

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria e Scienze Informatiche Magistrale

Inferring the behaviour and security of networked devices via communication analysis

Relatore:
Gabriele D'ANGELO

Presentata da:
Eugenio PIERFEDERICI

Sessione III
A.A. 2018-2019

A tutti i miei parenti, che con il loro caldo affetto e la loro disponibilità mi hanno sempre garantito un bellissimo ambiente dove crescere. In particolare a mio padre e mia madre, genitori veramente speciali, che con la loro guida e comprensione mi hanno permesso di diventare la persona che sono oggi e di raggiungere questo notevole traguardo. A mia sorella, che nonostante la distanza mi sta sempre vicino e mi fa viaggiare con lei.

Un particolare ringraziamento va a Cecilia, la mia ragazza, che da anni mi sostiene e incoraggia in ogni momento, anche quelli più difficili. È riuscita a trovare il meglio in me e mi ha sempre aiutato a farlo emergere.

Ringrazio anche tutti i miei amici, e in particolare Giovanni, che ogni giorno hanno condiviso con me difficoltà e successi. È grazie a loro che sono riuscito ad apprezzare ancora di più questo splendido periodo della mia vita.

Infine, uno speciale ringraziamento va al professor Gabriele D'Angelo, relatore di questa tesi, che con la sua passione, competenza ed estrema disponibilità mi ha guidato nel mio percorso formativo e nella realizzazione di questo lavoro.

Contents

1	Introduction	9
2	Background	11
2.1	The evolution of Internet	11
2.2	Tools for the traffic analysis	13
2.2.1	Firewalls	14
2.2.2	Anomalies detection	16
2.2.3	New powerful tools: Neural Networks	17
	Recurrent Neural Network	18
2.3	Behaviour analysis research	19
3	Problem definition	21
4	A model for the behaviour recognition	23
4.1	Everything is a Stream	24
4.2	Windowing	25
4.3	High modularity means easy cooperation	26
4.4	Some definitions	26
4.5	The main stages	27
4.5.1	Stage 1: properties extraction	27
4.5.2	Stage 2: behaviour extraction	30
4.5.3	Stage 3: behaviour aggregation	31
4.5.4	Stage 4: behaviour translation	32
4.5.5	Stage 5: result interpretation	34
5	An application for the behaviour recognition	37
5.1	Design	37
5.1.1	Core structure	38
	PropertiesPacket	38
	Behaviour	42
	AtomicBehaviour	46
	ServiceBehaviour	47

	DeviceBehaviour	49
5.1.2	Demo structure	50
	Stage 1	51
	Stage 2	52
	Stage 3	53
	Stage 4	54
6	Implementation of the application	57
6.1	The tools	57
6.1.1	Packet capture	58
6.1.2	Scala	59
6.1.3	Tensorflow	60
6.1.4	JSON	60
6.1.5	CSV	60
6.2	Stage 1 to 3: the Scala application	61
6.2.1	Packaging	61
6.2.2	Properties packet	61
	Finding the service name	63
6.2.3	Atomic behaviour	65
6.2.4	Service behaviour	67
	Service window behaviour	67
6.2.5	Device behaviour	68
	The role of the <code>ServiceManager</code>	68
6.2.6	The Engine	70
6.2.7	The sniffer	71
	The problem of the properties extraction	72
6.3	Stage 4: the Python application	73
6.3.1	The network model: LSTM	73
6.3.2	The dataset	77
	The structure of the dataset	77
	Dataset preprocessing	78
	Data gathering	81
6.4	Applications interaction: the file	84
6.5	Results	85
6.6	Real-time testing: the live version	86
6.7	The interface	87
6.7.1	Graphical User Interface	88
6.7.2	Command line	88

6.8	Some scripts	90
7	Knowledge applications	93
7.1	The basic interface	93
7.2	A behaviour encapsulation to detect anomalies	96
7.2.1	The design	96
7.2.2	Implementation in the demo	98
8	Conclusions	99

Chapter 1

Introduction

Nowadays everyone knows what the Internet is and almost all of us use it regularly; it has pervaded our life intensely and some of us almost can't anymore live without it.

In the last years the number of devices connected to Internet has been increasing exponentially and it has reached huge numbers. This means that we can access our devices from everywhere, but at the same time rarely we are aware of what those devices are doing other than what we use them for; an example of that is the fact that most people have at least one smartphone, but very few of them know what the device is actually doing, if it is spying them or with whom it is sharing the data.

That represents a big threat to the security of the unaware users and of the person around them. Luckily, in the last years the attention to security, privacy and awareness is ever increasing. This means that the users pay more attention to what their devices are doing and care about it, and at the same time many researcher are trying to find solutions to understand what the device do and possibly limit its capabilities.

While big software (like the operative systems and most mobile devices) make available an increasing number of tools to monitor the device traffic, some devices cannot be inspected or those same tools may be forged in a way that makes it impossible to detect some specific bad behaviours.

Therefore security firms, companies and researchers keep high the interest in any kind of analysis of the behaviour of the devices in the network, mostly for protecting the privacy of the user or the company, but also to detect and track malware [1]. Until now every successful method to detect or filter the behaviour of the devices has been trying to only detect anomalies or manually lock some specific behaviours; this means nobody ever tried to understand at a discrete level of accuracy what the device does relying only on the analysis of the traffic intercepted.

That's why in this thesis I will define a model able to detect the behaviour which is occurring on the device by the mere observation of its network traffic. For start I will define a model that takes the raw low-level information regarding the communications occurring, process them and return information about the high-level operations occurring on the device. I will then build a demo that uses that model and demonstrates its feasibility. The data used are low-level information on the traffic without ever trying to use sensible data regarding the information content exchanged.

This document is structured in five chapters. In the first one I will introduce what is happening, why we need to do this work and what are the main limitations of the current tools. In the second one I will analyze in depth the problem and I will try to explode it in many sub-problems. In the third I will finally introduce the solution that I theorized and modeled. In the fourth will be explained how I implemented the solution and I will expose all the problems and difficulties encountered. Finally in the last one I will draw the conclusions, analyzing the results obtained and the efficiency of the system.

Chapter 2

Background

2.1 The evolution of Internet

When the Internet was born, it was a niche thing; extended among a very strict network of well known computers, it was used to share research data and for communication [2]. Everything was strictly controlled and it was very easy to detect and block every attack. That was because the network was so small that the technicians could manage and check it manually, having a pretty good awareness of the network topology. In those days there were still no problems regarding malicious behaviours; there was no conception of a malicious agent willing to spend his life in specializing onto how to break the defences of a system or network in order to gain money from it.

Then the number of machines connected to the Internet started increasing so rapidly that people started losing control of it and at the same time the people started gaining interest on how to exploit this phenomenon to their own advantage. In response to that some specialists started researching tools, protocols and techniques to increase the overall security of the systems and networks. This phenomenon marked the beginning of a new era distinguished by an increasing number of people interested in exploiting the weaknesses of the network and a little ensemble of people that have always been trying to protect the Internet, the devices and the people that use it [3].

All this kept expanding until now that the network is so extended and complex that its really difficult event to track the path of the data. While on one hand it is true that the overall security of the systems and networks has been increasing, on the other there are still a lot of vulnerabilities both found and yet to be found. This means that other than keeping mitigating the vulnerabilities, we still have to improve our detection systems. The detection systems allow us to detect an ongoing attack right when it is in progress or after it has been completed (the first the best).

In addition to that, the workload of every device has been increasing continuously; this means that we all knew that soon or later we would have lost control over the network, and it's becoming every day more complex to understand the network traffic or to detect the presence of anomalies in it [4]. In some cases it takes months or even years for a company to detect that its network has been compromised [5].

Now we are at a point where everyone knows what the Internet is, it is a widespread presence in our lives and the number of people that knows how to use it is continuously increasing; but it has become extremely difficult to control what the devices are doing, who they are talking to and what they are saying. This is the reason for the current rush in finding the best tools that can reverse that trend and allow us to regain an understanding of what is happening in our networks and devices.

For example the people reading this thesis probably own a smartphone or even a home device (Alexa, Google Home...), but they don't actually know with whom it is communicating and what information it is telling them, if the microphone is currently sending data or the camera recording videos. A smartphone is equipped with GPS, microphone, camera, all tools that can be used to transform the smartphone into a spying device that we take with us all the time every day.

The sensibility of the users in that matter is growing stronger, but at the moment there aren't tool advanced enough to allow an aware and complete control of the network traffic. This is an extremely complex task to accomplish and until now it hasn't been fully fulfilled yet, there are only tools that allow its interpretation in a partial and uncertain way.

To make thing worse, in the last years there has been a huge improvement on the use of privacy techniques: cryptography started being widely used in everyday applications and not only for research or high-security communications. This is an enormous and fundamental step in the user privacy and reliability, but it gets in the way of detecting the behaviour of our devices on the Internet [6]. Cryptography is spreading like never before, meaning that all those analysis techniques that directly use the content of the communications, without ever considering the possibility of encrypted content, are becoming almost useless. That's why there is an ever increasing necessity of new and more powerful tools to interpret the traffic, specially now that content encryption breaks the current algorithms and increases the complexity for the new tools.

2.2 Tools for the traffic analysis

There are many kinds of traffic analysis; there is the one meant to understand what the user is doing in order to infer its preferences, for example in the field of marketing and advertising; there is also the one used to protect the user itself from the Internet or the same Internet from a malicious or careless user.

The kind that most interest us is the second one, meant to detect anomalies to the average behaviour on the server for security purposes. There are many companies that are in the market of analyzing the web traffic with the purpose of detecting anomalies or malicious behaviours. They aim to protect the company from any kind of malicious behaviour, be it an external attack or an employee that likes to play during work time. For this type of analysis there exists two types of tools: the ones that detect anomalies in the incoming traffic, when protecting the servers, and the ones that filter some connection types, be it a specific protocol or destination.

The first type is used to protect server farms, or single servers/services, from external attack or to detect when some users may have problems in the utilization of the service. It's the case of a malicious entity that behaves abnormally in the attempt of fulfilling its purpose, for example downloading sensitive data regarding other users or damaging the system. This is a very specific analysis because the standard behaviour is well known, so it's relatively easy to write a set of rules that must be respected.

The second type is mainly used to filter certain connection types in order to protect the system or to prevent the local users from having certain behaviours. It is a tool often placed in between the terminal devices and the network, so that it can both protect the local machines from the "unknown" and disallow local devices to get unchecked on the network. This is usually limited to filtering certain protocol types or host destinations without inspecting the true nature of the communication.

Both those tools are important and widely used, but those aren't enough when it comes to the variability of the behaviours that devices like a smartphone may have. When we are under a company network the company itself could disallow us certain connections at all, after all we are at work! This could be accepted, but when we are home we want complete freedom of behaving as we like; the same company may be willing to leave us as much freedom as possible, but it wants to protect itself from both external and internal threat (like an employee that wants to disclose business secrets). This is a much more complex task to accomplish; in fact the behaviour of the users is very diversified and

so are the responses from the Internet. For this reason is nearly impossible to specify a restricted set of rules that each connection must match without the user incurring into exceptions and too strict limits.

One possible solution would be to install a software on the device itself that can monitor the behaviour of each application on the network (e.g. Glasswire [7]). This could be optimal, but it requires us to have full control over the device and the assurance that the device itself is trusted. The device is considered trusted when we can trust that whoever is able to control and use it and we are sure that nobody has taken control over the device in an illicit way. Every time that we consider the trust level of a device, we must consider what trust we put on whoever owns it, whats the confidence that nobody took control over it and whats the trust we put on whoever ever contributed on the software running on the device (whoever wrote the operative system or any application running on it). Often this isn't possible because either the software running on the device is proprietary (e.g. smart home devices, automobile, ...), we don't have the permission to install new software (e.g. devices of the family members in the house) or simply we don't have control over the devices that can connect to our network (e.g. free Wi-Fi in public places, networks for University's students or the networks for employee's smartphones). Another flaw of that approach is that the device could be compromised, meaning that somebody could have corrupted the device in a way that it no longer behave as it should, or its administrator could have malicious intent, therefore the monitoring software may become unreliable. This is the reason for the necessity of a centralized tool that allows us to infer all the behaviours of the devices despite the type of connection used, the destination of the traffic or the encryption itself.

2.2.1 Firewalls

The main purpose of a firewall is to separate, like a wall, the internal network of trusted devices from the external network, typically Internet. This is a complex job, therefore there are many different types of firewall that differs in the level of inspection of the traffic and, consequently, the workload and complexity of the tasks on the system. The simplest version of a firewall is usually pre-installed on every home router; this is the cheapest firewall, in effect it requires the minimum amount of memory and processor, but its also the less precise one.

There are other types of firewall [8], in particular:

- packet-filtering firewalls. It's the simplest kind of firewall, and the first one ever used. This type of firewall filters the packets by applying a simple

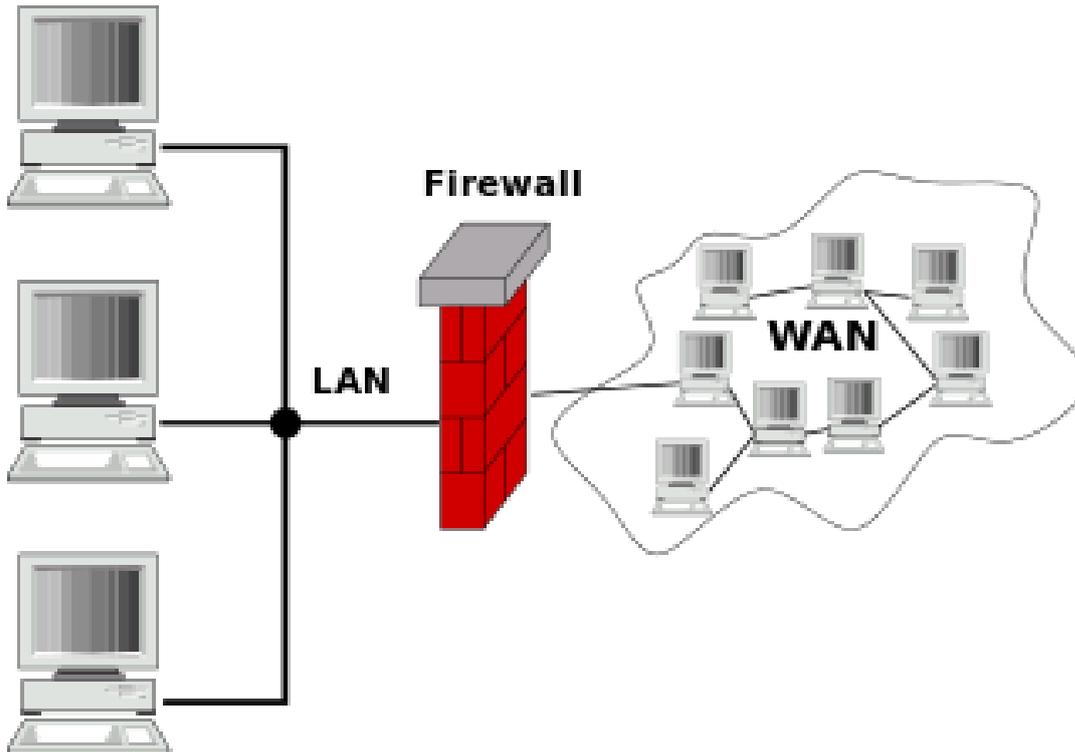


FIGURE 2.1: Simple representation of a firewall and its job.

Source: https://it.wikipedia.org/wiki/File:Gateway_firewall.svg

filter that checks parameters in the packet;

- stateful firewalls. It's an evolution in respect to the packet filtering ones. This kind of firewall preserve a state regarding the connection and are able to filter the packets with some more complex rules. For example it can allow packets of a connection that would have otherwise individually filtered;
- application-level gateways; this is the ultimate firewall that can understand certain applications and protocols that the other firewalls couldn't. It can detect even applications that try to bypass the firewall using disallowed protocols on allowed ports or similar attempts;
- next-generation firewall (NGFW) is a noteworthy, wider or deeper inspection at the application layer. An example of it is the deep packet inspection (DPI) that inspects the packet validate the type of protocol.

Those firewalls have existed for years and have been improved a lot since their beginning. Now we have very powerful, fast and optimized firewalls that allow us to filter network behaviours based on the protocol of communication; this is a much more powerful technique respect to a simple packet-filtering.

Those firewalls have grown very effective, but have recently met a new huge obstacle: the encryption of the payload. With the rising of encryption on the Internet, some certain types of firewall (like the application-level firewall) have been particularly limited and can't anymore check every connection. The only way to make the firewalls perform at their best would be to break the guarantees of the end-to-end encryption (that can't even be always done) otherwise we are forced to choose between a simple and blinded *allow-all* or *deny-all*.

2.2.2 Anomalies detection

Anomaly detection is a technique used to identify events that arise suspicions by differing from the majority of the data. In the network field it allows a network administrator to detect any kind of anomaly if he compares it to the majority of the other connections.

The objective of any tool of this kind is mainly to detect any kind of abnormal behaviour and notify the administrator or perform an action predefined; sometimes the detection can be easily accomplished by checking a threshold value, but other times it is much more complicated.

The task of identifying anomalies is a really complex one. In first place we need to understand what are the data that we can consider ordinary, then we can try to detect all the data that differ from that standard ones. The first step is achieved observing what is the structure of the majority of the data, and then we consider that as the standard. When talking of the Internet traffic, this is a relatively simple task when the traffic is limited to a certain type and a lot of users perform the same actions over and over again, the case of anomaly detection over a specific server (there already are researches even using Neural Networks algorithms [9]). When the tasks accomplished by the users on the Internet is much more variegated, the complexity rises enormously.

There is a huge difference between the variety of the communications on a server and the ones on a user device:

- a server must respond to all users in a similar way, its answers are all similar in structure and all users usually interact in the same way;
- a client device is definitely unpredictable and this complicates the identification of the expected behaviour.

The risk on a client device is that the anomaly detection algorithm doesn't perfectly fit the user behaviour, so the user may incur in limitations when he is doing a legitimate behaviour or some unwanted traffic could pass trough anyway.

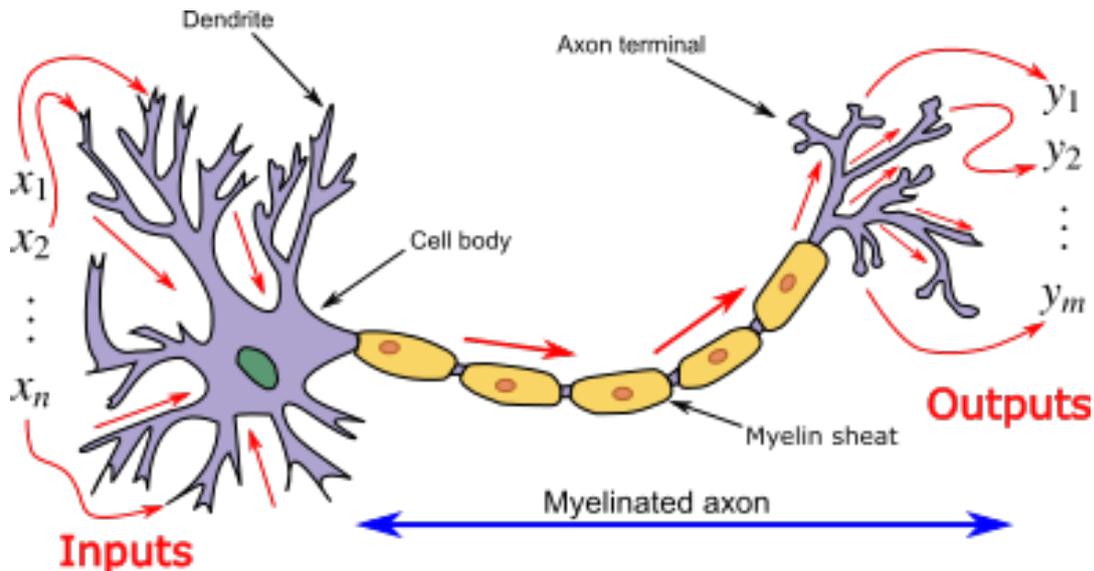


FIGURE 2.2: A neuron and its synapses. Some may be active and some inactive.

Source: <https://en.wikipedia.org/wiki/File:Neuron3.png>

For that reason the research has been focusing until now on anomaly detection server-side. The ability of identifying anomalies in the network behaviour of a device would be a huge progress and it could open to a lot of new capabilities for securing the Internet and protecting the users that use it.

2.2.3 New powerful tools: Neural Networks

Neural Networks (NN) are a computational technique inspired by the biological neural network of human and animal brains. The main feature of the neural networks is a training phase that allows the network to learn and generalize from a big number of examples. This means that the network can learn extremely difficult rules and patterns without the user specifying them explicitly (or even knowing or understanding them).

NN were theorized in the late 1940s but only the recent hardware allowed us to build and train them effectively. Since the proof of the big advantage brought from the use of NN, they have been spreading in almost every sector of the Information Technology (IT). Their adoption have drastically improved the performance, efficiency and precision of many systems.

The main principle behind a NN is to emulate the brain neurons with artificial ones: the nodes. Each node can transmit a signal to other neurons, exactly like the synapses in the biological brain. Each output of the neurons

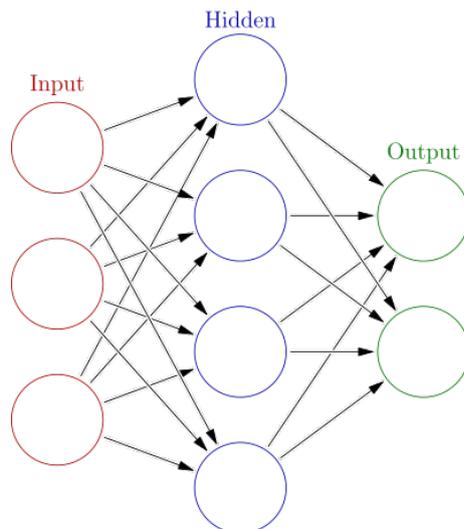


FIGURE 2.3: An example of layer representation. The number of layer is arbitrary. Each layer may have a different number of nodes and perform different transformations.

Source: https://en.wikipedia.org/wiki/File:Colored_neural_network.svg

is computed by some non-linear function of the sum of its inputs and usually have a weight that adjusts in the training phase.

Typically neurons are organized in layers. Each layer performs different transformation on its inputs. The first layer is the input and the last is the output, all the other layers in between are the hidden layers. The goal of a neural network is to elaborate the different inputs given to the input layer in such a way that the output layer returns a result as close as possible to the expected one.

Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of NN where the connections between the nodes may be retroactive, unlike the Feedforward Neural Network. This means that those are able to preserve sort of a memory of the previous states. With the ability of a memory, RNN can elaborate complex series of data that are connected to previous states or that can evolve in time.

This makes them able to process sequences of input data, like handwriting or speech recognition. RNN have been a revolution in this kind of systems and permitted a big improvement in tools like text predictions, complex speech recognition, games AI.

Another analogous flow of data is the network traffic, so RNN may be able to perform a more in-depth analysis of it. This means that their introduction could

finally allow us to gain a better understanding of actually what is happening in our devices, otherwise extremely difficult because of the quantity and complexity of the data shared between the devices and Internet.

2.3 Behaviour analysis research

This type of analysis has already been tried in the past and there exists some tools like those that implement DPI (Section 2.2.1).

Most of the research performed on the network traffic until now has been aimed solely to the detection of malware and anomalies. The companies have pushed to do it and to aim it at the detection of anomalies server side. Their purpose is to protect the server farms from possible attacks, to promptly detect an ongoing attack and to protect the privacy and the data of its users.

In this direction there are many different research and approaches. The standard one tries to either detect the application protocol by putting a third party between the user and the network (proxy) or to inspect the payload of each packet in order to infer the application used from it. [10]

There are also different approaches, for example it has been tried to detect and classify malware through the observation of the artifacts ordering. [11] This research has obtained promising results in the classification of different malware and could be coupled with other identification techniques for a more precise identification of malware.

Other than that the main problem of all those research is that they focus on the problem of identifying malware. Some of them try to do that at a low level. This kind of works one way or another give for granted some kind of knowledge of what is the standard expected behaviour. Some of them consider only certain types of applications and check them in a standalone way. [12]

Some others consider of being executed on the same machine. This means that they are always able to 'cheat' and obtain high quality information regarding the application-level application executed without the necessity of inferring it. [7]

Although this may be acceptable in most cases, our goal is to infer the behaviour without any information from the device.

The big difference between this work and all those tools is that its focus is to obtain the results observing the packets at a low level and without inspecting its content or gaining information of any kind from the device.

This could enable us to also infer the behaviour when the application encrypts its payload.

Chapter 3

Problem definition

The number of devices connected to the Internet is ever increasing. Many users even expose those devices directly on the Internet without knowing it. This is very dangerous and is becoming a big and complex problem.

To the current situation neither the network administrator or the user itself are able to check and control the behaviour of every device in the network. This problem is mostly felt in the enterprises, because of the business secrets, but it is even more sensible in the privacy of the home.

Every user in a home owns many devices, each one of them made by different producers. Each one of them then mounts a firmware that runs many different applications from different developers. This means that the number of possible threat agents is extremely high. Even if not even one of those agents is malicious, they may not have the time, or the will, to patch the vulnerabilities of the software.

All those factors are exposing us and our personal life, and the worst part is that we don't have any instrument to detect the type of communication occurring between our device and some third party. The main purpose of this thesis is to define a model and define a tool able to understand what the device is doing simply observing the network traffic of the device.

The problem can be broken down into a set of sub-problems:

- finding a way for intercepting the network traffic of the device;
- defining a model for the identification of the behaviour of the device;
- identifying a way for the identification of anomalies in the behaviours of the device.

The first problem can be resolved in many ways. For the purpose of this thesis we are going to suppose that we are able to intercept the traffic, for example because we are the administrators of the local network.

The second problem is the heart of this work and the one I studied in more depth. This is a problem extremely complex and requires a formal modeling, the definition of a whole new set of concepts and a new perspective on the problem able to find new ways to solve the problem. The greater complexity stands in the fact that the network traffic of a device used by an user can be extremely unpredictable. Some devices, like the embedded ones, may have a standard and well defined behaviour; but when the user interferes, like in a smartphone or pc, the behaviour is so unpredictable that identifying the boundaries of a certain behaviour is as much difficult as enumerating all the different behaviours allowed. For example a new behaviour could be a legitimate new operation permitted by the user as much as an unexpected and unwanted operation, possibly malicious.

Lastly the third problem regards the capacity of performing an analysis at a high level of the behaviours identified in the device. During this analysis we should be able not only to understand what's the standard behaviour of our devices, but also to detect any new anomalous one.

The main objective of this thesis is to build a theoretical model and, obviously, to test it. The objective is to define a model that is able to identify all the different typologies of behaviours a device can have only inspecting its traffic. The demo will demonstrate the feasibility of the project and the effectiveness of the model built.

Chapter 4

A model for the behaviour recognition

In the first phase of the work, I basically documented on the presence of other similar researches. I found out that there are many limited works mainly aimed to the malware identification. An example is the thesis of Hamidreza Aria, that focused its work in the identification of android malware [13]. Another paper tried to detect anomalies in the traffic of the single application [14]; in this case they were trying to self-updating malware in Android applications. In doing so, they installed a client on the Android device. All those solutions have two main problems:

- they all try to identify the presence of malware, and not the generic behaviour of the device;
- many of the solutions consider the possibility to install custom software on the device.

But I am trying to detect generic behaviours on any device, without any custom software installed on the final device or without limiting the research to the simple malware detection [15][16]. In this chapter I'll talk of the model that I've designed to accomplish that. It is able to recognize the behaviour of the devices through the analysis of its network communications.

The objective was to build a model that recognizes the Internet protocol stack, but it can be adopted to analyze any type of communication (Ethernet, Bluetooth, USB, etc.). Every single communication on every kind of interface brings information about what is happening on the device and can be used to deduce the high-level task executing on the device. Anyway, for the purpose of this thesis I will focus only on the network traffic, from the `network layer` up. This means that I will ignore the `link level`, ignoring if the device is connected through an Ethernet network or wireless or any other type (e.g. virtual machine).

I also tried to make the model as much generic and modular as possible; that way I will be able to extend it to use other communication interfaces or even the same interface but at a different level. This modularity also simplifies the work of improvement of the solution. This makes this solution extremely versatile to every condition and even applicable to contexts other than the Internet.

4.1 Everything is a Stream

Given the type of operation to be done and the type of data generated (a flow of information), I decided to manage the algorithm as a series of operations on a stream of data. The source of the stream is the series, possibly endless, of raw data intercepted during the communication. The output of the stream (the sink) is the series of *behaviours* recognized.

In the middle there is a set of layers of transformations to the stream of data. Each one of them will keep extracting, aggregating and transforming the information until we will obtain sensible information from the raw, bulky input.

The choice of the stream approach has been driven by two main factors:

- it allows an almost seamless parallelization of the work, so that it can sustain the large amount of raw input data generated at a possibly extremely high threshold;
- it permits a strict separation between the different stages of transformation of the data;
- it is a simple way of describing the sequential application of a number of function to the input packet.

The first factor also means that the model can scale very well and could also be applied to multiple devices in parallel, instead of only one at a time.

The second factor is extremely important for the modularity of the model: like for the `stack` TCP/IP, each layer is strictly separated and performs a specific task. That way we are able to replace every layer with a different one in a seamless way. For example we can perform the same kind of analysis on the same type of communication, but intercepted with a different tool or on a different interface, without the need of changing everything else except the first layer.

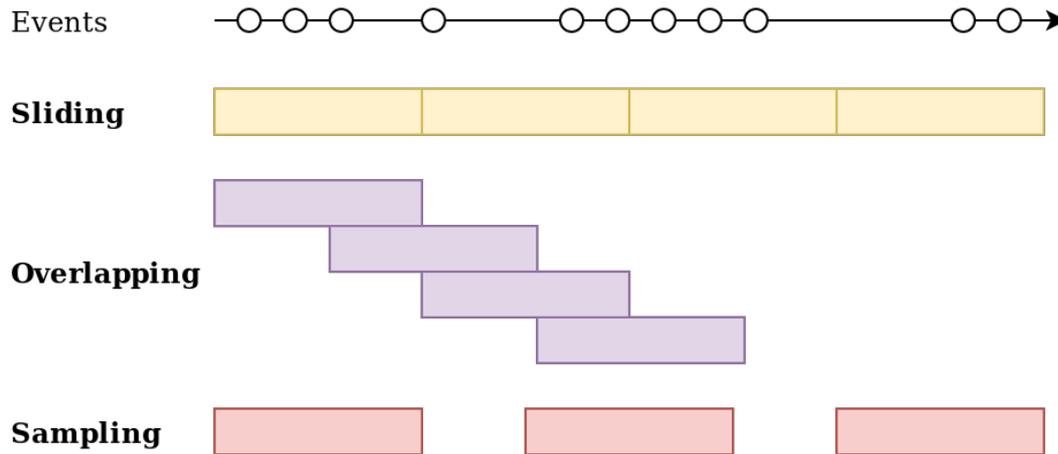


FIGURE 4.1: A graphical representation of the three types of window.

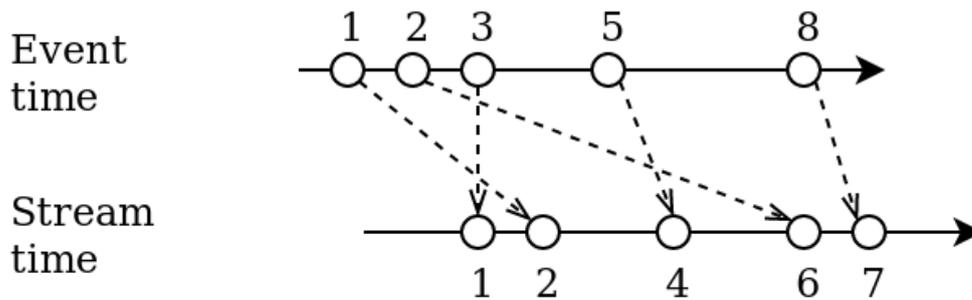


FIGURE 4.2: How the two times of a window affect the stream.

4.2 Windowing

Intercepting the network traffic means that there will be a huge amount of raw information flowing through the model. This incredible amount of data is extremely detailed and variable. To solve this problem I decided to adopt an approach based on windowing.

Windowing is a technique used to take the stream of data and group them in batches. Each window is defined by two different parameters: the *period* and the *size* of it. The first parameter defines how often we take the window from the stream of data; the second one defines how big is the window taken.

There are different types of window, depending on the value of the period and the size:

- **fixed**: when the period is the same as the size of the window;
- **overlapping**: when the period is smaller than the size of the window;
- **sampling**: when the period is greater than the size of the window.

We can define those parameters based on the time or on the data itself, but when we consider the time we can choose between two different times:

- **event time**: the time at which the event actually happened;
- **stream time**: the time at which the event entered the system.

The event time is more precise, but could cause problems when the events don't flow in order in respect to that time. In this case I was more interested on the precision of the readings and I could assume that the stream time respects the event time sequence. For this reason I choose to use a *fixed window* based on the *event time* and didn't worried about late readings. This choice gives me an extremely high accuracy and avoids the repetition of the data in multiple windows (*overlapping*) or the lost of some of the data (*sampling*).

4.3 High modularity means easy cooperation

A huge effort in the definition of the model went in doing it well and doing it with a look to the future. In this thesis I had to build a model from scratch and to do it at the best I could. In doing that I had to consider a lot of problems, but other than them I also had to consider the possible evolution of this model. The idea of this kind of traffic analysis is pretty new and it's in its infancy. For this reason in the future this could evolve a lot and I had to make the best effort in supporting, simplifying and helping this evolution.

The main technique I adopted is the design of a model that can be extended at will with an extremely low effort. Other than the subdivision in stages, the other big characteristic is the modularity of each stage. In the prospective of an open and cooperative work to improve this model, anybody could design and build a new module for a certain stage that could be used by other modules in turn.

The adoption of this technique obviously complicated the design process, but it already payed off in the demo and I expect that it will pay it off more in the near future.

4.4 Some definitions

In order to proceed with the creation of the model I needed to define some specific terms for the work I am doing.

- **device behaviour:** is the behaviour generated by the set of specific applications running on the device. The device generates a sequence of data that I am going to intercept and use to define this behaviour. My final goal is to deduce the set of applications running on it from the sequence of device behaviours;
- **atomic behaviour:** is a portion of the device behaviour. The device behaviour is deduced by collecting a set of atomic behaviour. While the device behaviour is the same for a respectable part of the stream, the atomic behaviour is different at each moment of the analysis and is different for each task attempted from the same applications;
- **properties packet:** it is a packet containing the sensible data for my analysis. By processing each raw packet of data I will extract only the properties useful for the analysis (or those I think will be) and those will be collected in this packet.

4.5 The main stages

As explained in the Section 4.1, I chose to manage the flow of data as a continuous stream, possibly unbounded. For this reason I defined some basic stages for the computation of the result.

1. *properties extraction;*
2. *behaviour extraction;*
3. *behaviour aggregation;*
4. *behaviour translation;*
5. *result interpretation.*

Until the pattern of stages and their partition is respected, it is possible to extend or replace each one of it independently extending the functionalities or precision of the system without the necessity of modifying all the rest of model.

4.5.1 Stage 1: properties extraction

This stage takes in input the raw data intercepted during the communication. Its main work is to extract and filter only the sensitive information and discards all the rest of the redundant or useless data.

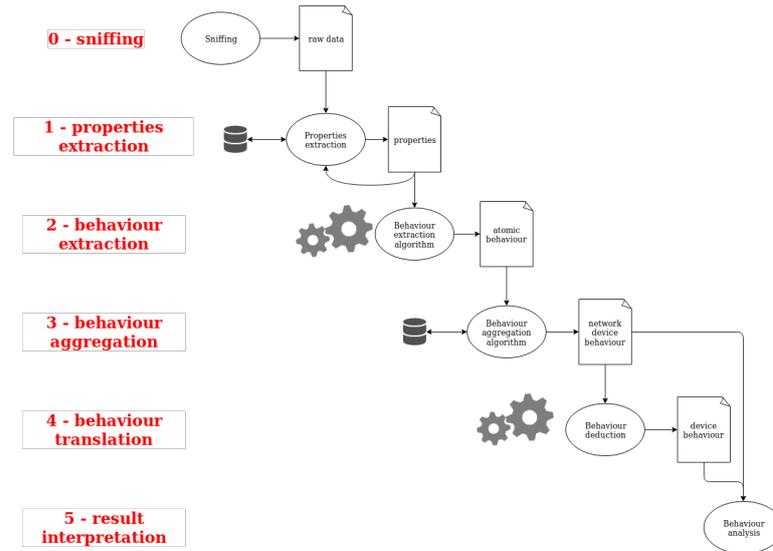


FIGURE 4.3: A complete representation of the full model and the interaction between all the stages.

The fundamental concept of this stage is that every single bit of information that may be used in the following stages, even if derived, must be computed and returned at this point. All those information will be contained in the *properties packet* (Section 4.4) returned.

The task is performed by the work of an ensemble of *modules* that respect the structure defined. The modules can be of two types:

- *direct*: this kind of modules only depends on the input raw data. In their transformations they apply simple functions to the data and return significant information;
- *indirect*: this kind of modules not only depends on the raw data intercepted, but also on the data generated by some other modules of this stage. That way it can use both the raw data and some extracted properties.

It is true that an indirect module could re-implement the procedure for the extraction of the information of the other modules directly from the raw data, but using the advanced-modules approach I avoided both duplication of code and I simplified the work of anybody that wants to extend a module. Anybody could extend a module for any reason; one of them could be to return redundant properties derived from other ones, but useful to compute for the following stages. They now don't have to know how the basic properties were extracted, neither they need to define again all the algorithms for the extraction of the basic properties.

An example of the indirect module could be when there have already been defined a direct module that returns the information regarding the time (e.g.

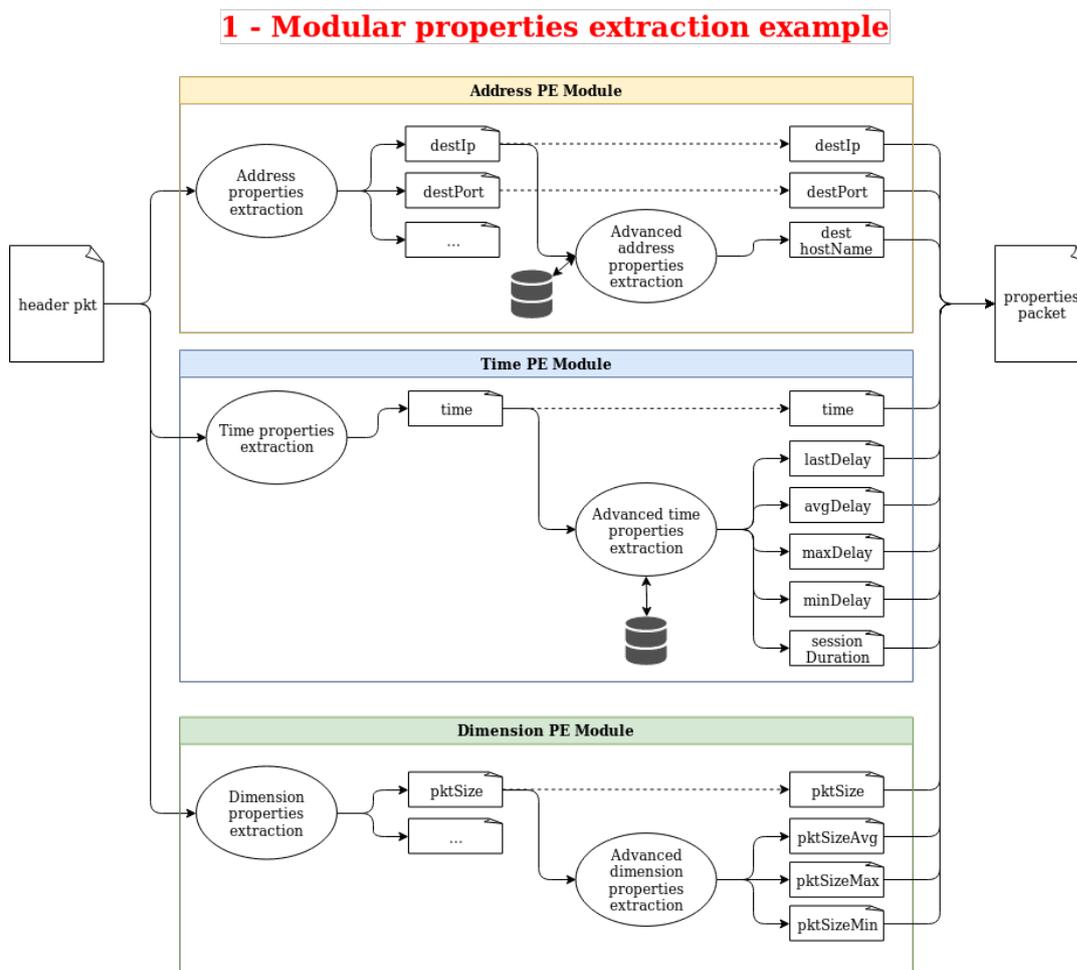


FIGURE 4.4: A detail of the first stage. It is highlighted the modularization with some examples.

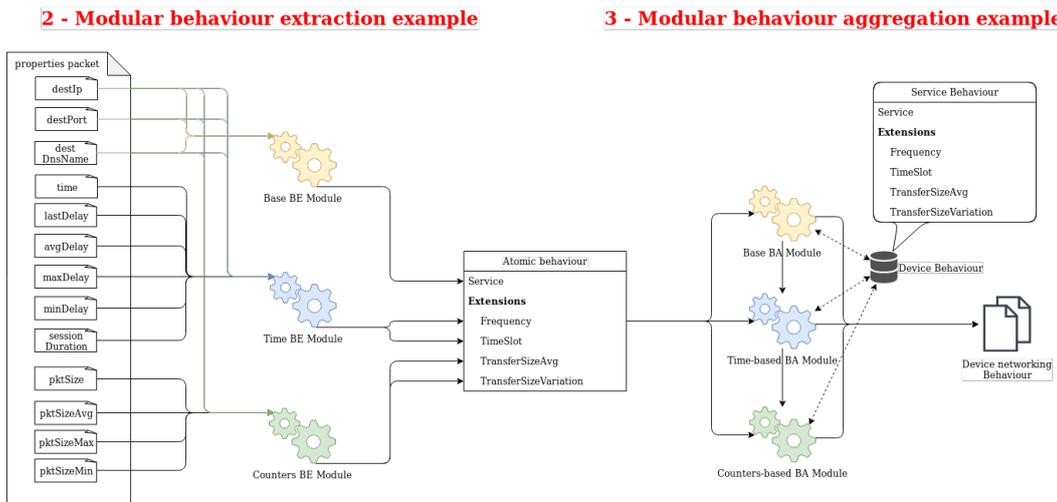


FIGURE 4.5: A detail of the second and third stage. There are some examples of modularization to better understand how it works.

event time) and another one that returns information like the size of the exchanged data. Well, then somebody could be willing to increase the precision the system while trying to use statistics regarding the size of the packet at a certain time. He could also discretize the information regarding the size or the time of the day (e.g. discretization of the time by hour, discretization of the size by order of megabytes).

Obviously the indirect modules could depend on other indirect modules too, generating a recurring pattern. This introduces a new problem, which is the order of computation of all the information. So when the system will be designed I will have to consider a method to extract all the properties by processing the packet through the modules in the right order. The solution to adopt will be freely chosen during the design of the model implementation and is not strictly defined in this model.

4.5.2 Stage 2: behaviour extraction

This stage is complementary to the next stage (Section 4.5.3): it prepares the behaviours in a redundant way, while the next stage aggregates them and returns the list of different behaviours without redundancy.

Anyway, as previously mentioned, the purpose of this stage is to transform the set of *properties packets* in input into the *atomic behaviour* describing the behaviour recognized at that precise moment. In order to do that, the stage combines the properties and uses a specific algorithm build with them the corresponding *atomic behaviour*.

Unlike the *properties extraction* (Section 4.5.1), these modules are of a single type (all direct). In effect the concept is that every module directly uses a subset of the input properties and generates a set of output features for the *atomic behaviour*. Everything that any module needs in order to compute the features must be provided by the *properties packet* itself.

The goal of this configuration is to model this stage as an direct algorithm that doesn't need any interaction with any kind of storage at all (e.g. database). This choice is determined by the fact that if it works as an algorithm, it can be enormously improved just optimizing it and without worrying of any external factor.

With a look to the future we may imagine how somebody could plug a module realized with a Neural Network (Section 2.2.3) to compute complex features from the correct combination of the input properties. This could allow an enormous improvement in the accuracy of the system [17]. More that anything else, it wouldn't need to extract additional properties or to create any kind of adapter for the network, it would just work and would just improve the system.

I also reckon that somebody might want to improve the algorithm. The person that tries to improve the algorithm could use new properties previously ignored as much as he could combine some properties already used by other modules in a different way. That's the reason why the modular solution is the best:

- it allows you to plug or replace modules to the algorithm to compute new features or improve existing ones;
- it doesn't require to modify the rest of the algorithm or even know or understand it.

4.5.3 Stage 3: behaviour aggregation

This stage is aimed at aggregating all the atomic behaviours received in input using some sort of algorithm. This operations permits the creation of the *device behaviours* (Section 4.4), the final representation of the behaviour of the device in that time interval.

In this stage we are going to take a set of *atomic behaviours* and we will collapse them in a single *device behaviour*. Usually the type of aggregations is made on the same static subset of features of the *atomic behaviour*, so it is one of the stages that is changed more rarely. As all the other stages, that can make use of a form of modularization too. Though, it may be of little use for both its complexity and the low frequency in changes; for this reason it's not

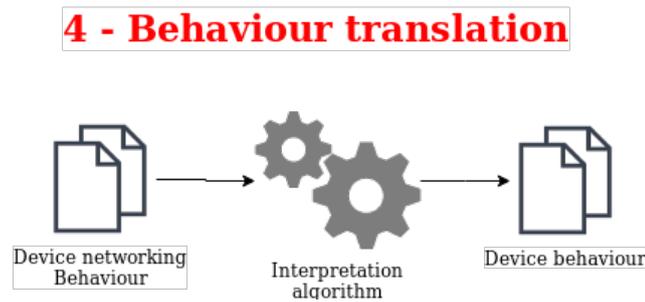


FIGURE 4.6: A detail of the fourth stage.

a requirement of the model that it is realized in a modular way. In some cases it could be useful, but in the most of them it is easier to rewrite the whole algorithm from scratch.

The task this stage has to accomplish is quite different from what the other stages do. That's why this stage has been created: to separate those tasks from the rest of the model. Another benefit of this choice is that now it can be realized once and used many times with different configurations. By isolating it, I tried to make the whole model as future-proof as possible. In effect, we could also use some advanced type of neural networks (the RNN, Section 2.2.3) to implement the algorithm without even touching one bit of the rest of the design.

It's fundamental to denote that the level of atomicity until now was the same of the intercepted raw data. Now it will be reduced (or increased, depending on how often the data transit on the interface), to a specific one defined at this level. The choices made in this stage are those who will define the level of atomicity for the next stages.

To accomplish that, I proposed to use a technique of windowing (Section 4.2) that permits to take some data in a fixed windows (like batches) that contains information that can be processed altogether. In particular, I suggest to use a fixed window based on the event time of when the data was in first place generated. This aggregation permits a better understanding of the device behaviours, that happens in a time interval, as well as it simplifies the elaboration of the data, given that we now know the actual frequency.

4.5.4 Stage 4: behaviour translation

This is the final stage of the analysis: the result returned from it is a set of labels describing the deduction of the application currently running on the device.

While until now the task accomplished by the stages was to transform and aggregate the information, the purpose of this stage is to infer something. This means that this is the first stage where we need to introduce the concept of a possible error or imprecision.

This stage is aimed to behave as a simple algorithm, without using any kind of storage during the intermediate work. The objective is to define a function, more or less complex, that is able to infer the set of application running on the device. This must be done with the lowest possible error (meaning that we would have a high accuracy of the results).

In input there is the *device behaviour*: a specific packet containing all the significant information that describes behaviour generated by a specific set of running applications. This data could be extremely simple, therefore pretty easy to read for the human, but it could also become extremely huge and complex, depending on the transformations accomplished by the previous stages.

Given the model and the complexity of the data, I immediately assumed that there will be used some kind of *Artificial Intelligence* to interpret the input data. The perfect example could be a *neural network* trained with a huge number of samples. Another factor in the complexity of the algorithm that justifies this choice is the elevate number of exceptions to consider. When we analyze the network traffic and we want to recognize the running applications, the task is extremely complex and there aren't easy rules to infer the application. In effect we just have to rely on the right definition of all the exceptions. To enforce the effectiveness of neural networks, I read many papers where they used those networks to detect the presence of malware in the network traffic [14][13]. It was even done a comparison between different networks used in different studies [18]. From this study it also emerged how most of the studies inspect the payload of the packets. This is a deep difference between my work and the previous studies.

An important observation is that often the same network behaviour could correspond to a different set of applications generating it. For this reason the ideal solution would be to design an algorithm that considers a bit of the **historic** behaviour to infer the set of applications running at that time. When I designed this model I was thinking that the best solution would be a *Recurrent Neural Network* (Section 2.2.3), so I tried to conform the model to it.

This is a fundamental stage for this model because it is the first and only layer that really actively use some form of knowledge to deduce precise information regarding what is generating the traffic. Indeed, all the previous stages passively used only a minimal knowledge or set of rules. They were just trying

to extract significant information, and not to use them to infer anything else. That's why we can consider all the previous stages as a structured form of *data preparation* for this stage.

If we inspect this layer in the perspective of the use of a RNN (*Recurrent Neural Network*), we can see how it is pointless to make it modular. In effect all the previous stages could be modular and tune/add more properties, instead this layer consider all the input together, processes them and returns the final result. Obviously it will be possible to tune the network in the future, but it will be a bulk operation consisting in the update of all the stage in its entirety.

In conclusion, I would like to state that I know that the choice of the Neural Network could be considered an early implementation decision that doesn't regards the model; at this regard I must say that the type of data generated by the network traffic would be extremely difficult, if not impossible, to understand for the human. Furthermore the number of exception to define would be enormous. For this reason I considered extremely useful for the realization of the best model to take for granted that the final realization of this stage will be done with a Neural Network; given that, I was able to tune the rest of the model around that choice.

4.5.5 Stage 5: result interpretation

Once the set of applications running on the device has been generated, we can combine it to other forms of knowledge. This combination allows both to inform and to protect the user that uses the device and/or the company who uses this solution, for example in its firewalls. The implications of the use of this model could be huge, and this is the layer where the new knowledge obtained is used to improve the systems or give the user an understanding of what is happening and what we learned.

At this stage we retain both the extracted *device behaviour* (or a set of them if we want to preserve the historic information) and the translated corresponding set of applications. Those data can be analyzed with a bit of historic knowledge, a set of rules or an external configuration to perform multiple operations.

An example could be an alarm system that, when detects an application with a specific behaviour, triggers an alarm that warns the user or blocks the device. Another example could be a firewall that uses this tool to gain an understanding of the applications running on the device to improve its policies.

All those operations that uses the results of the algorithm to control other systems or inform the user will be performed at this level.

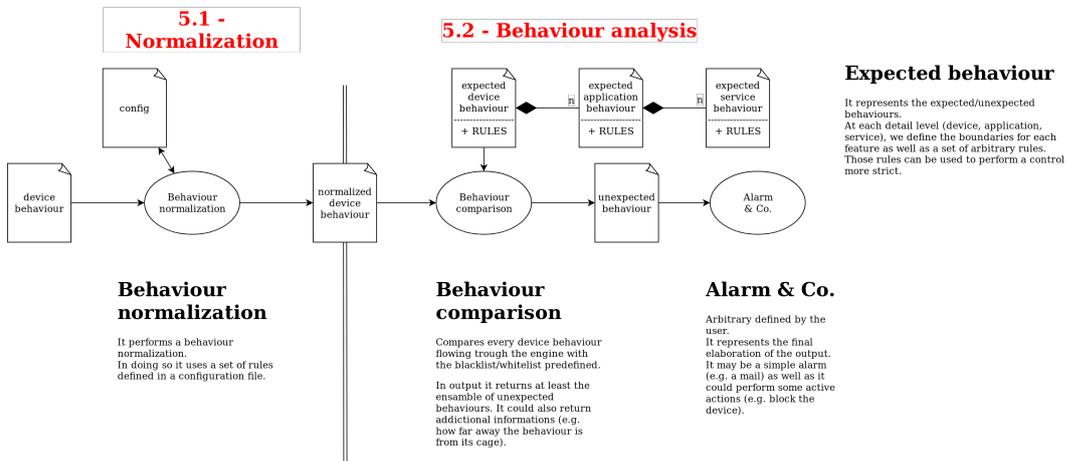


FIGURE 4.7: A detail of the fifth stage. The task executed in this stage are variables, so there is shown what we have in input and an example of possible interactions with external systems.

Chapter 5

An application for the behaviour recognition

There I present a possible application that uses the model introduced in the previous chapter. The aim is to demonstrate how the model works and how it could be actually used, but most of all, to test the model and verify that it can actually infer the device behaviour from its network behaviour.

In the first place, I designed the application according to the model, then I implemented it and tested the validity of the model. I tried to respect its modularity as much as possible, so that both the model and the demo can be extended for further testing/applications in the future. The design process is fundamental for the verification and demonstration of the theorized model. This means that the design of the application is almost as much important as the model itself. For this reason I had to spend the right amount of time to make the best design possible without rushing it up.

In this demo, I focused the case of use only to the analysis of the Internet traffic of networked devices. This allowed me to implement only the modules needed for that case. Despite that, it was important to reflect the flexibility of the model into the application itself. So I made a design so that it could always be reused and extended with many and many other modules.

Anyway, I limited the test cases of the application to the analysis to networked Android devices, like smartphones or some IoT devices (e.g. Google Home). This choice doesn't affect at all the design nor the implementation, it's a choice made for better defining the boundaries of the thesis and for simplifying the testing process.

5.1 Design

The application is subdivided in two great sections:

- *core*: containing all the basic structures required for the model to be defined and respected;
- *demo*: containing the specific extensions of the basic core structures, designed specifically for this demo.

First of all, I designed the core structure. It is imperative that it's boundaries and constraints are never violated, because they represent the model itself. After that I designed the demo section. This uses the core structure and extends from it. By doing that it takes the basic, generic model and specifies it to the case of the network analysis.

We can see the core functionalities as a core library that can be packaged and distributed. The second section is an application that uses the core functionalities and realizes its own solution for its own purposes.

5.1.1 Core structure

While the project started with the "simple" analysis of the network traffic, it would be useful to be able to apply this model to other kind of communication protocols (e.g. Bluetooth, USB, GPRS, ...). For this reason I designed it so that it could adapt to any kind of protocol or raw data.

The most of the work in designing the core structure went in the intermediate data (e.g. properties packet, device behaviour). In the following sections, I will describe in detail all those data structures I designed to support the model.

PropertiesPacket

This is the packet of information containing all the properties extracted from the raw data packet. It is generated by the first stage and used by the second one. Its properties are extracted from the original raw packet and are then normalized.

In order to grant the high freedom expected from a highly modular system, I inspired the structure of the packet to the one of the *SSL certificate*. Simplifying, those certificates have a main fixed set of attributes and a section containing all the custom extensions that anyone could define. In a similar way, I thought of organizing the packet in two sections:

- a first section containing all the fixed properties common to any kind of properties packet;
- a second section where all the extensions can organize their own properties.

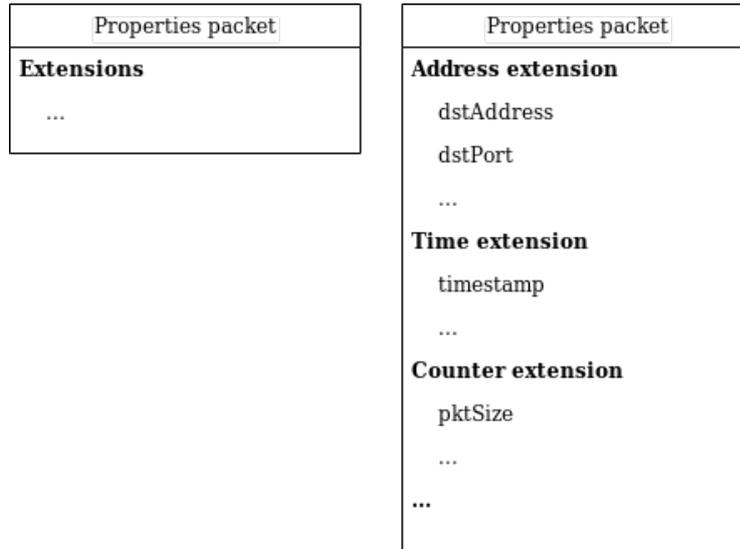


FIGURE 5.1: This is a representation of the structure of a *properties packet*. Here is highlighted how it is composed by a fixed set of properties and a number extensions, each one carrying its own properties.



FIGURE 5.2: This is the UML representing how I've designed the `PropertiesPacket` itself.

Given the high variability and low similarity between the various communication protocols, in the fixed section there isn't any common property yet. Despite that, the design remains like that because that leaves more freedom to a possible future standardization. In practice the properties packet, as it is defined now, is an empty canvas ready for adding any kind of new extensions.

In the actual design I made for the realization of the properties packet, other than the list of properties, it contains one other method: `extractProperties`. This method is used to perform the bulk extraction of the values from the raw data packet. In order for that to correctly work, every other extension must define how to extract its own properties in the `extractProperties` method.

With this design every property packet is empty when created, but it can be decorated with any of the extensions (according to the *decorator pattern* adopted). In the first stage, in order to compute and initialize the properties, it will be called the `extractProperties` method. This means that all the properties must be initialized in this method and not in the constructor.

There are two main reasons for this choice:

- to leave the construction of the packets as lightweight as possible;
- to define a strict separation between the packet creation and its initialization with the computation of all the properties. This separation will allow us to control at an extremely high detail level when and where (e.g. which thread) the bulk heavy computation is executed.

The extensions have been designed according to the *decorator pattern*. The pattern defines how to decorate an object with a specific *decorator*. In fact it is the case: I want to decorate the properties packet with as many extensions (the decorators) as I want. Considered that, each extension must respect this pattern and at the same time:

- define the list of the new properties;
- specify the function to extract the values of the properties from the raw packet.

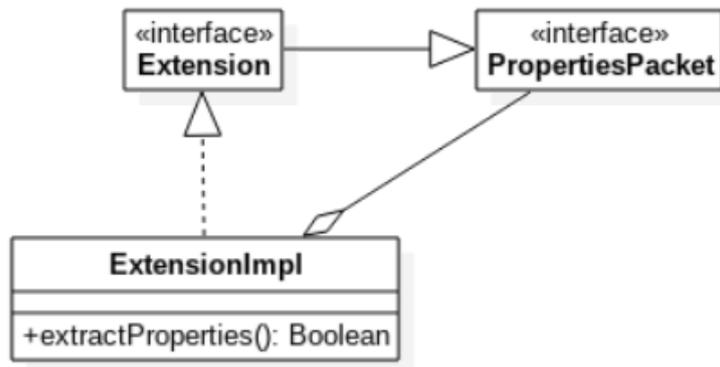
This structure allowed an extreme simplification of the way the extensions are created and appended to the basic packet. For example when somebody would like to compute the values of the packet, he wouldn't need to manually compute all the properties of each extension. It would be enough just to call the `extractProperties` once on the whole packet and it will compute every property in every extension automatically.

With this in mind, anybody could freely create new extensions any time in its own project. To create a simple extension, he just has to respect the given structure.

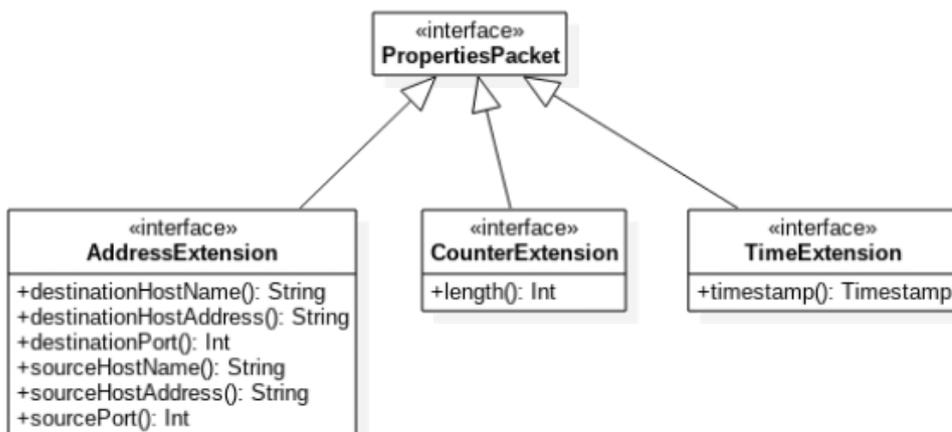
There is one exception in the computation of the value of the properties: when someone wants to reuse other properties already extracted. According to the model it must be possible to accomplish that; in this design it can be done by defining the decorator so that it decorates a packet that already has the required extensions.

In addition I designed some extensions in the core section too. Those are the ones that I believe are the most useful or, in certain cases, fundamental (e.g. the address or timestamp for the Internet traffic). If in the future other extension (proprietary or community ones) will be used in a consistent way, they will be added to the core section.

The extensions are:



(A)



(B)

FIGURE 5.3: This UML describes:

- (a): the structure of the extensions according to the decorator pattern I used during in the design;
- (b): some examples of possible extensions, those I considered fundamental for the work.

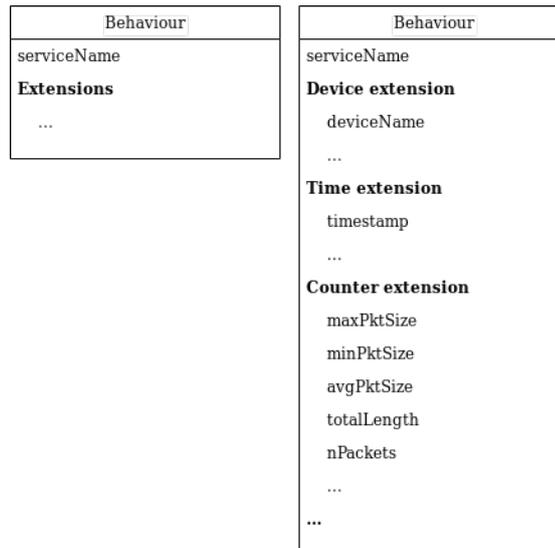


FIGURE 5.4: This is a representation of the structure of a generic *behaviour*. It contains the core attributes that every behaviour at every layer of atomicity must always have.

- **TimeExtension:** contains all the properties regarding the time, for now it is just the timestamp of the packet;
- **AddressExtension:** contains all the properties regarding the source and the destination of the raw packet. Regarding the Internet traffic it means the IP-address, the port and the host-name (redundant in respect to the IP-address, but often useful to have);
- **CounterExtension:** it is meant to contains a set of useful counters regarding all the packets. For now it just contains the full length of the packet.

Behaviour

It represents the abstraction of any kind of behaviour that we have in our model. It is the root of this data type and it is imperative that any kind of behaviour at any level always extends from this one. This choice not only allows us to define some fixed attributes that any behaviour should have, but it also permits to define an extension that can work at any level of aggregation of the behaviours.

Similarly to the `PropertiesPacket`, it is composed by two sections:

- a main fixed section, with the basic attributes of every kind of behaviour;
- a section for all the extensions and their attributes.

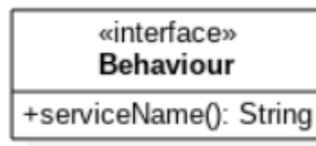


FIGURE 5.5: This is the UML representing how I've designed the **Behaviour** itself.

At the moment the only required attribute that every behaviour should have is the *service name*. This permits the identification of different behaviours and is obtained by the service that generated it in the first place.

As for the **PropertiesPacket** itself, I designed it according to the *decorator pattern*. This provides an easy and linear way to manage all the possible extensions and simplifies its management and design.

I also designed some basic extensions for the Internet protocols that I believe being very important and useful, if not fundamental. Those are:

- **ConnectionsExtension**: it contains significant information regarding the connection established by the device, for example the port number of both the source and the destination;
- **CounterExtension**: a set of attributes that store statistical data regarding the behaviour. This is extremely useful when aggregating multiple behaviours (e.g. applying the windowing);
- **TimeExtension**: one fundamental attribute regarding the behaviour, given the importance of the time, is the timestamp at which it occurred. For this reason it's almost impossible not to adopt this extension, whatever it is the analysis performed;
- **NeuralNetworkExtension**: this extension has been introduced when planning to introduce a NN elaboration in the fourth stage. It contains some useful information for the NN. An example is the discretized version of the **serviceName**. In effect strings of arbitrary length would be extremely difficult to handle.

There are 3 actual types of **Behaviour** in the design I formulated:

- **AtomicBehaviour**: the first kind of behaviour we meet in the flow, the one returned from the second stage of the flow;

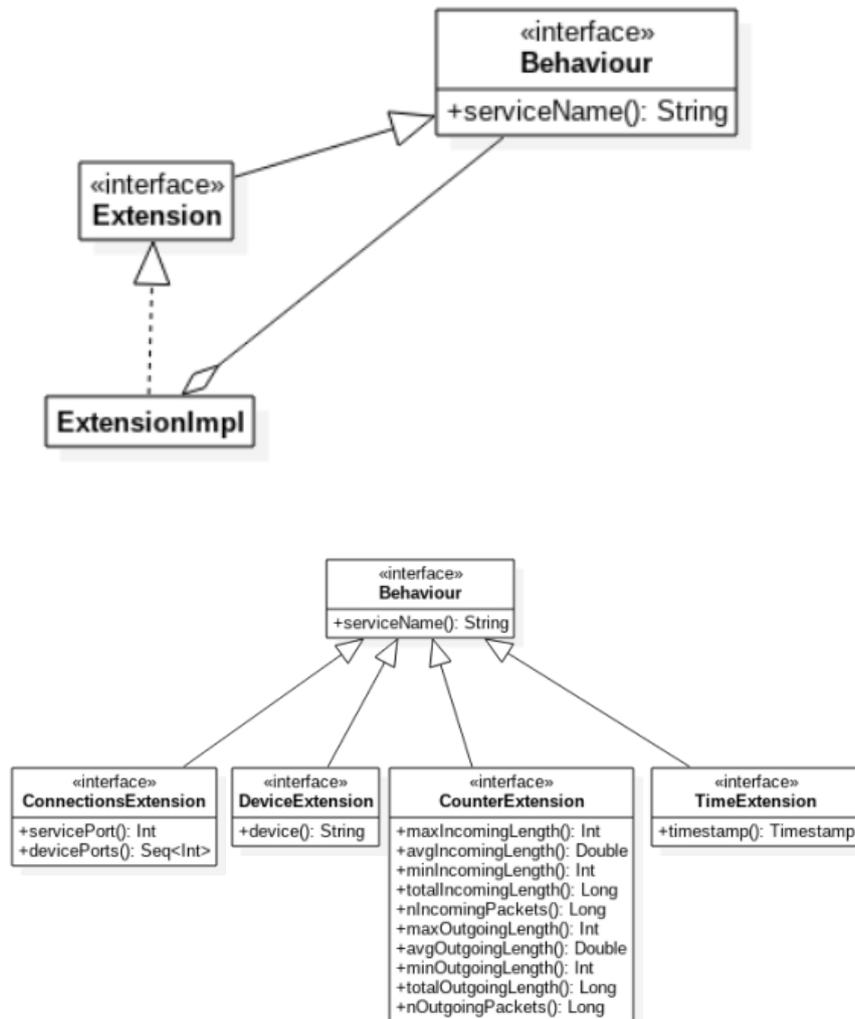
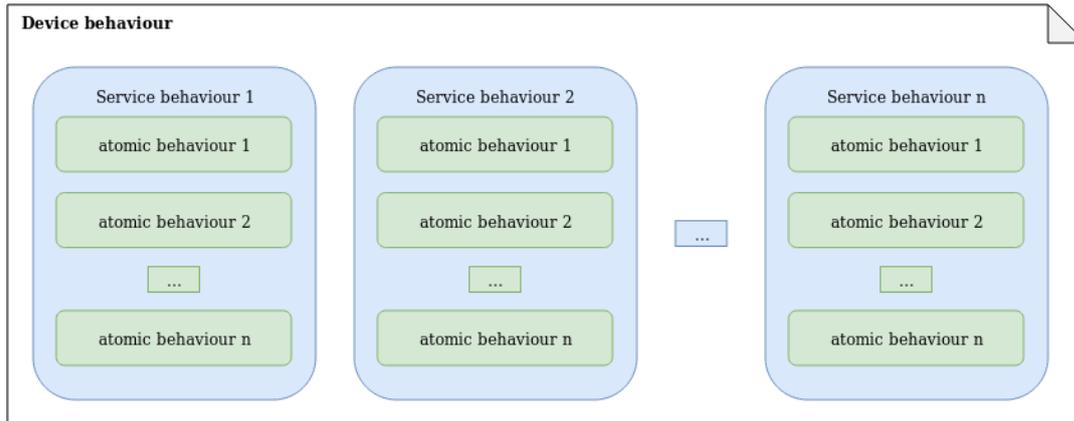
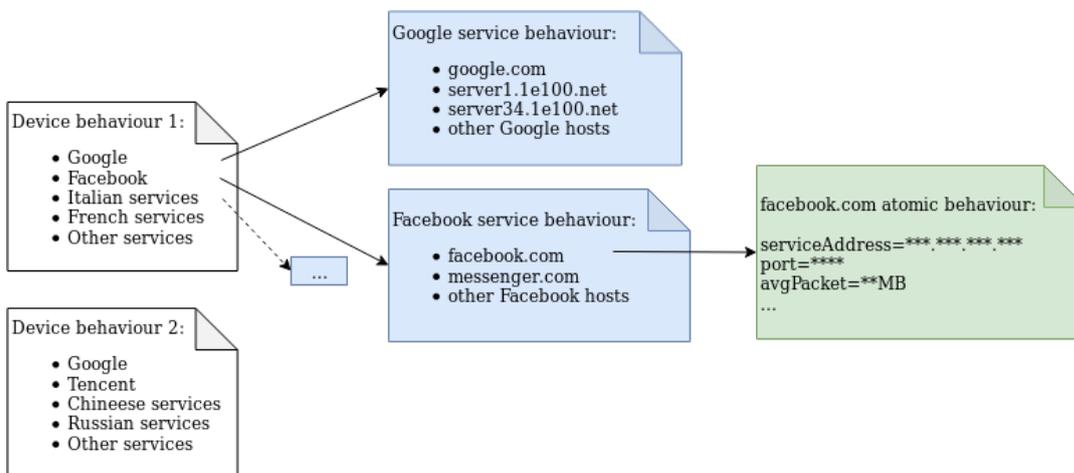


FIGURE 5.6: This UML describes the structure of the extensions to the **Behaviour** showing some examples of extensions I considered fundamental.



(A)



(B)

FIGURE 5.7: This image shows:

- (a): the relations existing between the different aggregation levels;
- (b): a representation of how the data are grouped together.

- **ServiceBehaviour**: represent the behaviour of a single service in a device. It aggregates all the atomic behaviours that a device generated in respect to a specific service;
- **DeviceBehaviour**: it is the behaviour that a device is having in that time. It is composed of all the service behaviours currently active from that specific device.

Both service and device behaviour are collected using the window defined by the model.

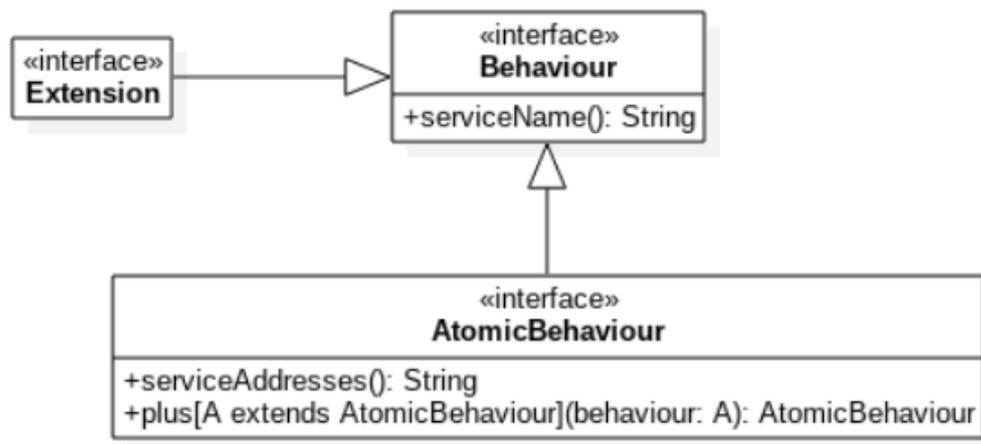


FIGURE 5.8: This UML describes the structure the `AtomicBehaviour` and how it is decorated with the extensions.

AtomicBehaviour

This first kind of behaviour is the one directly extracted from the properties packet. The atomicity of this behaviour is still the same of the source of raw packets. This means that it is the behaviour with the highest detail level, the unit of work of the behaviours. It is assembled during the second stage of the model.

As previously specified in Section 5.1.1, every kind of behaviour must extend the basic one, including this one too. This doesn't mean that the *atomic behaviour* can't have its own fixed attributes. Indeed, I already identified one fixed attribute: the service address. It represents the address obtained by the packet, that may be different from the service name already defined for any behaviour. As a matter of fact, the relationship between the service name and the service address is one-to-many, meaning that for every service name we can have many service addresses. For example one service (e.g. Google) could use multiple service providers (e.g. all the servers in the many clusters) or provide multiple services at once.

In regards to the demo test case, the Internet traffic, the *service address* will contain the specific host-name and port representing the specific communication session, while the *service name* will only contain the host-name of the destination. This allows us to acknowledge the different services despite aggregating them by service provider.

One useful feature of the atomic behaviour is its possibility to be aggregated with other atomic behaviours. This aggregation (through the method: `+(new behaviour)`) computes a new atomic behaviour. In fact the atomic behaviour

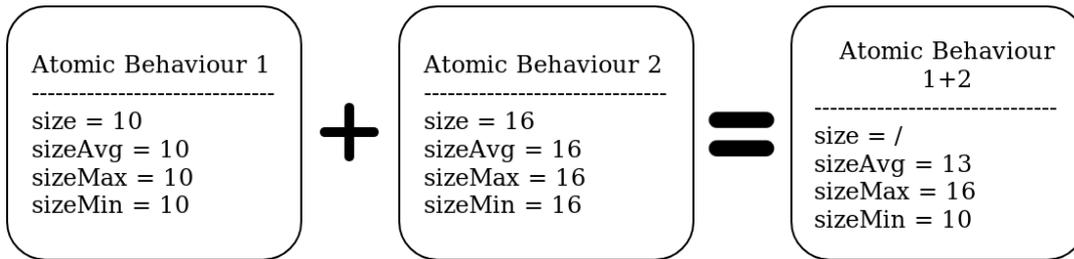


FIGURE 5.9: There is shown how the grouping of two *atomic behaviours* works.

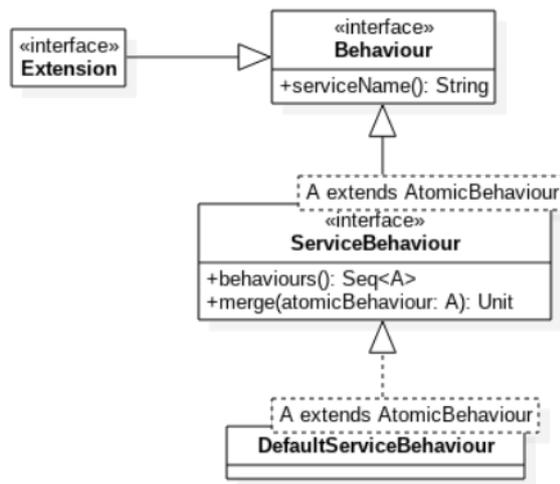


FIGURE 5.10: This UML describes the structure the `ServiceBehaviour` and how it is decorated with the extensions.

can be aggregated later on, providing all the same information but on a bigger set of data. The attributes of the new one are obtained through the aggregation of the values of all the behaviours that make it up. One example of use is when we want to aggregate the behaviours in the same window of time. Obviously the specific rules of computation of the new values depend on the choices of each extension.

ServiceBehaviour

This is another kind of behaviour, meaning that it extends the `Behaviour`, exactly as all the others. This one, though, isn't a simple behaviour like the atomic one; it is actually a behaviour of an higher level, that is composed by a set of the atomic ones. For this reason, other than the fixed attributes inherited by the *behaviour*, it also has a field that contains the set of the original `AtomicBehaviour`.

In effect, this behaviour was born to join all-together the atomic behaviours relative to a single remote service (distinguished by its `serviceName`). There is an important distinction to make: the difference between the *service name* and the *service address* resides in the fact that the first one generically represents a category of service (for the case of Internet it corresponds more or less to the service provider), while the second one represents the actual service provided. For this reason the aggregation of the service can be freely based on the value of the *service name*. That way the user can freely choose the level of aggregation he likes the best. All this is thought in a policy of greater freedom possible, requiring more work but rewarding with a design of higher quality.

A major reason for the existence of this behaviour is better understood in the perspective of the fourth stage, and most of all when thinking to the actual demo test case. On the Internet each `serviceName` (destination host) could have 65535 different `serviceAddress` (one for each port). Despite the fact that it's almost impossible that a device communicates with a service on every port, the dynamicity in the number of service addresses and the possible enormous number of them remains a problem. It can cause difficulties in the data preparation for the fourth stage of the model.

As previously anticipated, the fourth stage exploits a Neural Network. Those networks don't work very well with a variable-size input. In fact, the input layer is composed by neurons. Those neurons are in a fixed number, and it determines the size of the input vector. There are some possible partial solutions. For example, if the input is an image, it can be downscaled. Another possible solution is called zero-padding; it basically consists in setting a rather big input size, and then adding zeros to the input if it is too small. This meant that the best way to reduce the dynamicity of the input was to aggregate the *atomic behaviours* together, like it is performed in the *service behaviour*. Then, I adopted a solution similar to the zero-padding that consisted in specifying a bigger number of services and setting to 0 all the features of the absent services.

Other than the list of *atomic behaviours*, the *service behaviour* features a method, `merge`, that allows the service to merge-in other atomic behaviours. This method behaves in different ways, depending on the nature of the behaviour to merge:

- in case the atomic behaviour is already present in the service (there is another one with the same `serviceAddress`), it aggregates together the two atomic behaviours;

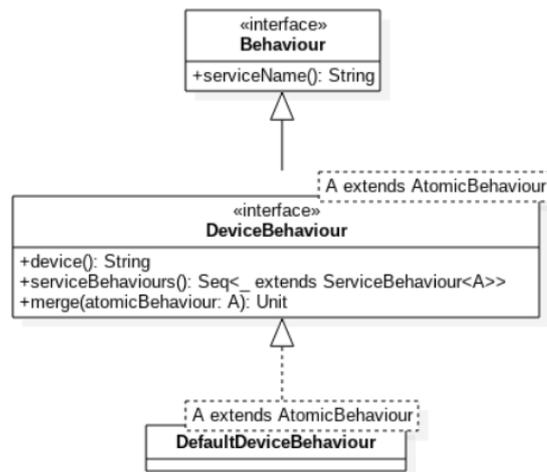


FIGURE 5.11: This UML describes the structure of the `DeviceBehaviour`.

- in case the atomic behaviour is new, it is simply added to the list.

The extensions for this kind of behaviour are the same specified for all the other ones (those specified for the generic `Behaviour`). The difference is that in this case their values will be computed as aggregation of the values on each atomic behaviour instead of extracting them from the properties packet.

DeviceBehaviour

This is the maximum level of aggregation of the behaviours. In the perspective of the analysis of a single device it is almost useless, but when we consider multiple devices (that's the case in almost every situation) it is a precious new structure.

It actually is the ensemble of all the atomic behaviours, grouped by service, that have occurred on the network for a specific device. It is composed by aggregating all of the `ServiceBehaviour` relative to a single device. For each window of time, it is returned a sequence of `DeviceBehaviour` out of the third stage, each one corresponding to a specific device.

Each one of the *device behaviours* is characterized by the name of the device that generated it and the list of service behaviours that the device produced. Cause the way that everything flows like a stream, it was useful to find a way to incrementally add atomic behaviours to the device. For this reason I designed

a method that takes in input an `AtomicBehaviour` of any kind and puts it in the correct place:

- if there isn't a service for that behaviour, it creates it and insert in it the atomic behaviour;
- if there is a service for that behaviour, it merges the atomic behaviour in it, as explained in the Section 5.1.1.

5.1.2 Demo structure

In this section it will be described the design of the actual demo application I realized. Obviously it uses the core functions already described in Section 5.1.1 but it extends and polishes them in the perspective of the Internet traffic analysis.

It is organized in two main layers:

- a top layer, containing all the generic modules required for the realization of the stages of the Internet analysis;
- a bottom layer containing the concrete implementations of the modules and the packets.

This separation is extremely useful for the standalone definition of the extension modules. In the specific, it permits to define generic stages that work with any kind of analysis until it is performed on the network traffic. It is even possible to design every stage independently from the others, just once, and use them whenever and wherever needed. This is useful in two situations:

- anyone can create a new module and plug it regardless of the stage implementation that is being used underneath;
- any stage can be designed autonomously from the rest of the application. Whoever designs the stages doesn't need to foresee what the source of the raw data will be or what modules will be used.

Those advantages come, in first place, from the separation applied between the design and the implementation. In second place, I worked hard on the design to make it so modular and grant all this freedom. A direct benefit of the modularity resides in the fact that, in the near future, it will be possible to prove the flexibility of the model just reusing this same demo.

For the purpose of this work, I only designed the Internet main stages, to support the Internet protocol. Each one of them computes the basic operations

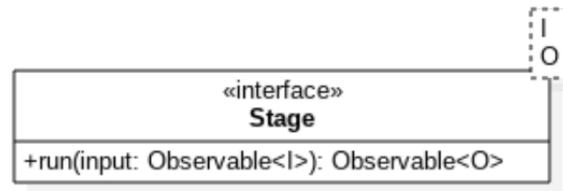


FIGURE 5.12: This UML describes the structure of the `Stage`.

required for the extraction of the required information. This information consists in the minimal amount of significant data present in the previous stage. Each stage specifies what are the basic modules required, but it is possible to add many more of them.

The specific design of each stage is deepened in the following sections. All those stages have in common the basic `Stage` from which they extend. It provides the definition of the main structure of a stage and how it can be executed, other than providing the management of the flow. It is a simple structure, but it permits the separation between how any two stages interact with each other and what each specific stage should do.

Stage 1

The first stage is the one in charge of extracting the properties from the raw data packets intercepted. Naturally it extends the generic `Stage` and specifies which are the input and output data types. As for the input, it remains generic. The reason is the fact that at this point we are still designing the solution and we don't know which sniffing tool will be adopted. The output, instead, is a `PropertiesPacket` with some fundamental modules for the network traffic (as the `AddressExtension`, that adds some properties for the memorization of the source and destination of each packet, or the `TimeExtension` that permits the memorization of the time at which the packet has been generated).

Those choices don't affect at all the possibility of adding more modules. They simply define the types of data the stage works with and what are the essential extensions. If anyone wants to add more, he can plug them without the necessity of modifying the stage at all. In effect the core of the stage only performs some basic actions, all the rest of the work is performed by the modules themselves. Just like that, we are now able to create any kind of packet decorated with an indefinite number of extension, as long as the minimal ones are respected.

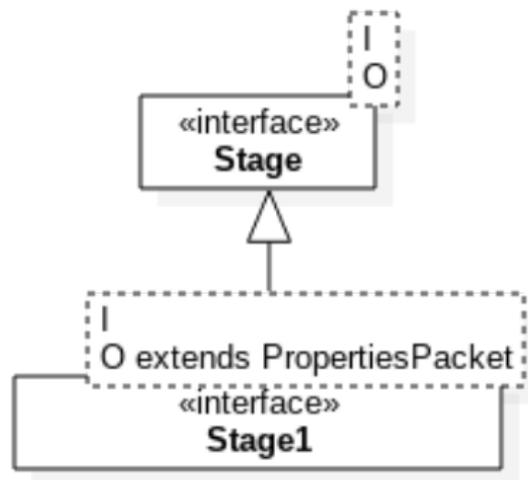


FIGURE 5.13: This UML describes the structure of the `Stage1`, in charge of the properties extraction.

To grant high modularity and extensibility, I used a *builder pattern*. It is used to create the desired `PropertiesPacket` with all the extensions the user wants to use.

One last thought is spent on the execution of the job. The extraction of the properties, despite being specified in the extensions of the `PropertiesPacket`, it will be computed inside this same stage. It is a powerful feature because it allows to adopt any kind of execution technique, for example to improve the performances of the system. Once the computation is completed, the resulting packet of properties is sent out in the stream to proceed to the following stage.

Stage 2

The second stage is charged with the task to extract the *atomic behaviour* from the specified properties packet. Similarly to the previous stage, that too extends from the generic `Stage`. It is structured in a similar way to the `Stage 1` and for this reason it enjoys of all the same benefits.

Other than the functions specified in the external modules, it also has a fundamental task: to distinguish the incoming traffic from the outgoing one. This job is accomplished by comparing the addresses of the source and the destination. Once we identified the direction, the extraction of all the other features is assigned to each respective extension module.

To grant high modularity and extensibility, I used a *builder pattern*. It is used to create the desired `AtomicBehaviour` with all the extensions the user wants to use.

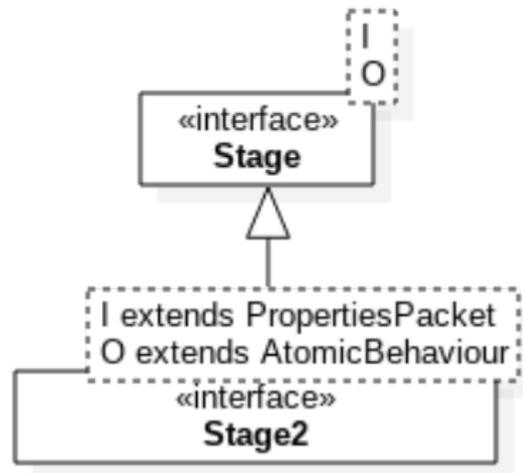


FIGURE 5.14: This UML describes the structure of the `Stage2`, in charge of the behaviour extraction.

Stage 3

This third stage is aimed at aggregating the *atomic behaviours* into *device behaviours*. Its design is similar to the previous stages, in effect that too extends from `Stage`.

While designing the stage, I had to keep in mind the fact that this stage performs a heavier computation than the other ones. This is to blame to two main factors. The first one is the fact that it needs to aggregate all the atomic behaviours. In particular it must:

- group in a device all the behaviour that are generated from or designated to the device;
- aggregates all the `AtomicBehaviour` of the same service into a service.

The second one is the fact that it also needs to fill all the gaps in the data intercepted. In fact, if a device didn't communicated for a while, it won't generate any form of network behaviour. Since the data flow is driven by the events and the windows are generated at the arrival of a new event, when the device doesn't communicate for more than the time of a window it will be lost. In fact, at the arrival of the new packet it will notice that the previous window was closed and will open a new one. If I wouldn't manually check that, all the intermediate windows would be lost. My solution was to manually check the difference in time between the events and, in case, it would emit as many empty windows (*device behaviour*) as many have been missed.

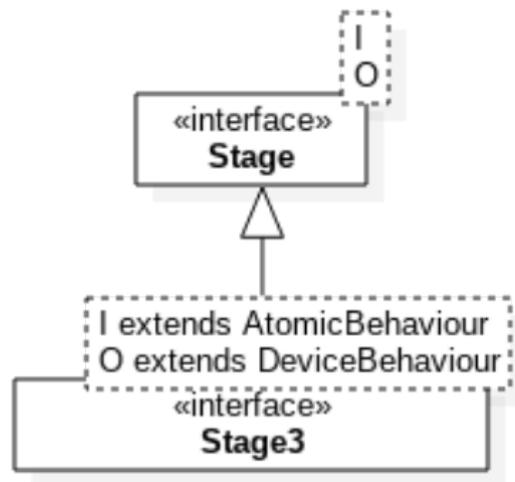


FIGURE 5.15: This UML describes the structure of the `Stage3`, in charge of the behaviour aggregation.

To preserve the high modularity, I adopted the *builder pattern* there too. As for the previous stages, it enables the creation of the empty `DeviceBehaviour` in an extensible way with as many extensions as you will.

Stage 4

In this stage we have by far the most complex task to perform: to infer the high-level set of application running on the device. Due to its huge complexity I immediately tried to find the best solution. I found out that the adoption of a *Neural Network (NN)* seems to be the best solution. In effect the algorithm would be full of exceptions, that NN seem to be great at handling. Other than that the data seems to be perfect for NN, after a first round of pre-processing.

Due to this choice, two new problems have emerged:

- how and where to find a data-set for the training, discussed in the Section 6.3.2;
- how to find which is the best kind of NN to use and how to model it.

The first problem was left to be solved later, in the implementation phase. Before that I was more concerned in defining the model for the network.

I immediately identified the necessity of using a *Recurrent Neural Network (RNN)*. The main reason is the necessity of preserving some sort of history. In effect, as previously said, the window itself is almost useless. The real information is gained by the analysis of a series of windows. After consulting

with *Professor Davide Maltoni* I decided to use a *Long Short-Term Memory (LSTM)*, a special kind of RNN discussed more in depth in Section 6.3.1.

This NN, as many other, requires a fixed input size. So I had to even think what would be a possible linear representation of a `DeviceBehaviour`. This wasn't an easy job, mostly because the size of the behaviour is extremely dynamic and the values of the attributes are extremely variable.

For example, if I tried to flatten each `AtomicBehaviour` by assigning a value to each feature for each service, we would have an array so big that it would be almost impossible to manage and it would decrease the chances of convergence of the network. In effect, for each service we would have as many dimensions as $65535 * \text{the number of features}$. This space of dimensions would be enormous.

For this reason I decided to:

- aggregate the services with some more sophisticated functions;
- discretize the address of each service by replacing it with a fixed integer value (using a map for the conversion).

The aggregation has been accomplished through an elaborate algorithm and a set of rules and exceptions. It uses the *service address* to group together all the *service behaviours* according to its rules (e.g. everything that hasn't been previously aggregated, that is "other" is grouped altogether). The direct advantage is that it enormously decreases the number of dimensions. At first I was afraid that the network wouldn't be able to converge with this level of aggregation of the data. But after some earlier tests I found out that it is a great solution.

The algorithm limits the maximum number of global services to pass to the network. This resulted in the necessity to aggregate together multiple different *service behaviours*. To do so, I designed a specific class whose task is to assign some values to specific services extremely diffused (e.g. Facebook, Google, etc), and the geographic location to the remaining services. Those values are the ones used to compare and group together all the services.

At last every `DeviceBehaviour` will be mapped in its corresponding vector. The historic information is preserved in the multiple instances of the same different behaviour in many time-windows.

The complete input vector is then composed as follows:

`window1 + window2 + ... + windowN`

Where N is the number of windows to provide to the RNN (the historic information). Each window is structured as follows:

`service1 + service2 + ... + serviceN`

Where N is the number of services that we want to use to describe the device behaviour. Each service is composed by the concatenation of all the direct attributes it has (counting the extensions too).

Chapter 6

Implementation of the application

In this chapter, I will describe the implementation process and the structure of the demo application. It is a prototype whose purpose is to prove the validity of the model and its main benefits. The main objectives of this applications, other than the one aforesaid, are to test the feasibility of the realization of the model, verify the presence of boundaries too restricting and, last but not least, to gather some actual data regarding the effectiveness and efficiency of the model.

The code of the application can be found on the repository at: <https://bitbucket.org/iugin/netbeh>. The prototype only focused on the analysis of the behaviour of Android device. It is only a choice made for testing reasons: I didn't have neither the time or the resources to train the system to perform an analysis on every possible different device. So I chose to focus the work on Android devices, conscious that in the future it can be exploited on every possible networked device.

6.1 The tools

The actual application is based on a widespread sniffing tool called pcap. In order to use it, I adopted the library **Pcap4J** (Section 6.1.1). This library works as interface with the aforesaid sniffing tool and permits to sniff the traffic or read pcap-dump files from any Java application. Its adoption allowed me to focus on the realization of engine and the tuning of the model, instead of having to manually intercept the packets from the network interface. It also exists a version of the library written in C/C++. Obviously the functionalities are the same, but I decided to use the Java version because it is a language at an higher level. In effect, it provides me an object-oriented language that can be compiled to be executed on any operative system. Another benefit of adopting the Java library is the possibility to include this library directly in a

Scala program. Scala is an advanced language that is described in the Section 6.1.2.

Despite being an high-level object-oriented language, Java still misses a lot of advanced features. An example is the complete absence of a way to realize multiple inheritance. Scala (Section 6.1.2), on the other hand, provides an easy way to realize the multiple inheritance, adds support to the functional programming and has some advanced features extremely useful.

Picking Scala immediately proved to be a good choice as I was able to realize a clean, easy to use, easy to understand solution in a relatively small period of time.

I was able to implement in Scala only the first 3 stages of the model. The fourth one needed the realization of a Neural Network. This can be an extremely complex task, when not provided with the correct set of libraries. For this reason I decided to implement this stage in **Python** (<https://www.python.org/>). This choice was driven by the fact that Python is a language widely used for Machine Learning and Neural Networks, so extremely rich of specialized libraries.

Even the Tensorflow library (Section 6.1.3) is provided for Python together with other tools that are extremely useful when performing this kind of computations, included preprocess a dataset, build a NN model, train and test it (e.g. PyTorch, Numpy, ...). Another great feature of Python is its ability to perform computations directly on vectors or matrices.

6.1.1 Packet capture



There won't be any engine if there isn't the fuel. The fuel that powers this particular engine is the network traffic. It consists in the packets exchanged during the communication between two entities. For this reason, the source of the stream is the sequence of packets exchanged between one (or more) device and the rest of the network. To capture this sequence of packets we need to perform an action called *sniffing*. The sniffing consists in intercepting and reading all the packets flowing through without blocking or modifying them in any way.

To accomplish this sniffing, I used an API called *pcap*. Its Linux implementation is realized by *libpcap* (<https://www.tcpdump.org/>), developed and maintained by *the tcpdump group*, while in Windows there is *WinPcap*.

There also exists a powerful command-line packet analyzer called *tcpdump*. It is maintained by the same developers that realized the *libpcap* library. It is extremely useful to sniff the traffic on an interface and then to generate some dump files. Those dump files can be used for a later bulk processing, instead of a live processing.

Pcap is an extremely widespread tools for the traffic analysis. *Libpcap* is also used by the same *Wireshark* (<https://wiki.wireshark.org/libpcap>), the world's foremost and widely-used network protocol analyzer. Pcap, and its libraries, already are pretty stable and rich of features. There is also library called *Pcap4J* that brings support to the pcap API on Java.

6.1.2 Scala



Scala (<https://www.scala-lang.org/>) is a general-purpose programming language providing support for functional programming and a strong static type system. Designed to be concise, many of Scala's design decisions aimed to address criticisms of Java.

The code written in Scala will be compiled in *Java bytecode*, allowing its programs to run on the JRE. Other than that, Scala also provides language interoperability with Java, so that Java libraries can be directly referenced into Scala code (e.g. Pcap4J).

Scala is a high level programming language that features not only the object oriented abstraction, like Java, but also the functional programming. The adoption of the functional programming enables an ensemble of useful programming techniques like currying, type inference, lazy evaluation, immutability and pattern matching. It is notable that the ability to adopt a functional programming permits an almost seamless use of many parallelization techniques of various kind.

It also allows a discrete speed-up in the development process, a simplification of the written code (that allows better understanding of the tasks accomplished) and, last but not least, an improvement in the performance of the program.

In addition to all that, a team is already working in the realization of a compiler aimed to compile the code into native binary (<http://www.scala-native.org/en/v0.3.9-docs/>). This will make Scala perform even better than the binaries executed in the JVM.

6.1.3 Tensorflow



Tensorflow (<https://www.tensorflow.org/>) is an open-source platform for machine learning. It is both used for research and production at Google. As many other libraries, it also permits the creation and training of a neural network model.

The library is rich of feature and tools, thus making it a very powerful platform. It is pretty easy to use and Google keeps spreading it. In effect it keeps adding support to other development languages; right now it supports Python, JavaScript, C++ and Java. Unfortunately, for now the Java version is experimental and badly documented, reason why I prefer to use Python, actually one of the best programming language for machine learning.

6.1.4 JSON

JSON (<https://www.json.org/>) stands for JavaScript Object Notation; it is a lightweight data-interchange format. There is a considerable number of libraries for parsing and generating it, and at the same time it is easy to read for the human.

It is organized as a structured object where the fields are couples key-value. Every value can be an object or an array, that is an ordered list of objects.

Given the flexibility and the huge availability of parsing libraries, it is a good solution to exchange structured information between different applications trough a human-readable file.

6.1.5 CSV

CSV, that stands for comma-separated values is a text file that uses comma to separate the values. It is the perfect solution to store tabular information in

plain text. Each line is a record, and every record consists of one or more fields separated by comma (or, occasionally, other custom symbols).

Given its ability to store tabular information, it is particularly useful to exchange data of this kind between different programs. It can also be used to export the data for human interpretation (e.g. by reading it as a sheet).

6.2 Stage 1 to 3: the Scala application

As previously explained, I implemented the first 3 stages of the engine in Scala, those that process the data from the single raw packet to the networking device behaviour.

6.2.1 Packaging

I immediately noticed a net separation between the generic Internet analysis and the specific use of the Pcap4J library. For this reason I choose to organize those two section in an independent way. That way both parts of the system are well separated and it will be extremely easy to plug other implementations for the sniffer while continuing to use the same *Internet engine*.

For example we could use another library for intercepting the traffic. Once I've realized the first stage for this new sniffer, all the remaining ones will be reused. At the same way we could reuse the implementation of the sniffer done with *Pcap4J* for sniffing other interfaces and protocols (e.g. USB, Bluetooth) by simply replacing the next stages.

6.2.2 Properties packet

The actual `PropertiesPacket` that I used for the Internet analysis, called `InternetProperties`, combines a specific set of basic extensions of the core module:

- `AddressExtension`;
- `CounterExtension`;
- `TimeExtension`.

All it's needed for us to do, is to implement the extraction function for the properties of each extension. The extensions in the core module define how every extension is structured, its properties and how it should behave; the actual implementation depends on the sniffing library used, so it still need to

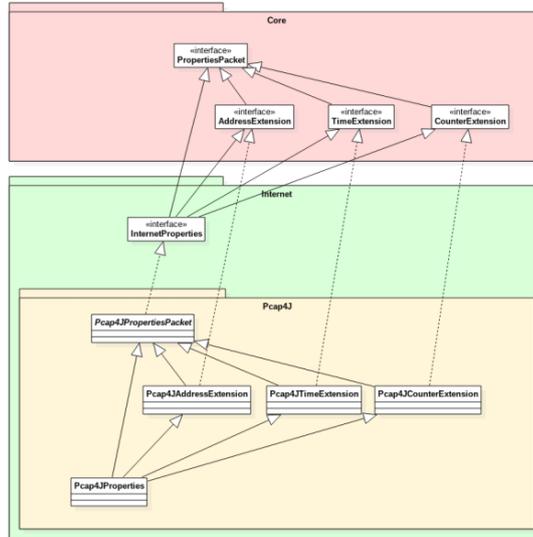


FIGURE 6.1: The UML of the properties packet implementation and its organization in packages.

be actually implemented. This has been done in the `Pcap4JProperties`, that realizes the *PropertiesPacket* using the Pcap4J library.

The implementation of the extraction function is spread in all the extensions, so that each one of them manages only its own properties, according to the modularity defined in the model (e.g. `Pcap4JAddressExtension`, `Pcap4JTimeExtension`, ...).

The set of extensions that I selected for those packets provides the fundamental set of properties required for the analysis of an *Internet behaviour*. Those are:

- *label*, required by the fourth stage only for the training of the Neural Network (this isn't strictly required for the Internet traffic);
- Address extension:
 - *sourceHostAddress*;
 - *sourceHostName*;
 - *sourcePort*;
 - *destinationHostAddress*;
 - *destinationHostName*;
 - *destinationPort*;
- Counter extension:
 - *length*;

- Time extension:

– *timestamp*.

Thanks to the design I made, implementing an extension is pretty easy. It's sufficient to extend its interface, that defines the structure each extension should respect, and implement the extraction function. The following is an example of the implementation of the *counter extension*:

```

1 trait Pcap4jCounterExtension extends Pcap4jPropertiesPacket with CounterExtension {
2
3   // Default values
4   override var length: Int = 0
5
6   // Extraction function
7   override def extractProperties: Boolean =
8     super.extractProperties && (for (
9       // Check packet not null
10      ethPkt <- Option(packet)
11    ) yield {
12      length = ethPkt.length()
13    }).isDefined
14 }
```

Finding the service name

When intercepting the packets with *Pcap4J*, we only have the IP address of each host (weather sender or receiver). This is an over-detailed and hard-to-read data for a number of reasons:

- the server of a certain service can change (e.g. it is part of a data-center, it has been moved, ...) causing the change of IP address;
- the same service can be delivered by different hosts, each one with a different IP address;
- the number of different IP addresses is incredibly huge and difficult to both handle and understand.

For those reasons, I decided to define a new parameter: the *hostname*. It is a redundant property derived from the IP address, but it's extremely useful to have. It permits to aggregate together many different IP addresses on the base of the kind of service (or the rules define by the programmer) and not on the specific IP address. This abstracts the concept of service from the specific IP address, that is a mutable information.

When realizing the system I incurred in a huge problem: the packet itself doesn't provide the hostname but only the address. Luckily *Pcap4J* provides a

function to perform a **rDNS** (reverse DNS) request. It is embedded in every packet as `getCanonicalName` and can be called anytime.

Despite that, I still had problems with it. First of all, it doesn't make use of any form of caching between different packets. Before this caching, the library opened a new connection to the DNS provider to solve each address of each packet. If the source of the packets is a pcap file, the resulting problem is only a lack of performances; but if we are performing a live analysis it results in an endless loop where the sniffer intercepts the new rDNS request and recursively tries to make a rDNS call for its host.

The solution I adopted was to implement a form of manual caching of the host addresses and its hostnames. To realize that I build an object whose sole purpose was to cache the host address-host name resolutions. Each time a packet wants to obtain its hostname, it is first checked if it is present in the cache, otherwise I make the online resolution and then cache the response. With that, for each different address there is only one online requests and all the others are solved offline with the cached value. This solved the problems by improving the performances and breaking the endless loop.

Unfortunately, the problems didn't end there. After solving this one, I noticed that some host addresses couldn't be resolved. This is due to the nature of the rDNS requests. They don't "magically" solve the hostname from the IP, but they query a specific kind of DNS records specifically compiled by the namespace owner to answer this kind of questions. In effect it doesn't either obtain the hostname originally used to obtain the IP address in question. Therefore, if the administrator of the DNS namespace doesn't provide the rDNS record, the request can't be accomplished.

To solve this other problem I had to use another tool: **whois**. Whois is a network protocol that permit to obtain information regarding an IP address and its organization/provider. The number of information varies from the network range of belonging, to the organization name and address, going on. It is composed by a server, public on the network, and a client, making the requests. The number of online servers owning the database with the information is considerable, but each one of them resolves only a subset of domains. There isn't a single server that can solve all the interrogations regarding any host. At the same time there isn't yet a protocol to chain them, like in the DNS servers.

There are different ways to build a *whois client* in Java/Scala, but none of that was fully functional. The main problem was the fact that I had to manually redact a list of all the possible whois servers to query, and then I should contact them sequentially. To keep the system working I would have

had to periodically update the server list, and it was a useless complication. For this reason I decided to use the native `whois` Linux client, the best client I could use. It is very well made and it manages all those problems by itself. Its server list is continuously updated and he uses it to query one server at a time until it obtains the information requested.

In my demo, I invoke the Linux command and then parse the first result found. The information I decided to take were only the name of the network (analogous to the hostname) and the geographical country, used in the third stage for a larger aggregation of the services.

6.2.3 Atomic behaviour

As for the *properties packet*, this too uses a specific set of extensions for the analysis of the Internet traffic.

This is used from the second stage on, so it's oblivious of what has been used to generate the properties that it want to use, as it should be. In effect from this point on the implementation is completely oblivious of the choice to use *Pcap4J* or any other sniffing tool to obtain the requested properties. This means that this stages can be used together with any kind of first stage (one for each kind of sniffing tool), until it generates the required set of properties.

The basic set of extensions making the *Internet atomic behaviour* are:

- `DeviceExtension`;
- `TimeExtension`;
- `CounterExtension`;
- `ConnectionsExtension`;
- `NeuralNetworkExtension` (only required for the training of the Neural Network at the fourth stage).

Altogether those extensions can represent any kind of Internet behaviour. Obviously, later on anybody can extend them with other extensions, for a more advanced analysis. Anyway those basic extensions cover any simple network analysis of the Internet traffic.

The actual implementation is accomplished in the `InternetAtomicBehaviour`. It is composed of all those extensions and defines for each of them how to compute its attributes. This means that the conversion from properties packet to atomic behaviour is accomplished with a code like the following:

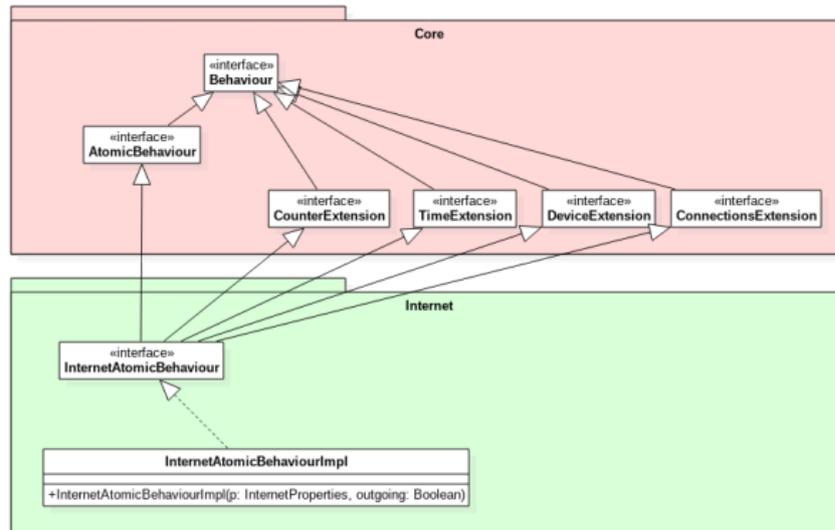


FIGURE 6.2: The UML of the atomic behaviour implementation.

```

1  /**
2   * Creates an AtomicBehaviour extracting the values from the single packet given.
3   * @param p the packet
4   * @param outgoing true if the packet belongs to the outgoing flow, false otherwise
5   * @return the AtomicBehaviour corresponding
6   */
7  def apply[T<:InternetProperties](p: T, outgoing: Boolean): InternetAtomicBehaviour = if (outgoing) {
8    InternetAtomicBehaviourImpl(
9      p.sourceHostAddress,
10     p.destinationHostAddress,
11     s"${p.destinationHostAddress}:${p.destinationPort}",
12     p.timestamp,
13     minOutgoingLength = p.length,
14     maxOutgoingLength = p.length,
15     totalOutgoingLength = p.length,
16     nOutgoingPackets = 1,
17     servicePort = p.destinationPort,
18     devicePorts = List(p.sourcePort),
19     labels = List(p.label)
20   )
21 } else {
22   InternetAtomicBehaviourImpl(
23     p.destinationHostAddress,
24     p.sourceHostAddress,
25     s"${p.sourceHostAddress}:${p.sourcePort}",
26     p.timestamp,
27     minIncomingLength = p.length,
28     maxIncomingLength = p.length,
29     totalIncomingLength = p.length,
30     nIncomingPackets = 1,
31     servicePort = p.sourcePort,
32     devicePorts = List(p.destinationPort),
33     labels = List(p.label)
34   )
35 }

```

Another important feature of the atomic behaviours is their ability to compare each other to identify identical behaviours. With this feature, the behaviours belonging to the same type can be aggregated together. Two atomic behaviours are considered identical when they have the same service name, service address and service port, even with different values for the attributes. This function is fundamental to aggregate different behaviours of the same protocol, without mixing different protocols up. It is implemented like follows:

```

1 override def equals(obj: Any): Boolean = obj.isInstanceOf[InternetAtomicBehaviour] &&
2   obj.asInstanceOf[InternetAtomicBehaviour].serviceName == this.serviceName &&
3   obj.asInstanceOf[InternetAtomicBehaviour].serviceAddress == this.serviceAddress &&
4   obj.asInstanceOf[InternetAtomicBehaviour].servicePort == this.servicePort

```

6.2.4 Service behaviour

The *service behaviour* is the aggregation of two or more atomic behaviour into one single object. The common characteristic of those atomic behaviours is their belonging to the same specific service. Other than the set of atomic behaviour, it also contains a sequence of features. Those features are defined in the extensions at the same way they were for the `AtomicBehaviour`, but this time they are computed as aggregation from its atomic behaviours.

To simplify the management of this new data structure in the stream, I defined a function that incorporates a new `AtomicBehaviour` in the current service behaviour. If the service already contains the atomic behaviour (has the same specific address), it chooses to aggregate the new one into it, otherwise it simply adds the new atomic behaviour to its own list. The code that implements that algorithm is the following:

```

1 service.behaviours.find(_ == a) match {
2   // If this behaviour already exists, merge it
3   case Some(b) => b + a
4   // If this is a new behaviour, add it
5   case None => s.behaviours = a +: s.behaviours
6 }

```

Service window behaviour

Given the fact that in the third stage the model requires to adopt a windowing technique, for every device I will have a sequence of windows, each one containing all the services identified in that time window. To simplify the handling of this complex structure, I defined a new data type that wraps the sequence of services of one single window: the *service window behaviour*. It has proven extremely useful when parsing the output files.

For what matters the Internet analysis, it has been implemented in the `InternetServiceWindowBehaviour`. It is a simple object that contains just the numerical window identifier (*windowId*) and the list of service behaviours occurred in that window.

6.2.5 Device behaviour

The device behaviour has the task to represent the networking behaviour of a specific device. To do so, it contains all the service behaviours intercepted for that device. It is then characterized by the device identifier (*device*) and the list of service behaviours.

Given the way we process the stream, proceeding atomic behaviour by atomic behaviour, it was provided of a method that automatically merges one new atomic behaviour into the device itself. It is similar to the function defined for the service behaviour, but this time it works on the whole device. First of all, it finds the service of belonging of that atomic behaviour, then it merges the new behaviour into it. The code charged to accomplish this task is the following one:

```

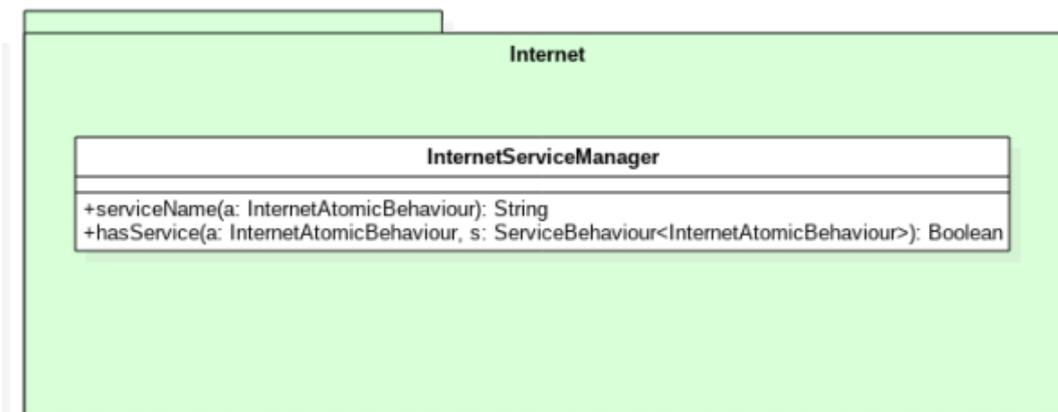
1  override def merge(atomicBehaviour: A): Unit = this.serviceBehaviours.find(ServiceManager.hasService(
      ↪ atomicBehaviour)) match {
2    // If there isn't any service behaviour for this atomic one
3    case None => serviceBehaviours = ServiceBehaviour(ServiceManager.serviceName(atomicBehaviour),
      ↪ atomicBehaviour) +: serviceBehaviours
4    // If there is already a service behaviour for this atomic one
5    case Some(s) => s merge atomicBehaviour
6  }

```

This device has a structure so generic that may fit almost any service behaviour inside it. For this reason, I provided a default implementation in the *core* section of the program. Its realization is trivial, the major complexity resides in the way we define to which service belongs an atomic behaviour. In order to separate the device implementation to the algorithmic choice of the service identification, I decided to use the `ServiceManager`. It is a utility class, provided during the construction of the device behaviour, that contains some methods to perform that specific task of service naming and comparison.

The role of the `ServiceManager`

During the design of the application, I incurred in the problem of passing the resulting device behaviours to a Neural Network. One of the problems was the dynamic number of service behaviours contained in each device behaviour. It may be empty (in certain windows) or contain only one service as well as it may even contain dozens of services.

FIGURE 6.3: The UML of the `ServiceManager`.

The possible solutions were two: I could either set a maximum number of services for each device and drop all the others, or I could apply a function to compress all the services into a fixed number of values.

In the first case, I would incur in the problem of a possible loss of understanding as well as the loss of precision, but, once defined, it would work seamlessly. Instead, with the second solution, I could write a function that aggregates all the services into a fixed subset, and I would fill the missing services with default values (zeros). This second solution would cause a loss of understanding too, but the loss of information would be minimal. The only flaw of this second solution in respect to the first one would be its necessity to be periodically updated when the network topology changes.

After considering both solutions, I decided that for the demo the second one was the best solution for both precision and comprehensibility. It could allow me to quickly implement a solution that would make it easier for me to check the validity of the model. Nonetheless, in the future it could also be replaced with a Convolutional Neural Network (CNN).

For this reason I defined the `ServiceManager`: a utility class that implements the specified solution through two main methods:

- `getName(serviceName: String)`: computes the new name of the service. This one can be the same for different services and depends on the actual implementation of the function. An example could be the aggregation of all the Google services, or all the Italian hosts, under the same hood;
- `hasService(a: A)(s: ServiceBehaviour[A])`: this method, instead, is used to check in which service an atomic behaviour should be put.

Given the atomic behaviour itself, it returns a *partial function* that takes a service behaviour and checks if it contains the specified behaviour.

6.2.6 The Engine

Once the structural objects were implemented, I had to realize an engine capable of performing all the computations according to the theorized model.

First of all, I defined the actual implementation of the stages from the first to the third so that they would use the correct version of the properties and behaviours (Figure 6.4). Each stage basically builds the object for the next stage and computes its values, as specified in the model.

The only exception is in the third stage, that has a more complex algorithm. It has the task to aggregate multiple behaviours into groups determined by the size of the window. During the implementation of this stage, I made the assumption that all the behaviours are occurring in an ordered way in respect to the packet that generated them (stream time and event time respect the same ordering). The assumption allowed me to realize a simpler, yet effective, solution.

Despite that, there still was a problem, as previously anticipated during the design: the stream proceed fluently processing the input events, but it doesn't consider the possibility of gaps in the source. In effect, the windowing depends on the event time, that is dictated by the packets themselves. When for a certain time, bigger than the window, the source doesn't emit packets, the stream won't notice the lack and won't count the missed window (or windows). So there can be missed windows, windows where no packets were exchanged.

As result of that, distant windows may result as close as sequential ones just because in the middle there is a sequence of missed windows. To avoid this loss, I modified the aggregation function so that, as it creates a new window, it also checks for possible gaps from the last window created, and eventually emits empty windows to fill the gap. That way in the following stage we can have awareness of the actual sequence of windows and the true distance between two of them.

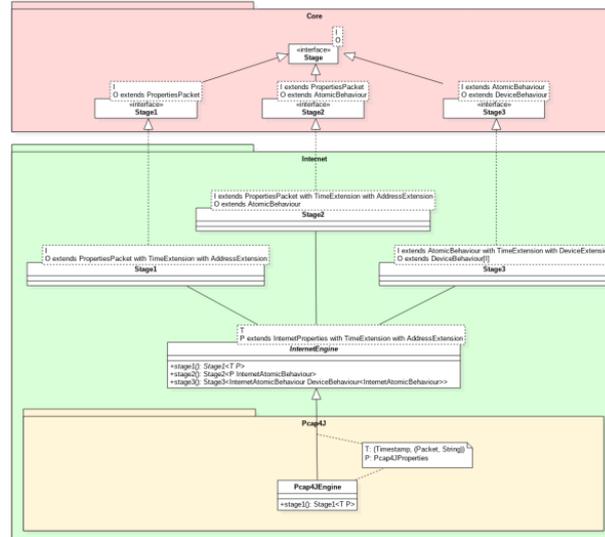


FIGURE 6.4: The UML of the implementation of the engine and its stages.

This engine, the *Internet engine*, processes all the data regarding network traffic. One major feature of this engine is its flexibility. In effect, made exception for the first stage, all the other stages can be used with any kind of sniffer, until it analyzes Internet traffic. The first stage is an exception because is the one that adapts the raw packet to the specific properties required, making the source independent from the rest of the engine (as well as the engine independent from the specific sniffer adopted). For this reason I modelled it in a way that separated the universally valid stages from the first stage, as shown in the UML at Figure 6.4 where the first stage is abstract and implemented in the *Pcap4JEngine*.

6.2.7 The sniffer

The sniffer is the source of the stream, the one that emits all the raw packets that will then be processed. As previously anticipated, I chose to use *Pcap4J* (Section 6.1.1). It is a library that provides an easy way to intercept the traffic on almost any networking interface of the device (e.g. ethernet, wifi, USB, ...).

To use it in my project I created a class whose only task was to interface with *Pcap4J* to create the stream of packets. Then I created the first stage of the engine, the properties packet and its extensions so that they extract their information from this specific type of raw packets. All this is implemented in the homonym package "pcap4j".

The problem of the properties extraction

Retrieving the data from the *pcap4j packet* wasn't easy. For each property to extract, we must check if the packet used that protocol, and only then we can cast the packet to the specific protocol from which to extract the information.

Doing this for each single property in each extension would have been an extremely redundant and complex code both to understand and to update. This is one of the reasons for which choosing *Scala* was a great idea: I was able to use the combination of some advanced language features to extremely simplify this process.

In particular there I used a combination of *implicit classes* and *for comprehension*. Through the use of an implicit class I was able to add a method to the pcap packet that, given a protocol, optionally returns its header only if present. Then every extension can use a for comprehension to retrieve all the properties using that method. By doing that, it automatically checks that every parameter is present in the packet at once. The following is an example of how the `Pcap4JAddressExtension` can extract its properties:

```

1  override def extractProperties: Boolean =
2      super.extractProperties && (for (
3          // Check packet not null
4          ethPkt <- Option(packet);
5          // Get ip address
6          dstAddress <- ethPkt.as[IpPacket].map(_.getHeader.getDstAddr);
7          srcAddress <- ethPkt.as[IpPacket].map(_.getHeader.getSrcAddr);
8          // Get port
9          dstPort <-
10             ethPkt.as[TcpPacket].map(_.getHeader.getDstPort) // If Tcp packet
11             .orElse(
12                 ethPkt.as[UdpPacket].map(_.getHeader.getDstPort)); // If Udp packet
13          srcPort <-
14             ethPkt.as[TcpPacket].map(_.getHeader.getSrcPort) // If Tcp packet
15             .orElse(
16                 ethPkt.as[UdpPacket].map(_.getHeader.getSrcPort)) // If Udp packet
17      ) yield {
18          destinationHostAddress = dstAddress.getHostAddress
19          destinationHostName = Pcap4jAddressExtension.getCanonicalHostName(dstAddress)
20          destinationPort = dstPort.valueAsInt
21          sourceHostAddress = srcAddress.getHostAddress
22          sourceHostName = Pcap4jAddressExtension.getCanonicalHostName(srcAddress)
23          sourcePort = srcPort.valueAsInt
24      }).isDefined

```

With this simple code the extension computes all its properties and returns true only if it successfully found the data for all the properties.

6.3 Stage 4: the Python application

In the design of the application, I decided that the fourth stage should be implemented with a Neural Network. This choice was driven by the nature of the data and the high accuracy obtainable with Neural Networks. For this reason, I decided to implement this stage in Python, a powerful language for implementing machine learning and neural network solutions.

As mean of communication between the third and fourth stage I decided to use the filesystem itself. The third stage writes a CSV file (Section 6.1.5) ready for the neural network, then the python application reads it and processes its content.

Once the application has the data, it can start processing them. It's important to know that, when working with neural networks, it isn't enough to design and implement them in order to evaluate the data; we also need to train it with a decent dataset. This is a fundamental step, without whom the model would be useless. In neural networks the correct training is even more important than building the perfect network. The main steps required for the correct training of a network are the followings:

1. dataset preprocessing;
2. model definition;
3. model training.

Instead, if we just want to evaluate some data on a network already trained, it's sufficient to load a trained model as follows:

1. dataset preprocessing;
2. model loading;
3. dataset evaluation.

Those steps are always valid, regardless the network built or the dataset used. In this specific case the type of data is discussed in Section 6.3.2 and the model is explained in Section 6.3.1.

6.3.1 The network model: LSTM

The task to accomplish in this stage is to infer what's happening on the device. This must be done trough the analysis of its behaviour generated on the network, that occurs while the time is elapsing.

In this case, the problem could be that the same network behaviour could be generated by different device behaviours. To avoid to be misled by that, I chose a neural network that considers also the previous states. Those networks are the Recurrent Neural Network, and in the specific I selected the one I thought being the best: *Long Short - Term Memory (LSTM)*. It is an artificial recurrent neural network architecture that, unlike standard feedforward neural networks, has feedback connections. This means that it can not only process single data points (such as images), but also entire sequences of data (such as speech or video or sequences of network behaviours).

A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. As many other networks, this too organizes its cells in layers. Each layer is independent by the other ones and can have a variable number of cells in it.

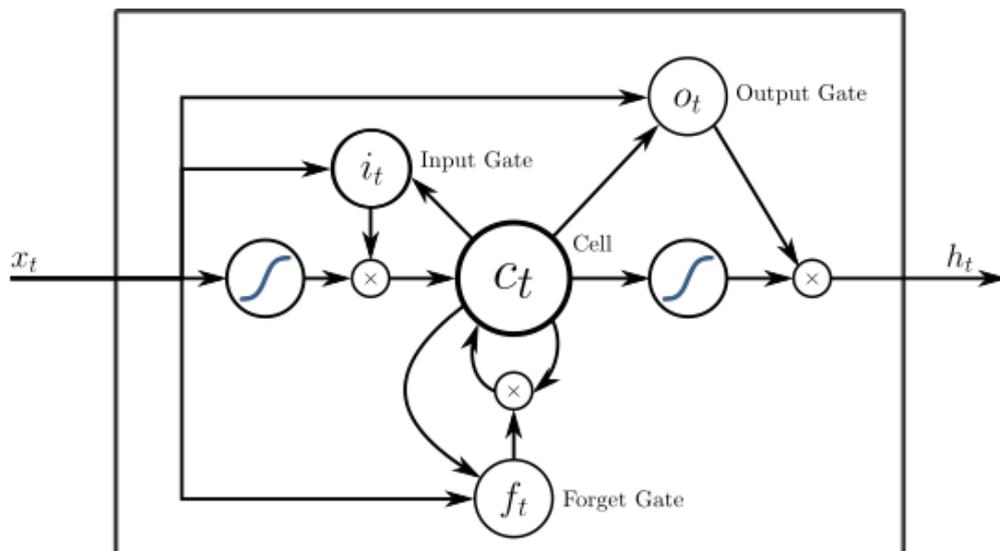


FIGURE 6.5: An LSTM cell with input, output, and forget gates.

Source: https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:Peephole_Long_Short-Term_Memory.svg

For the implementation of the network in Python, I used the Tensorflow library (Section 6.1.3). It provided all the required tools and extremely simplified the process of the network modeling. The actual model of my network is composed by 2 layers sequentially connected: a *LSTM layer* and a *dense layer* (Figure 6.6). The first layer is in reality composed by two different layers: the input layers, of the size of the input, and an hidden layer with LSTM cells.

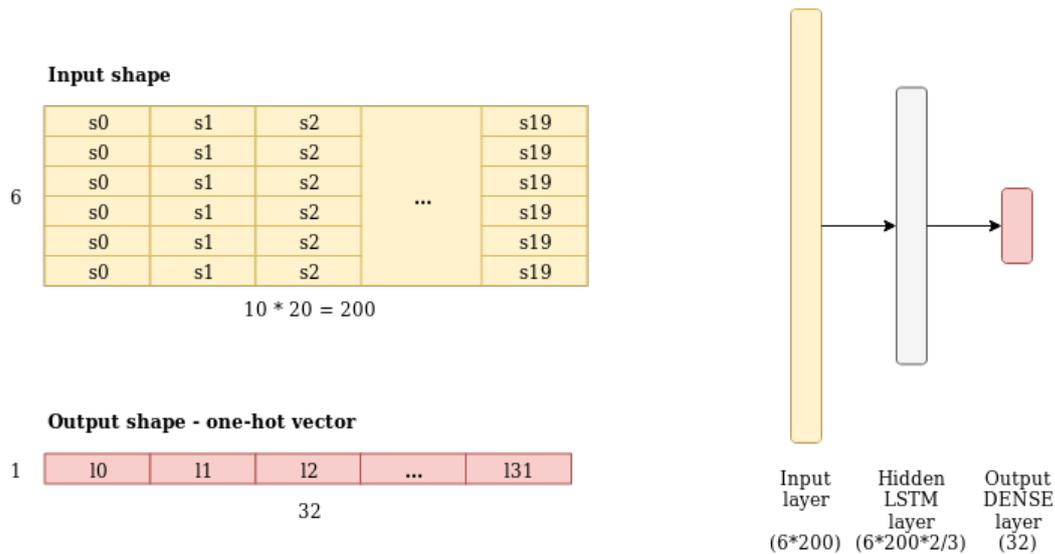


FIGURE 6.6: The input and output shape and a simple representation of the layers of the network.

In the end there is the dense layer that collects the data to generate the output vector. Each cell of this layer is a simple cell connected with every cell of the previous layer (therefore "dense"). Used with the *sigmoid* activation function, it returns a probability for each label, indicating if that is present or not.

The activation function defines the type of output of the node. There are two main types of function:

- *softmax*: returns a probability for each node, but the total sum must be equal to one. It is adopted when the objective of the network is to the single label that with the highest probability (e.g. classification);
- *sigmoid*: returns an individual probability for each node. In effect every single node can have a value between 0 and 1. The total sum isn't constrained, therefore they could even sum up to 0 or to a huge number.

The second function is particularly indicated when an indefinite subset of nodes can be active together (e.g. when tagging a picture). Therefore the reason why I chose the second function: I can select all the applications that generate the network device behaviour given in input to the network.

Two other fundamental properties of a model are:

- the *input shape*: the shape of the data that is passed in input to the model;

- the *output shape*: the shape we want the output to respect.

The input shape is sized $(t * n)$, where t is the number of historic states to consider and n is the number of dimensions of each state. The number of historic states is the number of windows that must be considered (how much back the networks remembers).

The number of dimensions of each state is forced to follow what is defined in the previous stage of the engine. In particular it is $(d * s)$, where d is the number of dimensions for each service s . Those values respectively depend on the number of features of each service behaviour (driven by the extensions used) and the number of services defined in the `ServiceManager`.

In the actual network:

- I have 10 features for each service behaviour, $d=10$;
- I consider 20 different services, including the "unknown" service, $s=20$;
- the significant historical information is retained for 1 minute (6 windows of 10 seconds), $t=6$.

Therefore, the input shape of the network is $(6 * 200)$.

The output shape, instead, is only driven by the number of different labels (l). Those labels represents the device applications that could have generated that specific behaviour.

For the purpose of the demo, I used an extremely restricted labels set of 32 different labels, therefore the output has a size of $l=32$.

Once the model is defined, I need to specify its training parameters. Those parameters are the *optimizer*, the *loss function* and the *metrics* adopted.

The optimizer is that optimization algorithm that can compute the best weights so that the cost function is minimized. Many optimization algorithms have been studied (e.g. Adam, AdaDelta, SDG, Momentum), but one of the best is Adam [19]. So I chose that and, after testing it, I found out that it actually works great with this network.

The loss function is the one in charge of computing the loss of the network. It is then used to optimize the network itself through the calibration of its weights. There are many different loss functions (e.g. mean squared error, binary crossentropy, categorical crossentropy, sparse categorical crossentropy). Of all those, the type of loss function I was interested in is the *binary crossentropy*. In effect, it permits to compute the loss of the output, as long as the final nodes return values between 0 and 1, that it's exactly my case.

The metric I used to compare the output to the labels was the *accuracy*, that counts how many times the predictions match the labels.

6.3.2 The dataset

The right dataset is a key element for the correct training of the network. While in the usual algorithms the data is of secondary importance, in the scope of neural networks its of primary importance, sometimes even more of the model itself (e.g. ImageNet [20]). This should outline its importance and, therefore, sometimes developers spend more time in building the right dataset than the time they spend to model the network.

This led me to study the best dataset to use to train the model. This problem is divided in three:

- defining **the structure** that the dataset should have when processed by the model;
- adapting the dataset to the network input layer (**dataset preprocessing**);
- **collecting the data** in order to build the dataset itself (Section 6.3.2).

The structure of the dataset

As every labeled training dataset, that too is composed by the combination of the features fed to the network, and the labels that the network should return. The structure of the features was anticipated at the end of Section 6.3.1, and corresponds to the output of the previous stage of the engine. It is composed by the concatenation of all the features of a number of services. Given the fact that the number of services and their positions were predefined, it hasn't been necessary to add a dimension for the name of the service. If they weren't already discretized by their positions, they would have been manually discretized: the input of a neuron can't be a variable string, so neither the IP address, hostname or alias could have been used.

The training labels, instead, consist of the concatenation of all the expected labels for that specific network behaviour. The goal of the network is to infer the device behaviour after analyzing a series of network device behaviours. The training labels are nothing but that list of applications that compose the device behaviour.

This means that each record of the training dataset consists of the concatenation of all the features previously specified and a string containing the list

of applications that generated that behaviour. For the simple evaluation it's sufficient to provide to the network only the features. After their evaluation, will be the network itself to return the inferred labels.

In theory the structure is simple and straightforward, but in the practice it gave some problems, as explained in the following sections.

Dataset preprocessing

Given any dataset containing all the required information, it must be normalized before it can be fed to the network. The normalization steps can be easier or more difficult, depending on the condition of the dataset itself.

In my case I had control over the dataset generation (executed during the previous stages of the engine). An example of a record and its header, built by the engine, is the following one:

```

1 nIncomingPackets0,minIncomingLength0,avgIncomingLength0,maxIncomingLength0,totalIncomingLength0,
  ↪ nOutgoingPackets0,minOutgoingLength0,avgOutgoingLength0,maxOutgoingLength0,
  ↪ totalOutgoingLength0, nIncomingPackets1,minIncomingLength1,avgIncomingLength1,
  ↪ maxIncomingLength1,totalIncomingLength1,nOutgoingPackets1,minOutgoingLength1,
  ↪ avgOutgoingLength1,maxOutgoingLength1,totalOutgoingLength1, nIncomingPackets2,
  ↪ minIncomingLength2,avgIncomingLength2,maxIncomingLength2,totalIncomingLength2,
  ↪ nOutgoingPackets2,minOutgoingLength2,avgOutgoingLength2,maxOutgoingLength2,
  ↪ totalOutgoingLength2, ..., nIncomingPackets19,minIncomingLength19,avgIncomingLength19,
  ↪ maxIncomingLength19,totalIncomingLength19,nOutgoingPackets19,minOutgoingLength19,
  ↪ avgOutgoingLength19,maxOutgoingLength19,totalOutgoingLength19, labels
2 2,66,82,99,165,2,66,83,101,167, 0,0,0,0,0,0,0,0,0, 2,46,401,756,802,2,60,261,462,522, ...,
  ↪ 5,88,194,312,972,6,72,81,106,486, "0;91;5;178;100;8"
```

Since I already planned to use a neural network since the design of the general solution, after the third stage we already have some records almost ready to feed the network. But since I use a csv file to communicate between the stages, there are anyway some operations to do:

1. load the dataset from the file where it was written (Section 6.4);
2. parse the features into numerical values (made exception for the labels that are strings);
3. convert the labels string into a **one-hot vector**;
4. **extract random batches** from the dataset, now we have the features to feed the network;
5. define the labels of the batch, now the labels too are ready to be used.

Those steps that seem so easy, absolutely are not. In the first place, I had to load a text file (csv) in Python and parse each line to be an array.

That was one of the simplest tasks of the list. The library `pandas` (<https://pandas.pydata.org/>) has the tools to do exactly that, so I simply integrated it in the application. Therefore, with this library I executed the first step.

The next step consisted in the parsing of all the string features into numerical. To do that I adopted a feature of the library `numpy` (<https://numpy.org/>). With this I created a matrix that represented all the dimensions (features, columns) for each window (lines). Numpy is a powerful library and allowed me to automatically parse the values during the matrix creation. The only limitation of this approach is that there isn't a simple way to validate the type of the parameters. In case the type isn't float, it will trivially throw an exception; for this reason it's important that the previous stage returns the correct values.

```
1 features = np.array(features, dtype=np.float32)
```

The conversion of the labels to a *one-hot vector*, instead, was a bit tricky. A one-hot vector is a sequence of 0 or 1 that represents the presence or not of each label. It is sized as the number of all the possible different labels (the size of the output of the network, Section 6.3.1).

The reason behind the use of the one-hot vector implicitly resides in a property of this kind of networks: in order to return multiple classes (labels), its output must be composed by a set of neurons as big as the possible number of different labels. After the evaluation of each input it will turn on or off each neuron depending on the fact that that label is present or not. Obviously, it's a simplification of how it works, but it's useful to understand the necessity of the one-hot vector: it must match the output of the neural network, so that each neuron has an expected value to compare to.

The function that converts the sparse list of labels (sparse because it contains only the true labels and not the others) iterates over all the possible, known labels and verifies which one is active or not. The code used is the following:

```
1 def labels_to_one_hot(labels_set, all_labels):
2     ground_truth = np.zeros(len(all_labels), dtype=np.int32)
3     if labels_set:
4         idx = 0
5         labels_set = list(map(int, labels_set.split(";")))
6         for label in all_labels:
7             if label in labels_set:
8                 ground_truth[idx] = 1.0
9                 idx += 1
10    return ground_truth
```

The extraction of the batches is required by the nature of the network. In fact, it is a LSTM, therefore it wants in input a set of consecutive states. In my case, those are six consecutive windows of the stream. In order to provide the correct input to the network, I converted the sequence of single windows to the sequence of set of windows (batches). So one batch consists of the sequence of historical information that happened in sequence and that are provided as input to the input layer of the network.

I choose to adopt one of the simplest but more precise version possible of this batch selection: I take all the odd windows. For each one of them, I also select the 5 following ones, so that I have the batch of 6 required windows.

In the last step, I had to define the set of labels that correspond to each batch. The possible options I thought of are:

- collect together all the different labels present on each window of the batch;
- take only the labels of the last window;
- all the intermediate steps (like taking only the labels of the last two windows).

In the end, I decided to adopt the second solution: to take only the labels of the last windows. The reasoning behind this choice resides in the fact that more labels I take, the more the actual device behaviour is affected by the previous windows. All the windows have a size of 10 seconds, meaning that we consider an history of 1 minute. If I consider all the labels of the windows I would infer a device behaviour that is the compositions of all the behaviours occurred in this minute.

The only reason I used a recurrent network is to improve its behaviour recognition, not to bias the actual output. The historical information helps me to differentiate between many similar network behaviours generated by different applications, but must not affect the type of output. In conclusion I don't want the network to detect a behaviour already terminated because it is present in a previous window, so I train the network only with the labels of the last window of each batch.

After all those steps, I finally have the sequence of batches ready to be processed by the network. I also implemented it so that it can build the normalized dataset from a single file or from all the files in a folder (the batches

are concatenated). The last step before the network can be trained consists in the separation of train and test set. To accomplish that I used a functionality of Scikit Learn (<https://scikit-learn.org/>) function: `train_test_split`. It performs a train/test random split based on the proportion defined. In the demo I chose to adopt a test size equals to 1/3 of the dataset size.

Data gathering

Once the whole system was implemented and the neural network built, all was left to do was to find enough data to build the dataset and train the network. After a consistent web research I couldn't find any dataset that I could exploit. Those I found either were too small, or were missing some information or contained only certain types of connections.

Any of those cases was problematic. Obviously if the dataset is too small it can't be used to train a network, mostly if it doesn't have enough information to even build a single batch. If the packets contain only certain information, it means that I could miss the properties that I need either to process the data through the engine or to label it properly (as discussed later on). At last, if the packets are filtered, it means that I'm not able to correctly train the network. In effect the network must learn to label the data in real case scenarios, and not processing a single type of traffic carefully selected. Some examples of those kind of datasets are the followings:

- <https://www.netresec.com/?page=PcapFiles>
- <https://digitalcorpora.org/corpora/scenarios/nitroba-university-harassment-scenario>
- <https://digitalcorpora.org/corpora/network-packet-dumps>

After some time I came to the conclusion that the best, nonetheless fastest, way to gather enough data was to generate the dataset by myself. For this reason I built a device capable of replacing the wireless network in my house, so that I could start sniff my wireless devices in a seamless way (especially given the fact that almost all of them are Android).

Before reaching the best way to accomplish that, I tried many different ways to do that. Unfortunately I got to the right solution the long way round: testing before all the other faulty solutions. The final solution consists of a Raspberry PI (<https://www.raspberrypi.org>) that works as a gateway of an internal network, distributed through an access point. In the meanwhile that it provides the IP addresses to its clients and redirects all the packets (NATing



FIGURE 6.7: The picture of the hardware system installed on my network.

them), it also dumps all the packets that flow through it. This means that the Raspberry was able to generate some files containing a copy of the packets that flowed through it. In order to replace my current networks inside the house in a seamless way, I used an access point that provided the same SSID with the same security protocols of the network I had in place. After completing this configuration, I was able to replace the wireless network without the need to reconfigure all the wireless devices present in the house.

I left the system running for almost 2 weeks, with an average of 4 devices always connected, during which it collected almost 10GB of data. When it was a dataset big enough to start training the network. The problem was that the data weren't labeled. Now that I had the data I ran into another problem: the data wasn't labeled.

This was the only way I had to rapidly collect enough data to train the network, but the packets weren't labeled and I had no idea about which applications generated them. To solve this problem it came in handy a paper that I read some days earlier, while I was searching for a public dataset [21]. In this paper the researchers explain how they collected and labeled enough data for their network.

They adopted an approach similar to mine. First of all they intercepted a huge number of packets, then they used a tool called nDPI [10] to label them. This tool belongs to the family of the *Deep Packet Inspection tools (DPI)*. It is a type of data processing that inspects in detail the content of the packets sent over a computer network. It can take action by blocking or re-routing the

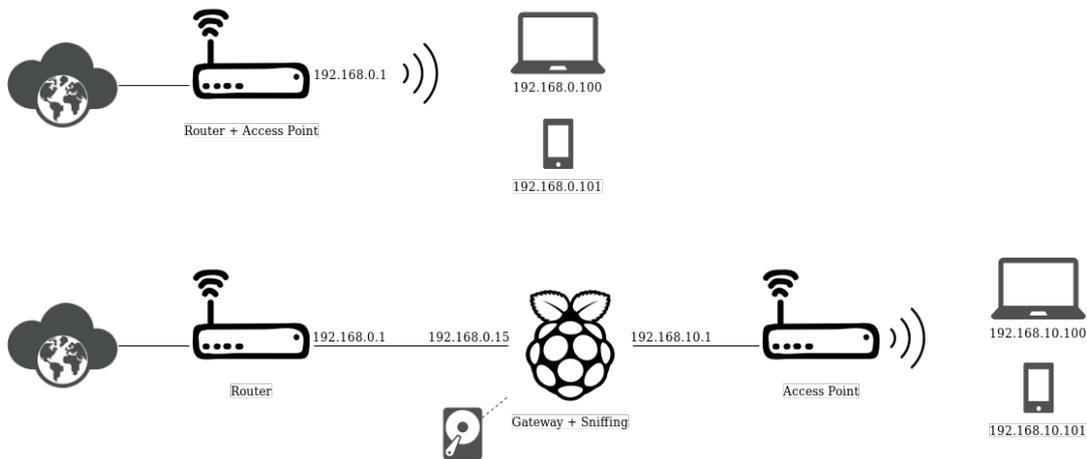


FIGURE 6.8: A representation of the structure of the system I installed on my network. On top the previous infrastructure, on bottom the new infrastructure with the sniffer-in-the-middle.

traffic, or it could simply log it.

While deepening the research, I found out that there are many tools like it (e.g. PACE, OpenDPI, nDPI, ...). In a technical report those tools have also been compared, and it emerged that one of the best and most updated was actually the same nDPI [22]. After those discoveries, I decided to adopt nDPI in my project.

This library is huge and provides a lot of functionalities, but I only needed to label the packets with the alleged high-level application that generated it. Theoretically it is a simple task, but the size of the library and the fact that it is written in C gave me some problems. First of all it was complex to find out how to process a dump file and log every single label, instead of statistics. Then the integration with the main application was extremely difficult. After a long time while I've been trying to correctly make it work with JNI/JNA, the tools that permit an interaction between Java and C, in Gradle, I decided to keep the two programs separated. To make it interact with the main application I decided to use the same filesystem, like between the third and fourth stages. So now the engine will take in input one mandatory parameter: the dump containing the packets, and an optional parameter: the file containing the labels. If the second parameter is provided, the various stages of the engine propagate the information regarding the labels themselves (contained in the respective extensions). Otherwise it simply ignores it, moreover that the labels are only required for the dataset training.

At the end of all that, I was able to sniff the raw packets, label each one of them through an inspection of their content and, at last, to generate the

training dataset by processing the packets and their labels through the engine.

6.4 Applications interaction: the file

Making two different programs interact isn't always an easy task to accomplish. It requires to use some sort of middleware of communication. For example, we could use a web socket, a shared space, or the storage space of the filesystem.

For this application I adopted the third solution: the filesystem. It was the fastest solution to adopt and also permitted a form of logging that was extremely useful in the debug phase. In effect, any stage could make some sort of logging by writing its output to file too, but it isn't always necessary and would be an extremely heavy computation of Input/Output.

Instead, I only chose to log the output of the third stage, so that it could be both displayed (in the fifth stage) or inspected (in the fourth stage). To make it easier to understand the data, I saved it in different files at a different aggregation level and with different structures.

The main file formats that I used were *JSON* (Section 6.1.4) and *CSV* (Section 6.1.5). Every device I analyze on the network has its own set of files. Those files contain different representations of the device behaviour. Obviously each device behaviour is split in its many windows during whose it has been detected. The files that describe the behaviour of each device are:

- a json file that contains the full device behaviour. It is sufficient for any other application to retrieve all the knowledge generated at the third stage. The only reason for which there are other files is to make it easier for the user and other programs to extract certain information without parsing the whole json;
- a csv with the full device behaviour at the maximum detail level. It contains the same information present in the json, but in a tabular format. Each line is a record with the data from a different atomic behaviour, while the information regarding the service behaviour are repeated in a redundant way;
- a csv with partial information regarding the device behaviour at the detail level of the single service behaviour. With one record for each service, it doesn't retain any information regarding the atomic behaviours that compose that service. It's extremely useful to visualize charts without losing ourself with the enormous number of behaviours. It gives an idea to the user of what its happening on the device;

	Accuracy	Loss
Train	98.16%	04.90%
Test	97.93%	05.67%

TABLE 6.1: The performances of the network at the 15th epoch.

- a csv that contains the information required by the fourth stage: the *Neural Network*. Each record is already discretized and structured for the NN. It is almost impossible for the user to understand those data, given the fact that those are just a series of numbers. In order to prepare the data for the NN, the file also contains the empty windows (filled with the default values) and the rows are formatted so that they contain all the services. If a service didn't occurred in a certain windows, that one is compiled with the default values (zeros).

Obviously there could be a lot more files as could just be one file containing all the information of the behaviour. The reason why I used those files is to facilitate both the handover to the fourth stage and the data visualization for the user during the debug.

6.5 Results

After two weeks of data gathering, the time had finally come for me to test the effectiveness of the engine. I used the scripts to run the data through the first 3 stages. This operation required approximately 12 hours. From this execution I obtained the set of files containing the dataset ready for being used to train the network. The total occupied space was of under 100 megabytes.

To train the network I used a machine with a *Intel(R) Xeon(R) CPU E5-2640* with 32 CPUs running at 2.00GHz. Once the dataset were loaded on the machine, I launched the network training. As shown in the Figure 6.9, after just 10 epochs we had an accuracy of 98.09% on the train set and 97.9% on the test set; the loss was just of 5% for both the train and the test set.

Overall, the best results were obtained in the 15 epoch. At that point the performances were of an average accuracy of 98.00% and average loss of 5.25%. From that epoch, the network encountered a famous problem in neural networks: it started overfitting the training set. This means that, while the train set was ever increasing its performance, the test set was performing badly, much worse than at the 15th epoch. For more details refer to the Table 6.1.

Given the little information I gave to the network (under 100 megabytes regarding the traffic of multiple devices for 2 weeks), I considered those data

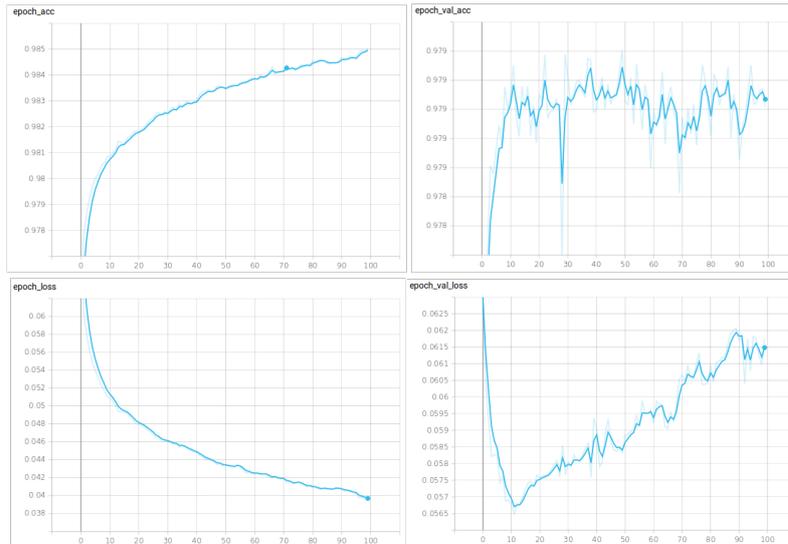


FIGURE 6.9: This picture represents the improvements of the network in accuracy and loss. On the left we can see the values applied to the train set, on the right those applied to the test set.

as great. This was the first proof that the model could work, but I still had to make sure that the network was really inferring some significant information from the data, and that it wasn't just matching meaningless labels (e.g. Unknown). To perform this further check, I modified the application to be able to process real time information and provide an immediate feedback to the user: Section 6.6.

6.6 Real-time testing: the live version

The realization of the live version consisted of three essential steps:

- build a sniffer that captures the live traffic from a network interface instead of loading a dump file;
- make the python application able to be notified of any change to the input file (modified while the application is running);
- load a trained network and use it to evaluate the input windows.

To complete the first step, I basically modified the `Sniffer` so that it can be both initialized with a dump file, and it uses its content, or with an interface of the device, and it uses the live traffic flowing through it. The effort required was more than the one needed to implement the version with the dump files, but in the end I was able to make it work with both the configurations.

In order to simplify the debug, I gave the sniffer the ability to dump all the packets it intercepts into a dump file. This allows me to reprocess all the packets later on, so that I can double check the result or tune the engine.

Once the sniffer was implemented, the model and the good design of the application proved themselves again: all the following stages were already ready to process live traffic. In effect, for those stages the data flow of live traffic isn't any different from the flow generated from the live traffic. Without any other application updates it already generated some output files that it kept updated as the stream proceeds.

However, the Python application required to be redesigned in order to finish the second and third steps. It was designed to perform bulk processing on large datasets and use them to train a network. Now I want to use the aforesaid trained network to evaluate the live windows.

To do that, I integrated in the application a new tool: *Watchdog* (<https://pypi.org/project/watchdog/>). It observes a folder and notifies me every time a file is created, deleted or modified. Thanks to this library I was able to call a function every time the file is updated.

In this function, I take the last 6 records (the last window and its 5 previous ones) and evaluate them trough the network. In order to use an already trained network, I had also to find a way to save it in a file. Luckily, Tensorflow itself provides a functionality to do that. With a specific callback, we are able to save all the parameters of the network in a file after each training epoch.

6.7 The interface

For the program I planned two possible usage scenarios:

- via GUI (Graphical User Interface);
- via command line.

The first scenario is useful when the user just want to manually execute the program, for example a final user or myself during the first stage of the testing. The second scenario is designed for those who want to embed the program in a script, run it remotely or execute it periodically. It takes a set of arguments that specify the task to perform and directly process the data without any kind of graphical interface. The only exception is the the console logging of the application.

6.7.1 Graphical User Interface

For the graphical interface, I decided to use **JavaFX** (<https://openjfx.io/>). It is a powerful open-source library to develop graphical user interfaces in Java.

I designed 3 main application windows:

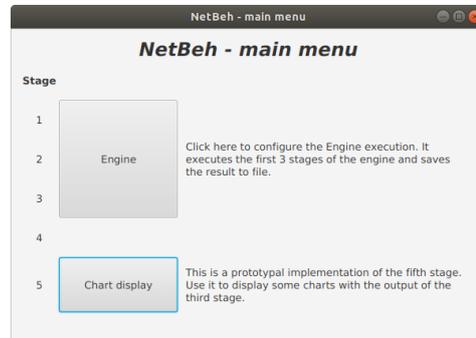
- The *main application window*: it's the first window displayed when the user executes the application. It guides the user in the selection of the task he wants to accomplish. Actually there are 2 possibilities: either the user starts the engine, or he can open an example of the implementation of the fifth stage;
- the *extraction window*: this window allows the user to configure the engine and launch its computation. There are 2 main possibilities: either the user wants to start the engine with a dump file as source, or he wants to perform a real-time analysis. Obviously, the user can configure all the parameters of each one of those configurations;
- the *analysis window*: this window is a prototype of a version of the fifth stage that uses the results of the third stage, other than the fourth. In this case, it uses exclusively the output of the third stage to display some charts regarding the traffic. It could be extremely useful to the user to understand what's happening on the device and to identify some simple patterns. More details about this interface can be found at Section 7.1.

6.7.2 Command line

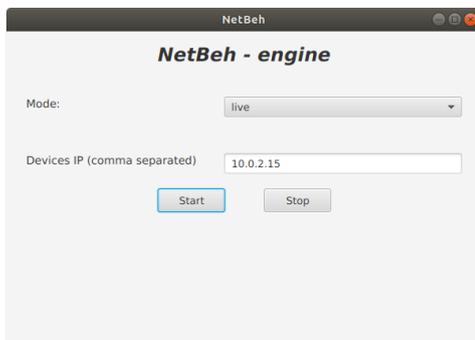
This kind of execution is perfect for scripting and automatic executions. I designed it so that it would be easier to run the program automatically on a series of files or to provide it the default values for execution.

For the Scala application, the arguments it supports are:

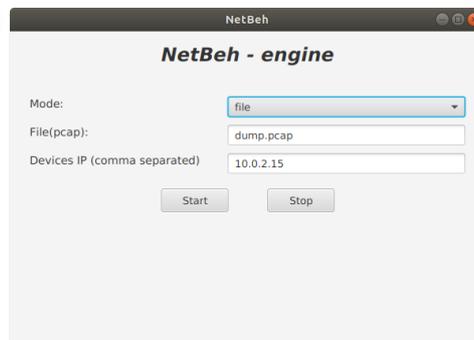
- `-no-gui`: disables the graphical interface;
- `-i $path$`: specifies the input path of the pcap file;
- `-d $device$`: specifies the index of the device that must be used for the live sniffing;
- `-ip $address$`: a list of comma separated IP address of the devices we want to analyze in the network traffic.



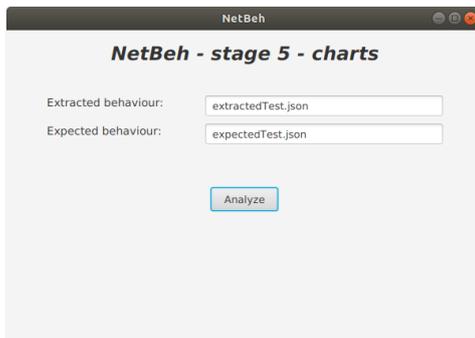
(A)



(B)



(C)



(D)



(E)

FIGURE 6.10: Some screenshots of the application GUI.

- (a) the main application window
- (b) the extraction window live and (c) the extraction window from dump files
- (d) the analysis window
- (e) some analysis charts

The first parameters is used when you don't want the graphical interface to be shown, but, instead,

The first parameter is used when you want the processing to immediately start with a specific configuration (provided through the arguments), without the need to interact with a graphical user interface. In this case, the GUI isn't either shown at all. If it is present, then either the path or the device must be specified (respectively depending on whether you want to use a dump file or a live device sniffing), together with the IP address.

In case the GUI is displayed, those parameter are all optional. They are only used to fill the input fields with the default values.

For what matters the Python application, all the arguments are mandatory. There are 2 possible version of the application to execute:

- the training program, that uses some dumps to train the network;
- the live program, that keep observing a file to detect new live updates.

The first one, `Application.py`, only requires the definition of the folder where it can find all the dumps and the destination where it can save the model of the network. The second one, `Application_live.py`, similarly requires the user to define what is the file to observe and where is the network model he can load. Both applications, once those parameters are correctly defined, run seamlessly. They don't require any kind of interaction with the user except the application termination.

In order to improve the usability of the whole system, I also wrote some scripts that make use of those parameters, as explained in Section 6.8.

6.8 Some scripts

I created some scripts to simplify and automate the system execution. In particular, I created two scripts to automatically process all the *pcap* files in a folder. Their creation was driven by the necessity of creating a consistent dataset for the training of the neural network at the fourth stage (Section 6.3). For this reason, the primary goal of the scripts was to process the raw packets through the first 3 stages and to pile them up, ready to be used to train the network. In practice, it should just simplify the dataset creation. Despite that, the scripts can be slightly modified to perform bulk analysis of some network traffic, even through the following stages. The scripts are:

- `dataset_generation.sh`: given the pcap path, the IPs to analyze and the destination of the dataset, it computes and saves the dataset, ready for NN, in the destination;
- `dataset_generation_recursive.sh`: given the pcap folder path, the IPs to analyze and the destination path, it elaborates all the pcap files and saves all the dataset, ready for NN, in the destination path.

In practice, the second script calls the first one to process all the pcap files in the folder, so the main complexity resides in that one.

The first script has to perform 3 main actions:

1. generate all the labels of the traffic from the pcap file;
2. generate the dataset files using the pcap and the labels;
3. move the file ready for the NN to the destination path.

The first step is accomplished by executing the *C* program responsible of the labeling. It generates a file that, together with the other arguments, is provided to the Scala application. By executing this program it completes the second step and obtains all the output files. The last step is to take the neural-network-ready file and to move it in the destination folder for the datasets.

At last, I also wrote a script to automate the execution of the version in real time (`live_run.sh`). It takes as parameters:

1. the path to the configuration of the trained network;
2. the IP to watch;
3. the number of the interface where the traffic is observed;
4. the path to the Java runtime.

In order to make the script work, it must be executed by an user with administration privileges (e.g. using `sudo` or providing the specific Linux Capabilities). The reason resides in the fact that the live sniffing requires special permission. This kind of permissions can only be granted by the administrator of the machine, therefore the necessity to execute it with a privileged account.

This script configures the system and executes all the application needed to build a full engine. Then it starts sniffing the interface and processes the raw packets trough the engine, until it is provided to the user the inferred device behaviour.

Chapter 7

Knowledge applications

I designed a special stage in charge of using all the information gathered: the fifth stage. It is the one in charge of using the information retrieved to perform some specific operations, that can vary from the simple data display, to the more complex anomaly detection.

Obviously, in order to test the prototype, I had to realize a basic version of the fifth stage that displays a minimal amount of knowledge. This is deepened in Section 7.1.

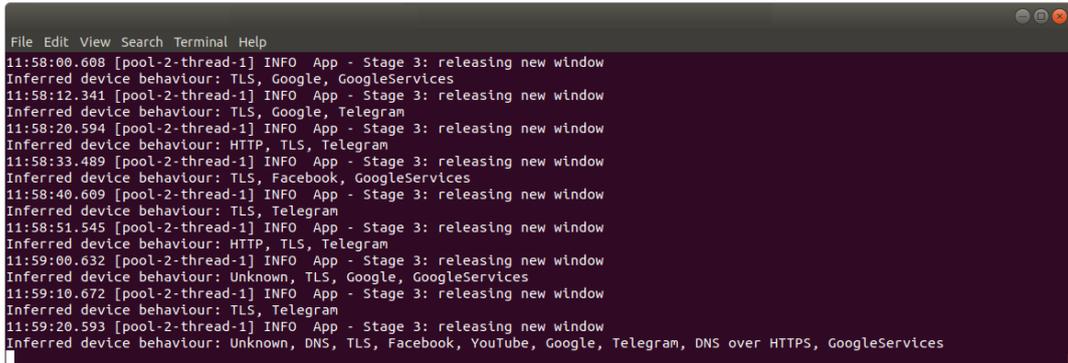
Other than that basic necessity, I also modelled a more complex version of this stage. The goal of this design is to prove that the fifth stage is actually one of the more ductile stages in the whole engine. It can take all the information provided, and use them anyway he wants. Therefore, other than displaying the results of the previous stages, I designed a possible solution to detect anomalies in the device behaviour. Its purpose is to "encapsulate" the device behaviours so that any anomaly triggers an alarm. This version of the stage is detailed in Section 7.2

7.1 The basic interface

A simple data display can be performed in many ways and can show different things. The simplest version of the interface displays the device behaviour inferred at each window. In order to prove the possibility to use the information generated at the third stage too, I made another interface that displays some charts.

The first interface, the one who displays the device behaviour detected, is a simple console output. The purpose of an interface like that is mainly limited to the debugging and validation of the model. In some real-case scenarios, nobody would ever want to spend his days in front of the display in order to see all the occurring behaviours. It would also have problems with the scalability of the

system (e.g. many devices at once). A possible output of this interface can be found in Figure 7.1.

A terminal window with a dark background and light text. The window title bar shows 'File Edit View Search Terminal Help'. The output consists of multiple lines of log messages. Each line starts with a timestamp, followed by a thread identifier, a log level, and a message. The messages are grouped into pairs: the first line of each pair is an 'INFO' message about releasing a new window, and the second line is an 'Inferred device behaviour' message listing various services and protocols. The inferred behaviours include combinations of TLS, Google, GoogleServices, Telegram, HTTP, Facebook, and YouTube, along with DNS and DNS over HTTPS.

```
File Edit View Search Terminal Help
11:58:00.608 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: TLS, Google, GoogleServices
11:58:12.341 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: TLS, Google, Telegram
11:58:20.594 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: HTTP, TLS, Telegram
11:58:33.489 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: TLS, Facebook, GoogleServices
11:58:40.609 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: TLS, Telegram
11:58:51.545 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: HTTP, TLS, Telegram
11:59:00.632 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: Unknown, TLS, Google, GoogleServices
11:59:10.672 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: TLS, Telegram
11:59:20.593 [pool-2-thread-1] INFO App - Stage 3: releasing new window
Inferred device behaviour: Unknown, DNS, TLS, Facebook, YouTube, Google, Telegram, DNS over HTTPS, GoogleServices
```

FIGURE 7.1: This is an example of the output from the fifth stage when executing the live application.

The fifth stage isn't limited to display only the data flowing out from the fourth stage. In fact, it can see all the output generated by the third stage as well. To prove that feature, I realized the second interface. It takes the output of the third stage (the networking device behaviour) and displays it through some charts. This can provide another level of awareness to the user, because with that the user can address the reasons why the system says that a certain device is having a specific behaviour. It can also be used by the user to manually identify some simple patterns. A visualization of this kind of interface can be found in Figure 7.2.

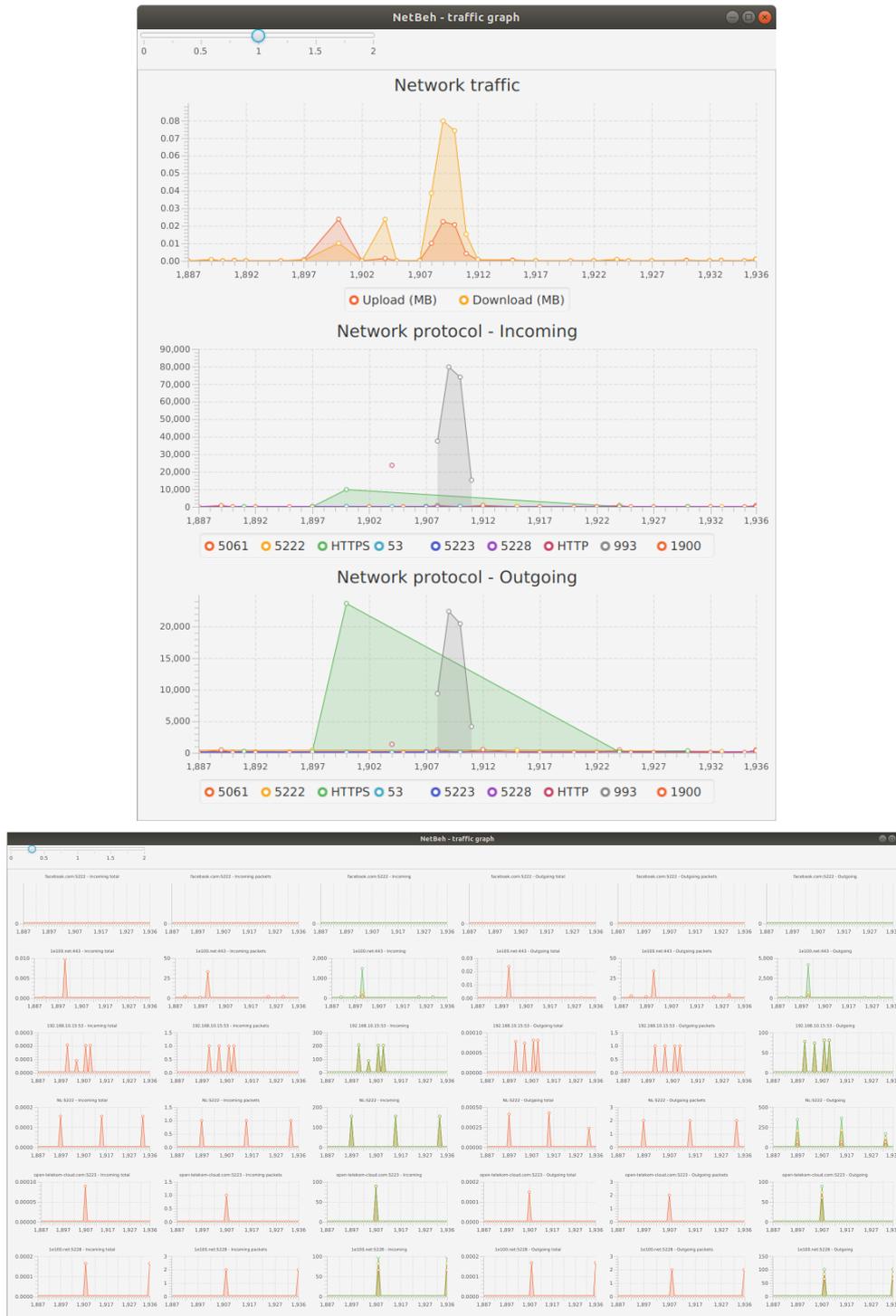


FIGURE 7.2: This is an example of what charts are displayed when observing a network device behaviour. On top some global stats regarding the global traffic; on bottom each line of charts is relative to a single service and details all its different features.

7.2 A behaviour encapsulation to detect anomalies

As mentioned above, those discussed in Section 7.1 are interfaces mostly useful during the debug (and in the validation process of the model). Outside of that, in some real-case scenarios, the administrator of a system could use this model to detect the behaviour of the devices inside its network. This detection could then be combined with a set of rules that check that the behaviour of those devices respects, for example, the company policies.

Therefore, I believe that one of the most useful, and possibly required, functionalities could be a tool to detect anomalies. Despite that, it is a different feature than the common anomaly detection tools. Common anomaly detection tools are trained to detect only a certain type of anomalies, and usually on a specific kind of traffic. They can be easily adopted server-side, but it's often difficult to detect anomalies in the traffic of a device like a smartphone: it is too unpredictable.

For this reason I designed this detection in a similar way to tools like *AppArmor* (<https://wiki.ubuntu.com/AppArmor>). In practice, this kind of tools build a "cage" around an application to protect the system. At the same way, we can build a cage around the device behaviour to protect the network and, at the same time, possibly detect the presence of malware. Anyway, it's important to point out that the purpose of this tool is not to detect malware, but anomalies of any kind.

7.2.1 The design

In order to encapsulate the behaviours, I designed a structure that allows the definition of the encapsulating "cages". It is called *expected device behaviour* and permits the definition of the boundaries of a single detected device behaviour. How it works, is that it defines some bounds for certain features of the behaviour. For example, we could specify that a certain behaviour (e.g. all the communications with a certain service) can never exceed a certain network throughput in upload or download. Other than those boundaries, the expected behaviour is also composed of a set of rules. Those rules are required to define other controls, possibly more strict or simply different, and to specify how the boundaries must be respected (e.g. percentile tolerance).

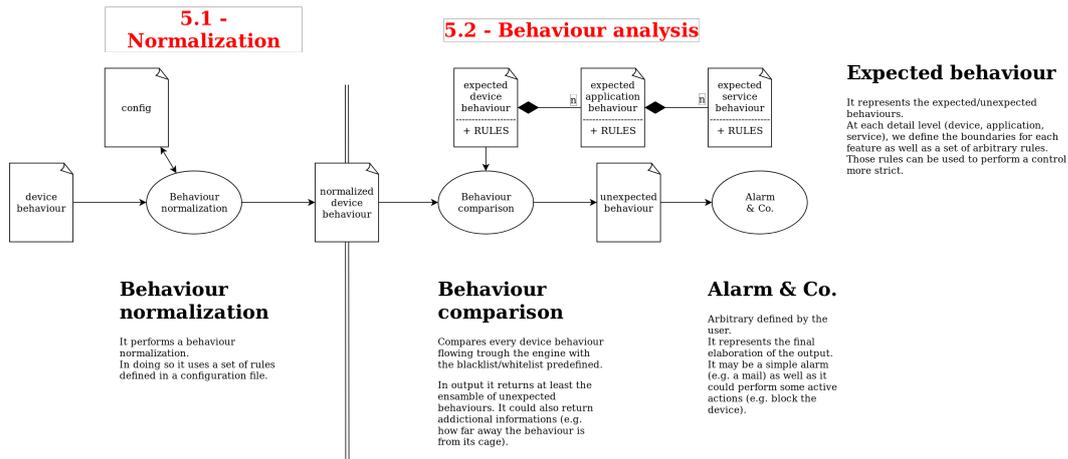


FIGURE 7.3: This image represents the structure of the behaviour encapsulation. On the left the device behaviour detected, on the right the structure of the cages for the devices behaviours (expected device behaviour) and their comparison (behaviour comparison).

The structure of the device behaviour, anyway, is complex and composed by many applications that generates many service behaviours. For this reason I designed the *expected device behaviour* to map this same structure.

For each device behaviour, we can have many applications. Each *expected application* is structured similarly to its parent: it can match a set of features of the application and has a set of specific rules. At the same way, the application behaviour is composed by many *expected service behaviour*. In effect, each application can communicate with different services, generating different service behaviours on the network. Again, every *expected service behaviour* defines the boundaries for its features and can make use of a set of specific rules.

The reason for which I introduced the concept of *application behaviour*, that doesn't exist in the detected *device behaviour*, is to simplify the management of those cages. That way, the user can define a set of applications (e.g. Instagram or Browser) and each one of them contains the corresponding services (e.g. Instagram communicates in a specific way with both Facebook and Instagram). Then, many different applications can include the same service (e.g. Instagram includes Facebook as well as Facebook does) with different boundaries that must be merged together.

This means that it can be defined some generic profile for each application (the cage). This would be a standard profile, that could even be distributed through an official online database. This would allow each user to download and activate it for its device, without the necessity to build it manually.

At the end, the principle is that each device behaviour detected is compared

with the whitelist or blacklist of expected behaviours. Then, eventually, those can trigger an alarm when the device is exceeding the boundaries defined.

7.2.2 Implementation in the demo

When I designed the application, I also designed those comparison tools and, during the implementation, I implemented the basic structures. At the end I didn't end up using them fifth stage, but the basic structures are ready to be adopted and configured.

To generalize the structure, I adopted the concept of *expected behaviour* and *complex expected behaviour*. The first one defines the boundaries and rules for a generic behaviour in a specific time-window. The second one defines some more complex rules about the possible sequences of behaviours in a sequence of windows. This is a generic structure applicable to any layer (device, application or service), but I didn't designed the single specific versions.

In the implementation, the `ExpectedComplexBehaviour` only contains a sequence of `ExpectedPatternWindowBehaviour`. Each one of those objects basically is a pattern o behaviours. It specifies the boundaries that a sequence of behaviours should respect and the additional rules. An example of a possible rule could be the type of comparison that should be done between the sequence of detected behaviours and this sequence (e.g. ordered or random, consecutive or sparse, etc). To define the boundaries of each single window behaviour, it uses the `ExpectedBehaviour`.

I also provided a basic implementation of the `InternetExpectedServiceBehaviour` and of the `InternetExpectedServiceWindowBehaviour`, that actually compare some Internet behaviours with the specified pattern.

At the end, we are able to compare any sequence of behaviours, occurring in time, with the specified structure. With this implementation it is possible to perform a low-level comparison. By removing the concept of application, we can define the cages for specific sequences of service behaviours. Anyway, it isn't ready to be adopted out-of-the-box yet. It will need some more tuning and the definition of some rules and profiles, so that it can be actually tested.

Chapter 8

Conclusions

The purpose of the thesis was to design a model for the recognition of the behaviour of networked devices. Without installing any application on the device itself, or even modifying the network structure, it must only use the network traffic to obtain the required information. Due to the spread of cryptographic techniques, the challenge of the thesis was to do it without even using the payload of the packets.

This led me to discard all the previous researches, that all broke one of the constraints. Instead, I designed a model that was able to only use the network traffic metadata to identify the networking behaviour of the devices, and then to infer the device behaviour itself.

In order to prove the validity of the model, I built a prototype application. The application implements a basic version of the model, but already provides promising results. This prototype was built and trained to detect the behaviour of Android devices, but it's important to note that it was designed in an extremely modular way. With a slight modification it would be able to detect the behaviour of any kind of networked device, not only Android ones, or even using other protocols (e.g. Bluetooth).

As previously said, the results are extremely promising. The prototype was able to infer the right behaviour with an extremely high accuracy (about 98%). Given the little information it retains from the network traffic (just some aggregated data regarding the metadata of the packets), it is an astonishing result. The model proved to be extremely effective and the efficient. The bigger risk of the research was that the metadata may not be enough to infer the device behaviour. With the prototype I was able to prove that they already retain an incredible amount of information, and can be used to infer significant knowledge.

With the designed model, I was even able to even perform a live analysis

of the network traffic. That proved the ultimate effectiveness of the model, since it was able to detect the custom behaviours that I was producing live on purpose. In fact, I was worried that the prototype could mistake similar network behaviours. This didn't occurred, showing that retaining 1 whole minute of data is enough to consistently infer the correct behaviour.

This is an extremely promising work, that could enable new techniques in the traffic analysis for both protecting the network and the user itself. This is an earlier stage for this type of analysis, so there are many possible developments. First of all the prototype could be improved. In fact, it is just a simplified prototype and only uses the basic properties of the Internet packets. It even manually extracts from them the networking device behaviour. This could be replaced with a neural network properly trained (e.g. CNN). It would also be possible to use many more features to refine the result, like the duration of a communication session, the type of established connection and of the client.

Another important characteristic of the model is its polymorphism. In fact, it is not only able to detect the Internet traffic at the transport layer, but it could also be used to infer the behaviour of other layers (e.g. encrypted WiFi packets) or other protocols (e.g. Bluetooth). It wasn't deepened in this research, due to obvious reasons of focus and of time, but it is as much powerful as the Internet analysis.

At last, but not least, the fifth stage can be considerably improved. I only provided a demonstration of its potential, but it can be replaced with most functional interfaces, especially if it is adopted in the industrial environment. Right now the inferred behaviour is composed by a simple list of application/protocols; it can be improved with information like the weight of each one of them, its probability and other similar properties.

The use cases of this work are many and various. It was thought to support a network administrator to gain a better understanding of the behaviour of the devices in its network. This new knowledge helps him both protect the organization from a malicious user as well as protect the user itself. In effect, the features of this new tool can be used to detect the presence of anomalies, like malware installed on the device. At the moment, there is the necessity of providing better tools for Operational Technology (OT) security [23], given the rising Industry 4.0 [24]. This is as much of interest to the company as much

to the oblivious user that is informed of the risk he is taking (Bring-your-own-device (BYOD) policies [25]).

The last important feature of this work is its wide flexibility. It works if installed on a network router, but it simply uses the header of the packets. This means that if it is meant to be used to supervise the device in a WiFi, it can even be installed on a custom device in that same network. Just observing the wireless traffic, it would be able to detect the behaviours of all the other networked devices (easy-to-install).

At last, I would like to note that the aim of this work is to detect the device behaviour. Even if the global behaviour is the same, different users could behave slightly differently. In the future, this slight difference could be exploited to make a fingerprinting of the user using a device, therefore recognizing him as soon as he enters the network. For example, this could be used to make a local authentication of the user or to detect if its device is being used by someone else. So I chose to focus the work on Android devices, conscious that in the future it can be exploited on every possible networked device.

Bibliography

- [1] K. Kannan and R. Telang, “An economic analysis of market for software vulnerabilities”, Apr. 2004.
- [2] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff, “A brief history of the internet”, *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 5, pp. 22–31, Oct. 2009, ISSN: 0146-4833. DOI: 10.1145/1629607.1629613. [Online]. Available: <http://doi.acm.org/10.1145/1629607.1629613>.
- [3] E. H. Spafford, “The internet worm incident”, in *ESEC '89*, C. Ghezzi and J. A. McDermid, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 446–468, ISBN: 978-3-540-46723-6.
- [4] C. Cimpanu, “For two hours, a large chunk of european mobile traffic was rerouted through china”, *ZD Net*, Jun. 7, 2019. [Online]. Available: <https://www.zdnet.com/article/for-two-hours-a-large-chunk-of-european-mobile-traffic-was-rerouted-through-china/> (visited on 10/23/2019).
- [5] R. Sobers, “Data breach response times: Trends and tips”, *ZD Net*, Mar. 13, 2019. [Online]. Available: <https://www.varonis.com/blog/data-breach-response-times/> (visited on 10/23/2019).
- [6] P. Velan, M. Čermák, P. Čeleda, and M. Drašar, “A survey of methods for encrypted traffic classification and analysis”, *International Journal of Network Management*, vol. 25, no. 5, pp. 355–374, 2015. DOI: 10.1002/nem.1901. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/nem.1901>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.1901>.
- [7] S. LLC, *GlassWire*, <https://www.glasswire.com/>, [Online],
- [8] R. Zalenski, “Firewall technologies”, *IEEE Potentials*, vol. 21, no. 1, pp. 24–29, Feb. 2002, ISSN: 1558-1772. DOI: 10.1109/45.985324.

- [9] T.-Y. Kim and S.-B. Cho, “Web traffic anomaly detection using c-lstm neural networks”, *Expert Systems with Applications*, vol. 106, pp. 66–76, 2018, ISSN: 0957-4174. DOI: <https://doi.org/10.1016/j.eswa.2018.04.004>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0957417418302288>.
- [10] ntop, *nDPI*, <https://www.ntop.org/products/deep-packet-inspection/ndpi/>, [Online],
- [11] A. Mohaisen, O. Alrawi, J. Park, J. Kim, D. Nyang, and M. Mohaisen, *Network-based Analysis and Classification of Malware using Behavioral Artifacts Ordering*, <https://arxiv.org/pdf/1901.01185.pdf>, [Online], 2019.
- [12] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, *Mobile malware detection through analysis of deviations in application network behavior*, <https://cyber.bgu.ac.il/wp-content/uploads/2017/10/1-s2.0-S0167404814000285-main.pdf>, [Online], 2014.
- [13] H. Aria, *Android Malware Detection Using Network Behavior Analysis And Machine Learning Classifiers*, <https://csec.it/MSTheses/Aria.pdf>, [Online], 2017.
- [14] H. P. Barge and P. R. Chandre, “Malware detection in mobile through analysis of application network behavior by web application”, 2016.
- [15] R. Jin and B. Wang, “Malware detection for mobile devices using software-defined networking”, in *2013 Second GENI Research and Educational Experiment Workshop*, Mar. 2013, pp. 81–88. DOI: 10.1109/GREE.2013.24.
- [16] D. Bekerman, B. Shapira, L. Rokach, and A. Bar, “Unknown malware detection using network traffic classification”, in *2015 IEEE Conference on Communications and Network Security (CNS)*, Sep. 2015, pp. 134–142. DOI: 10.1109/CNS.2015.7346821.
- [17] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret, “Network traffic classifier with convolutional and recurrent neural networks for internet of things”, *IEEE Access*, vol. PP, pp. 1–1, Sep. 2017. DOI: 10.1109/ACCESS.2017.2747560.
- [18] S. Rezaei and X. Liu, “Deep learning for encrypted traffic classification: An overview”, *IEEE Communications Magazine*, vol. 57, no. 5, pp. 76–81, May 2019. DOI: 10.1109/MCOM.2019.1800819.

- [19] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. arXiv: 1412.6980 [cs.LG].
- [20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “Imagenet large scale visual recognition challenge”, *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015, ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. [Online]. Available: <https://doi.org/10.1007/s11263-015-0816-y>.
- [21] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret, “Network traffic classifier with convolutional and recurrent neural networks for internet of things”, *IEEE Access*, vol. PP, pp. 1–1, Sep. 2017. DOI: 10.1109/ACCESS.2017.2747560.
- [22] T. Bujlow, V. Carela-Español, and P. Barlet-Ros, “Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification”, Universitat Politècnica de Catalunya, Tech. Rep., Jun. 2013.
- [23] R. Piggin, “Industrial systems: Cyber-security’s new battlefield [information technology operational technology]”, *Engineering Technology*, vol. 9, no. 8, pp. 70–74, Oct. 2014, ISSN: 1750-9637. DOI: 10.1049/et.2014.0810.
- [24] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, “Industry 4.0”, *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, Aug. 2014, ISSN: 1867-0202. DOI: 10.1007/s12599-014-0334-4. [Online]. Available: <https://doi.org/10.1007/s12599-014-0334-4>.
- [25] Y. Wang, J. Wei, and K. Vangury, “Bring your own device security issues and challenges”, in *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, Jan. 2014, pp. 80–85. DOI: 10.1109/CCNC.2014.6866552.