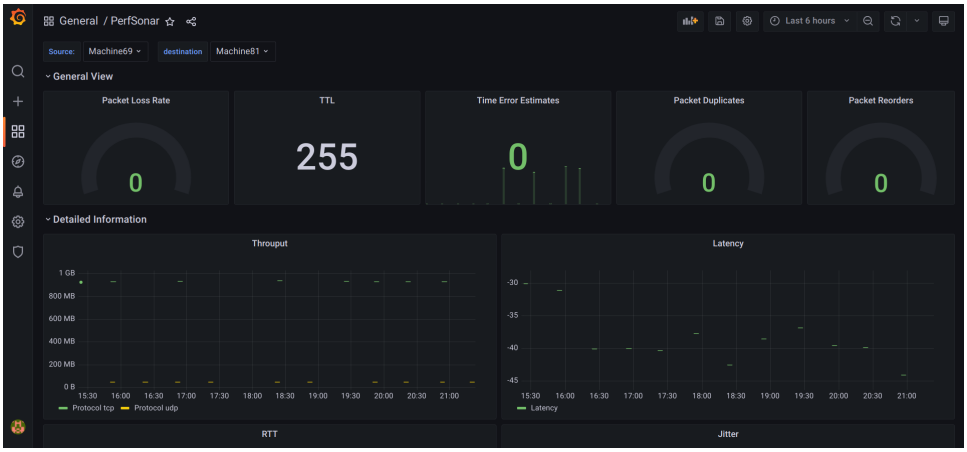

Figure 5.4: PerfSonar Dashboard

5.3 5G Core Metrics

The collection of metrics for the 5G core was a slow process, mostly associated with the quality of the data source, as already explained. Another imposed limitation was the time that we were allowed to gather data. The required API is part of the whole 5G infrastructure and there was a preoccupation with the possible interference of frequent requests to said API in the running tests. Therefore, we are only collecting two hours of metrics daily, from midnight to 2 am. Despite all that, we were able to provide relevant metrics for both 5G interfaces and slots through two separate Grafana dashboards.

5.3.1 Interfaces

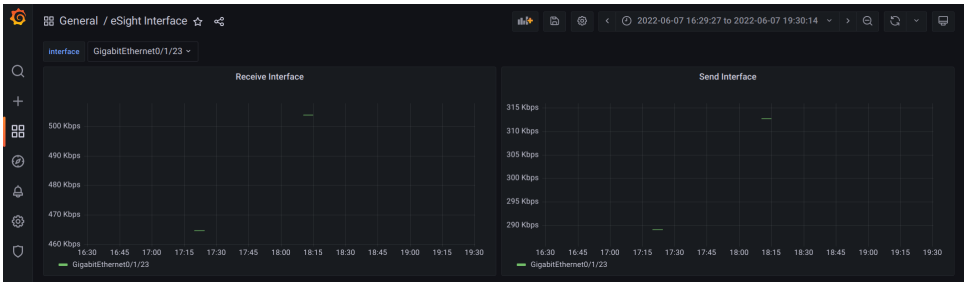

Figure 5.5: eSight Interface Dashboard

5.3.2 Slots

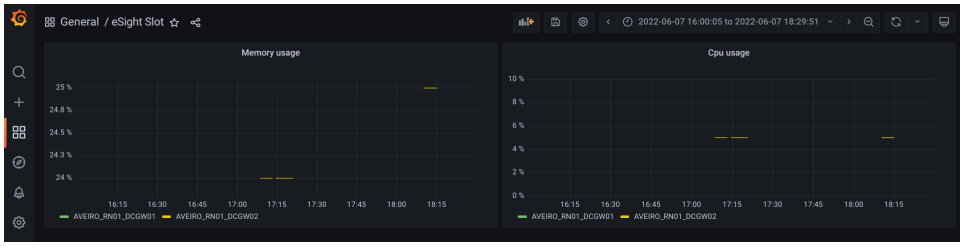

Figure 5.6: eSight Slot Dashboard

Security implementations in our work, being a sensible topic, were both limited by what we were allowed to do from the ITAv administration. In both cases we were able to achieve solutions that only lacked proper network configuration to work as expected.

5.4.1 Suricata

We were able to achieve a fully-configured Suricata instance, with alarmistic via Slack webhooks. As also mentioned before, we were not allowed to install this solution in an actual network entrypoint, leaving us with the notification of the packets addressed to our VM.

5.4.2 Infection Monkey

Infection monkey was also left ready for deployment, with an extra Selenium API, given the lack of a proper built-in API. The created interface allows starting tests with a single request as seen in Figure 5.7, which is an important step for automating the interaction with this tool. Our limitation in this case was that we were not allowed to run penetration tests around the whole network and therefore were limited to a different virtual network, where we only had a sample of machines. The results of this simulation (network 10.0.11.0/24) are presented in Figures 5.8 and 5.9

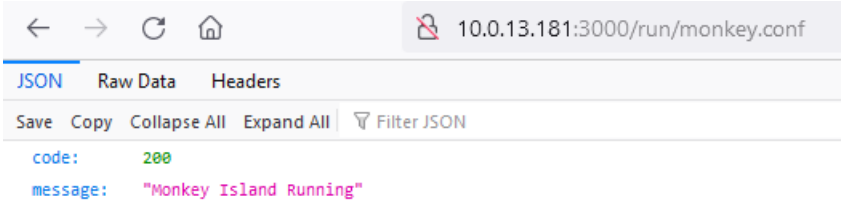

Figure 5.7: Running the monkey from the API

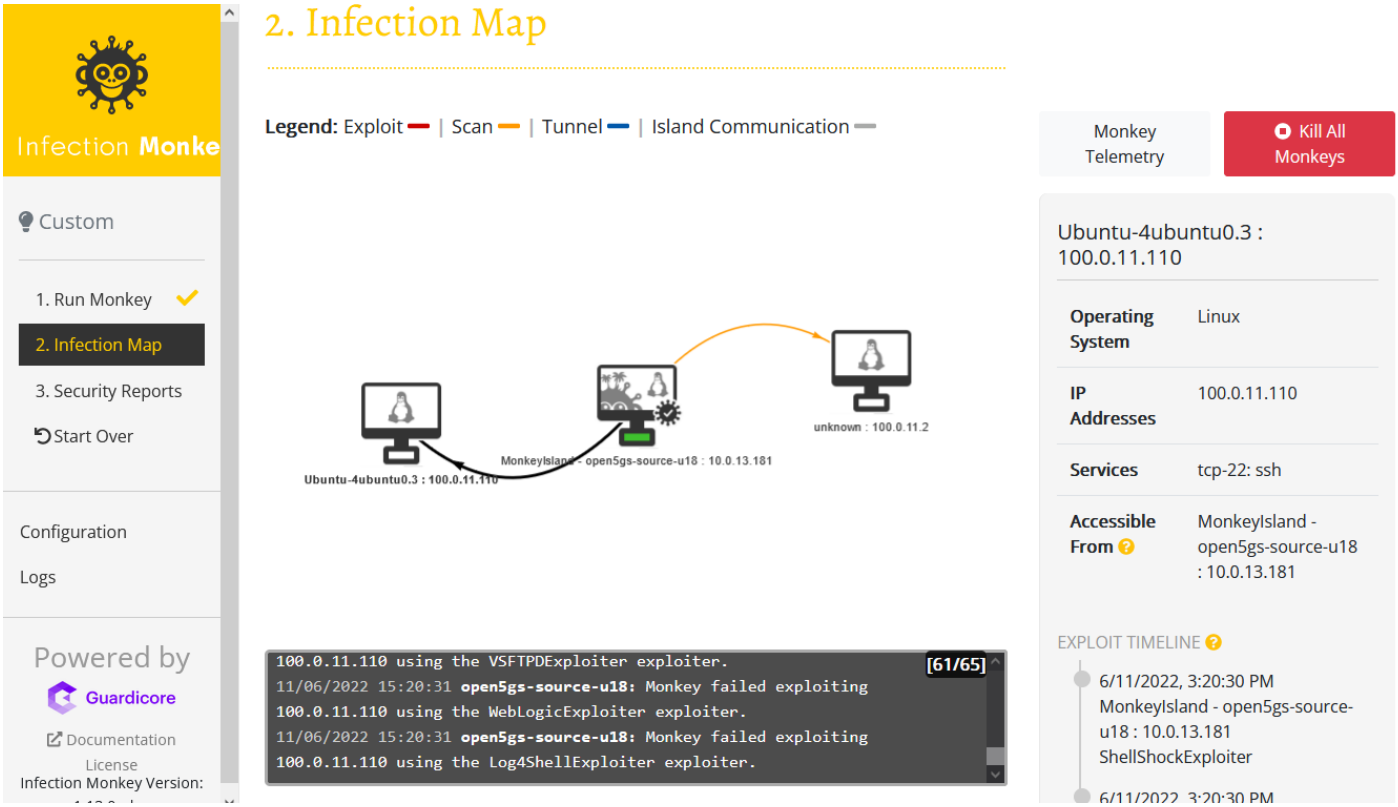

Figure 5.8: Infection Monkey detecting machines

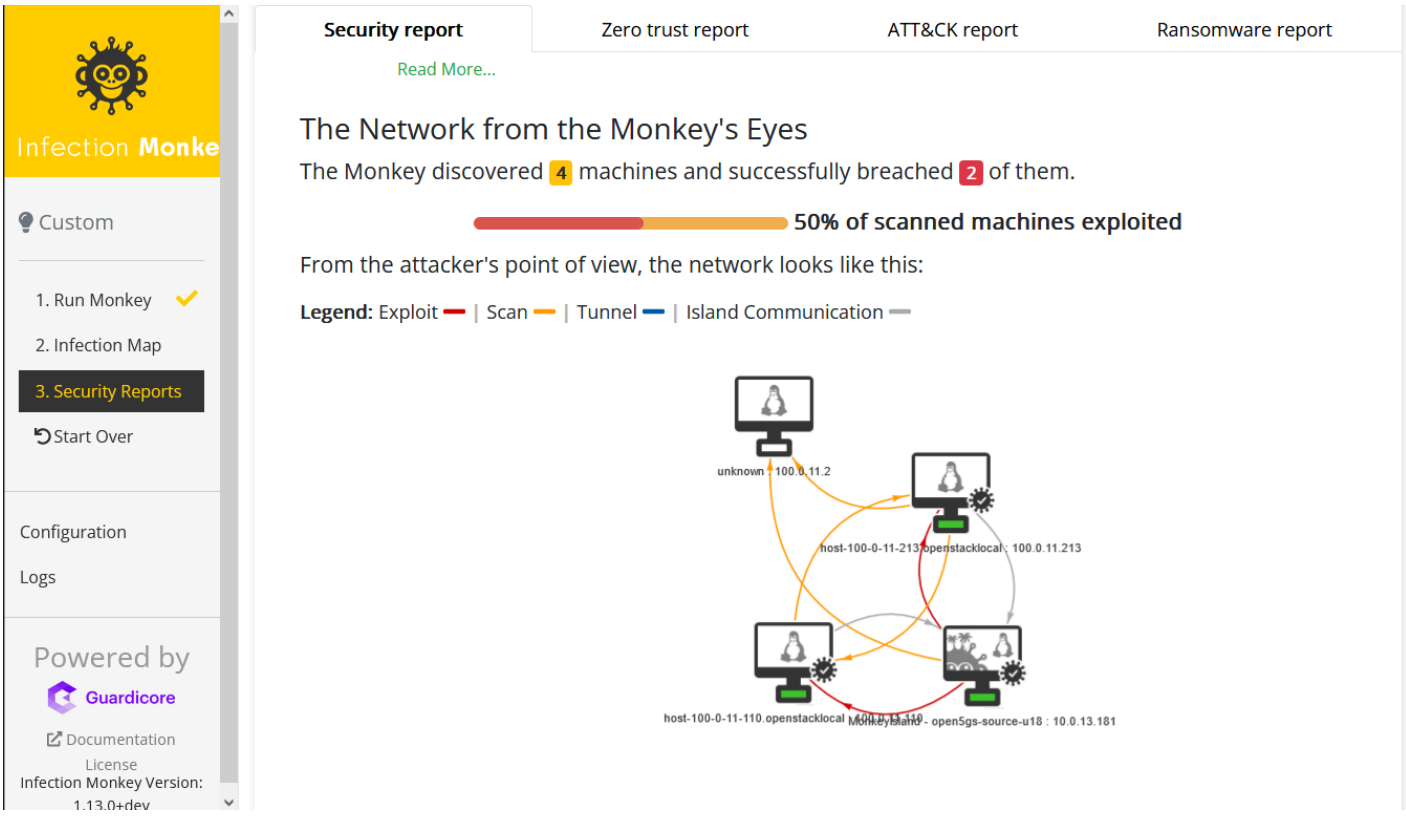

Figure 5.9: Breaching Report

CHAPTER 6

Orwell Monitoring System

6.1 Backlog

Our backlog was built over the ClickUp[1] platform, where we are able to define the tasks on weekly sprints, assign that task to one of the developers, estimate the effort of given task in sprint points and measure the time that it took to complete given task.

Taking advantage of their API we were also able to automate updates of the backlog in our website.

6.2 Pull Requests and Code Review

For source control, GitHub[2] repositories were setup, containing the code for the wide range of Orwell's components. These can be accessed on the organization's page.

Our pull request policy required one code review by someone who did not develop any of the commited additions. Since various independent components were being built simultaneous, this policy allowed the team to always make sure everyone was on the same page, preemptively detecting and fixing situations which could raise issues or incompatibilities in the future.

6.3 Git Workflow

In the terms of workflows and branching strategy, a similar approach was taken in the whole development process. Each repository has its main branch, where the production's version's source code resides. To work on new functionalities, feature oriented branching was used, where branches would follow the naming **feature/<feature_name>**. After reviewing, the feature is integrated in the production environment. To handle bugs and fixes to already implemented features, a branch was created with the **fix** prefix, with the format **fix/<feature_name>**.

Branching allowed the team to always work with a production ready solution without congestioning parallel component developments.

6.4 Repositories

Organization: https://github.com/OrwellMonitoring

[1] https://clickup.com/
[2] https://github.com/

core middleware	
translators	https://github.com/OrwellMonitoring/orwell-translators
translator python package	https://github.com/OrwellMonitoring/orwell-python-package
storage visualization	https://github.com/OrwellMonitoring/orwell-storage-visualization
manager WebApp	https://github.com/OrwellMonitoring/orwell-manager
boot services	https://github.com/OrwellMonitoring/orwell-boot-services
infection monkey REST API	https://github.com/OrwellMonitoring/infection-monkey-api
suricata configs	https://github.com/OrwellMonitoring/suricata_config
perfsonar utils	https://github.com/OrwellMonitoring/perfsonar-utils
eSight connector	https://github.com/OrwellMonitoring/esight_connector
documentation	https://github.com/OrwellMonitoring/orwell-documentation

6.5 Deployment

Orwell's proof of concept deployment relies on containerizing services using Docker and launching them as modules aggregated by Docker Compose which will run in any host in the network. This allows a very flexible, configurable environment offering a great degree of scalibility and simple component migration. The modules provided in the final solution are:

- Middleware (core API, PostgreSQL, Redis and Kafka)
- Storage Visualization (Prometheus and Grafana)
- Translator Pack (one container of each developed translator)
- Management Plataform (Angular Web App)

In addition to these modules, other external services were deployed throughout the network like for example the aforementioned Infection Monkey and Suricata instances.

The final deployment follows the diagram on Figure 6.1

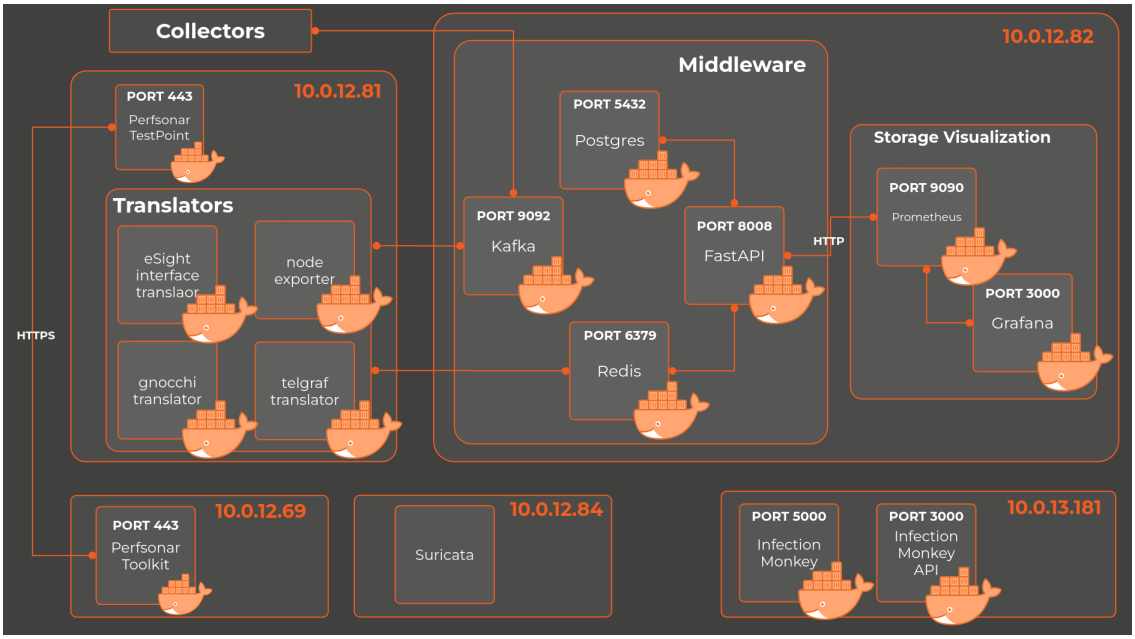

Figure 6.1: Proof of Concept Deployment

6.5.1 Deployment tutorial

Thanks to Orwell's modularity, besides a few required pieces, most of the components are optional, allowing users to pick what matches their use cases.

The environments of the modules are documented on the documentation website (https://orwellmonitoring.github.io/documentation/)

Middleware

Orwell's deployment entrypoint should always be the Middleware. As mentioned before, this component aggregates tools to integrate all other modules, such as the REST API and the message broker which are containerized in a docker-compose file.

Requirements:

- docker
- docker-compose

Run command:

```
$ docker-compose up --build
```

Storage Visualization

To store and view the collected metrics, we provide a compose with a Prometheus and Grafana instance, as well as a prometheus.yml configuration file where the url to the middleware's service discovery should be updated.

Requirements:

- docker
- docker-compose

Run command:

```
$ docker-compose up --build
```

Translators

Once the system can collect and store data, depending on which exporters the targets are running, translators should be setup to make the bridge between the exporters and the cache where Prometheus will fetch its metrics. Again, we provide a compose with a pack of translators implementing the translation of the services on ITAv's testbed.

Requirements:

- docker
- docker-compose

Run command:

```
$ docker-compose up --build
```

As mentioned before, new translators are easily created and containerized. The Dockerfile provided in the repository can be used as a template to dockerize these components.

Infection Monkey + API

Infection Monkey is fully optional. After setting up an IM server (Guadicore official tutorial), simulations can be immediately ran based on configurations by interacting with the Web App.

Requirements:

- MongoDB database

Starting the Monkey instance depends from the installation method used. Since we opted for the docker version, after extracting the tar file, the run command is:

```
$ docker run --tty --interactive --name monkey-island \
    --network=host guardicore/monkey-island:v1.13.0
```

The API can also be deployed to make this service's usage easier. In this compose, two services are created: the API to accept requests and a Chrome driver container to use with selenium. Requirements:

- docker
- docker-compose
- Infection Monkey server

Run command:

```
$ docker-compose up --build
```

Suricata

Since Suricata's setup requires the installation and configuration of other external services, the extent of this process would miss the scope of this section. For more information on this topic, a full detailed guide can be found at our documentation

Perfsonar

PerfSonar util's repository contains a init folder with two bash scripts, one for setting up PerfSonar Toolkit and the other to set up PerfSonat Testpoint, both in a docker container.

As mentioned, the Toolkit version is considered the main node and the tests with the Testpoint nodes need to be configured through a request to our Middleware. In the middleware there is also a configuration file which can be editted so as to customize the relevant tests to be scheduled when this endpoint is called.

PerfSonar tests will only run if the two participant nodes are synchronized through NTP and, therefore, we also include a ntp.conf file in the utils repository which will be included in the containers on start up. Requirements:

- docker
- Orwell's PerfSonar utils

6.5.2 CI/CD pipelines

For continuous delivery, when necessary, services were auto deployed when the main branch changed, using Github agents setup on the machines for their respective hosts. This pipeline allowed faster development cycles and quickly get the system running the latest version.

6.6 One Touch Deployment

Due to the containerization of the modules, Orwell's deployment can be as easy as creating containers on the desired hosts. This process could also easily be automated with simple scripts.

Since our system has some optional features and really customizable environments, we give users the freedom to really configure the application to meet requirements assuring a full production-ready deploy with, for example, a single docker-compose command.

6.7 Documentation Website

Considering that Orwell is an open-source project and that every system admin should be able to use this system and adapt it to meet the requirements, we created a documentation website, which is available at `https://orwellmonitoring.github.io/documentation/`. There we provide vital information about all the modules of the system and give examples of how to configure them. This website is built using Docussaurus[3], allowing us to write documentation through markdown files and, since we do not need a database and a backend, we can easily deploy this website in Github pages.

[3] https://docusaurus.io/

CHAPTER 7

Conclusion

The main goal of this project was the development of a toolset to monitor a 5G testbed. This toolset should include tools capable of monitoring the VNFs, the network and all the 5G infrastructure. The monitoring process should not need to access the VNFs, since they are private applications belonging to the NetApp developers during the testing phase. Thus, we treated all the VNFs as blackboxes which we could not access.

Looking at the list of objectives purposed for this project, it is safe to say that we met most of them including some extras, even though some were limited due to reasons that do not concern the group. We developed a toolset with more than one tool for VNF monitoring, including a preventive one that collects metrics through the hypervisor, tools to measure the performance and assure the safety of the network, a security validation tool and a 5G infrastructure monitoring tool. While working with these tools we developed some new ones that we are happy to share with the community through our open-source toolkit, under the GNU General Public License v3.0[GPL].

Due to the modularity of this system's architecture, Orwell can be adapted to meet different use cases and needs which allows this project to be useful outside its scope. We have also made a strong contribution to the 5GASP European Project which aims to boost the development process of 5G applications.

Throughout the development of this project we had to deal with a lot of challenges. These were only surpassed due to the good dynamic and dedication of every team member, including the advisors. This being said, this project was a valuable experience for all of us in many levels, since we had to develop something completely new to us, along with technologies no member had experience with, turning the process in a set of challenges that we had to pass.

Although the goals were met for the scope of this proof of concept, there are a few features and tweaks that require planning and implementation, both at the functionality and scalibility level.

First, since our context had a very limited number of targets to monitor, horizontal scalibility was not really a concern while developing. In a real, production environment, the Kafka broker and Redis cache should be deployed in scalable clusters to meet the testbed's rate of metric collection without losing performance.

The same applies to our translators. Ideally, the system should be able to deploy translators dynamically when necessary. This should be a trivial orchestration task, as launching a translator is as simple as running a container.

For an even easier setup of Orwell, a config based install script would be created. In this script, the user would indicate, in the configuration file, the components and respective environments and in what hosts they should be deployed. Our script would then clone the repositories accordingly and bootstrap each module.

Finnaly, we wanted to create a GUI with the goal of displaying the state of Orwell in real time. This application, designed for network admins, would function as a hub to access every other Orwell resource (Grafana dashboards, Grafana alerts, Suricata alerts, Infection Monkey reports). In addition to this, the state of the components would also be presented, allowing access to logs and statistics.

References

[1] A. Wolke and D. Srivastav, "Monitoring and controlling research experiments in cloud testbeds," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 962–963. DOI: 10.1109/CLOUD.2013.97.

[2] E. Magana, A. Astorga, J. Serrat, and R. Valle, "Monitoring of a virtual infrastructure testbed," in *2009 IEEE Latin-American Conference on Communications*, 2009, pp. 1–6. DOI: 10.1109/LATINCOM.2009.5305030.

[3] A. Wolke and D. Srivastav, "Monitoring and controlling research experiments in cloud testbeds," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 962–963. DOI: 10.1109/CLOUD.2013.97.

[4] D. Kaur, B. Mohammed, and M. Kiran, "Netgraf: A collaborative network monitoring stack for network experimental testbeds," *CoRR*, vol. abs/2105.10326, 2021. arXiv: 2105.10326. [Online]. Available: https://arxiv.org/abs/2105.10326.

[GPL] *Gnu general public license*, version 3, Free Software Foundation, Jun. 14, 2022. [Online]. Available: http://www.gnu.org/licenses/gpl.html.