Rapport de Thèse Professionnelle

Filippo Anarratone
Décembre 2017

MANTIS: a Machine learning Approach to Network Traffic Inspection for Security

Société: Lastline

Encadrant dans l'entreprise: Ingénieur Corrado Leita

Encadrant académique: Professeur Davide Balzarotti

Thèse NON Confidentielle

Filière d'attachement:
Sécurité des systèmes de comunications

POLITECNICO DI TORINO, EURECOM - TELECOM PARISTECH

MANTIS: a Machine learning Approach to Network Traffic Inspection for Security

by

Filippo Anarratone

Industrial supervisors:
P.h.D. Corrado Leita,
P.h.D. Marco Cova

Academic supervisors:
Professor Davide Balzarotti
Professor Maurizio Matteo Munafò

Master of Science Degree
in
Computer Engineering

January 2018

"I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."

Alan Turing

POLITECNICO DI TORINO, EURECOM - TELECOM PARISTECH

Abstract

Intrusion Detection Systems and Intrusion Prevention Systems have been for a long time an essential component of network security, especially in enterprise environments, where a data breach can be catastrophic. Unfortunately, breaches are still happening despite the efforts on the defenders side, hence the need of **Breach Detection Systems**.

Once a breach has occurred, it is important to be able to react as soon as possible to investigate the incident, in order to be able to avoid further compromises. This task is still considered as an art by many, where it is the duty of a skilled *incident responder* to produce a report of the breach and to suggest mitigation and recovery actions. A Breach Detection System is an automated engine that should be able to detect malicious activities once an entity in the network has been compromised, but at the same time it should collect enough data for the incident responder to effectively investigate the breach.

With **MANTIS**, we present a possible approach to the development of part of a BDS, exploiting machine learning techniques to categorize hosts in a network based on their behaviour, while identifying singular hosts and tracking the *behavioural dynamics* of each entity. We choose an *unsupervised* training model that works on aggregated *Netflow* data. The use of Netflows is justified by the enormous advantages in terms of scalability that this format provides, while keeping enough information to effectively categorize traffic. We consider as features the *distributions* of different Netflow fields, and effectively apply this system on real data captured on different networks.

Our conclusion is that this approach can give very good insights about what is happening in a large network, being able to identify suspicious behaviours and giving the incident response team the ability to effectively investigate a breach.

POLITECNICO DI TORINO, EURECOM - TELECOM PARISTECH

Abstract (french version)

Les Systèmes de Détection d'Intrusion et les Systèmes de Prévention d'Intrusion ont été essentiel pendant longtemps pour la sécurité des réseaux. Malhureusement, on observe toujours des violations des données, donc il pousse la nécessité des **Systèmes de Détection des Violations des Données** (BDS).

Après une violation des données, il faut réagir tout de suite pour éviter autres dangers. Ce tâche est très complexe pour le *répondeur d'incident*, et un Système de Détection des Violations des Données doit aider dans cette direction.

Avec **MANTIS**, on va suggérer une possible solution pour le développement d'un BDS, avec l'utilisation de techniques de machine learning pour catégorizer les ordinateurs entre plusiers *groupes de comportement*.

On a choisi un modèle *unsipervised* qui marche avec *Netflows* et on a appliqué cette système avec des données real arrivant de different reseaux.

Nôtre conclusion est que est possible utiliser cette système pendant l'investigation d'une violation pour avoir une vue mellieure dans la situation du réseau.

Acknowledgements

The last year has been full of new experiences for me, and I would like to thank all the people that made it possible. Starting from EURECOM staff and Professors, with a special mention to Professor Balzarotti for helping me find this internship. A big thanks also to all the people I met at EURECOM, for making me enjoy the best academic year of my entire career.

Moving to the amazing people I found in London, that helped me grow also as a person: thanks to Alessandro, Andrea, Claudio, Corrado (the youngest), Enrico, Giancarlo, Luukas, Maurizio, Raphael and Stefano for being such amazing colleagues. A special thanks goes of course to Corrado (the eldest) and Marco, for following me regularly, for the amazing suggestions and also for the *"get well soon"* that more than once the circumstances brought you to tell me. A thanks also to Lastline as a Company, for being a fantastic place to improve myself and to challenge the boundaries of my current knowledge.

Thanks to Professor Munafò, for following my activities from Italy despite the lack of possibilities for us to meet in person. Thanks to Professor Balzarotti also for being available to advise me as supervisor at EURECOM.

Ultimi ma non ultimi, un grazie speciale alla mia famiglia ed ai miei amici: per essere riusciti a starmi vicino, anche se distanti fisicamente, per tutto questo tempo. Grazie per avermi supportato, incoraggiato, per i preziosi consigli (e anche quelli un po' più scontati, ad esempio *"non farti rubare la carta di credito"*), le telefonate, i messaggi e per avermi fatto sentire a casa.

Contents

List of Figures

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BDS	**B**reach **D**etection **S**ystem
DBSCAN	**D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise
DNS	**D**omain **N**ame **S**ystem
ICMP	**I**nternet **C**ontrol **M**essage **P**rotocol
IDS	**I**ntrusion **D**etection **S**ystem
IETF	**I**nternet **E**ngineering **T**ask-**F**orce
IP	**I**nternet **P**rotocol
PCR	**P**roducer **C**onsumer **R**atio
SVM	**S**upport **V**ector **M**achine
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol

Chapter 1

Introduction and motivation

1.1 Breaches and Breach Detection Systems

In the systems security industry, the term *data breach* identifies a security incident that involves any unauthorized access to protected data. Breaches represent a serious threat to an organization, especially nowadays, as companies start collecting more and more data for analysis.

Figure 1.1 shows the summary of the Verizon Data Breach Investigation Report 2017[1]. From the picture, we can see that 62% of the breaches included some sort of *hacking*, meaning actions performed (either from the outside or from the inside of a corporate network) to exploit security holes and access data in an unauthorized manner. Moreover, 75% of the breaches were perpetrated from the outside, meaning that no one inside the organization helped get access to the data.

Disclosure of confidential and potentially sensitive data is catastrophic for an organization, but often affects also end users, that can experience identity theft and violation of their privacy. As an example, we can consider the *Equifax* data breach, which as per their last report on October 2, 2017 affected 145.5 million customers in the United States [2].

[1] http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017/
[2] http://www.bbc.co.uk/news/business-41474329

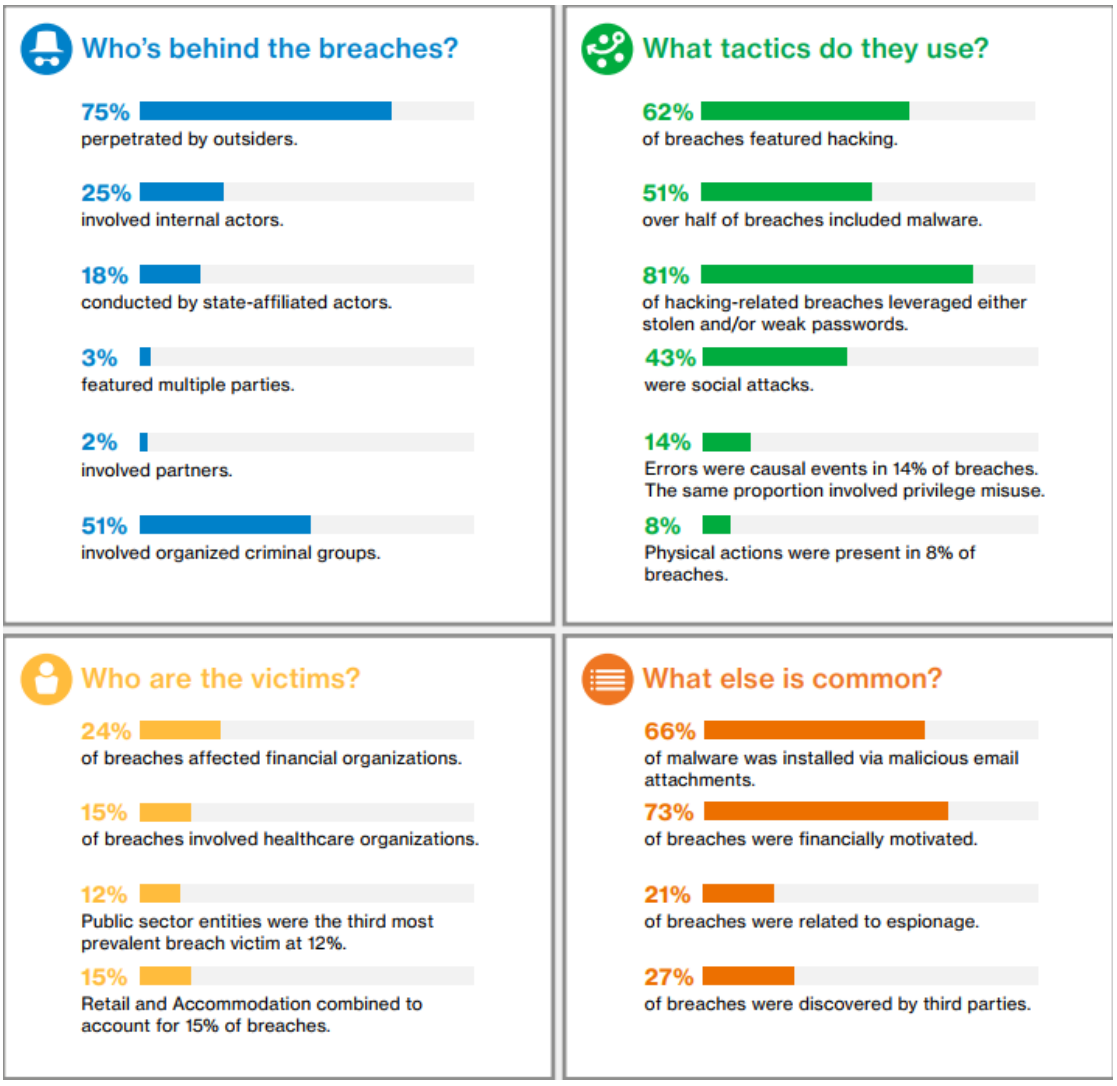

FIGURE 1.1: Summary of the Verizon Data Breach Investigation Report 2017

Over time, many security tools have been designed and adopted to prevent these breaches, such as **Intrusion Prevention Systems**, **Intrusion Detection Systems** and **Next Generation Firewalls**. Despite the large number of systems that work together to prevent them from happening, we still count many breaches every year. Moreover, many companies do not disclose incidents if possible. We see from Figure 1.1 that 27% of breaches are reported by third parties: this means that there are potentially many undisclosed breaches that organizations hide in order to protect their market value. Sometimes we even learn that companies paid in order to keep the breach hidden, like in the case of *Uber*[3].

[3] http://www.bbc.co.uk/news/technology-42075306

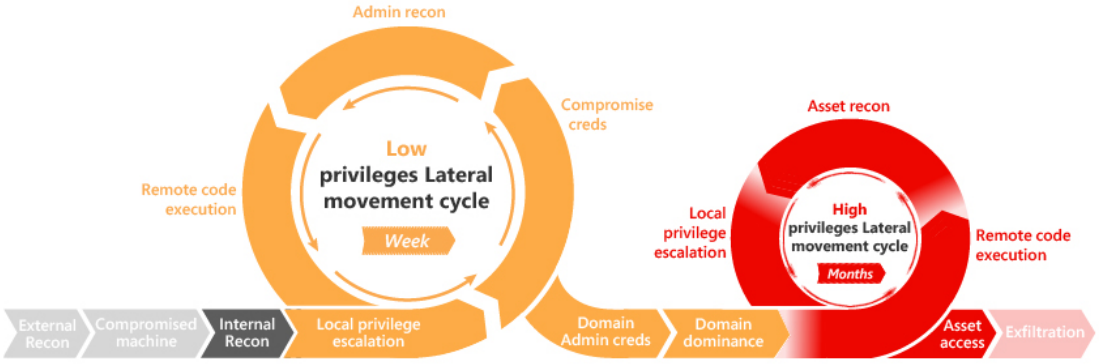

FIGURE 1.2: Microsoft's ATA kill-chain

Given the partial failure of the aforementioned tools, there has been an increasing need for a new way to defend against breaches. We observed the birth of a new activity in the security industry: *hunting for anomalies*. The *hunter* is a new figure that is meant to *actively* search for anomalies in the network in order to spot the early stages of a breach and take actions to prevent or mitigate it. Figure 1.2 shows Microsoft's version of a breach *kill-chain*[4]. Ideally, the hunting process should detect a breach before the *Asset access* step.

A **Breach Detection System** is a security device designed to help the hunter in performing this activity. BDSs differ from the aforementioned perimetral security tools, because they are designed to also look for suspicious activity *within* the monitored network, rather than just scanning incoming traffic. This distinction is important when choosing the techniques to apply in order to detect anomalies, and even more relevant when interpreting the output of the system: when an IPS achieves a detection, it means that an intrusion *attempt* has been identified and blocked, while a detection from a BDS could mean that a breach *is in progress* or *has already occurred*. It is important to notice that BDSs still aim at preventing intrusion in a network, but the underlying assumption is that they should also provide as many tools as possible to the hunter, because they will not always succeed in the prevention process.

[4] https://docs.microsoft.com/en-us/advanced-threat-analytics/ata-threats

1.1.1 Types of Detection Systems

Many earlier works, such as *"Intrusion detection: a brief history and overview"* [11] detail the concept of Intrusion Detection System. According to the taxonomy provided by Debar *et al.* [6], we can distinguish two different types of systems:

- Knowledge-based (or misuse-based)
- Behaviour-based (or anomaly-based)

Provided that the difference between Intrusion Detection Systems and Breach Detection Systems has already been detailed above, the considerations made on the aforementioned works apply also in the case of BDSs.

Misuse-based IDS/BDS rely on a set of definitions of malicious activity to identify evidences of compromise in a network. This is done by checking the network traffic patterns against these definitions. As a specific example of misuse-based system, we can take a *signature-based* IDS such as Suricata[5]. The process of specifying signatures can be done at different abstraction levels and on different data types, and the results can be more or less specific to compromise between *detection rate* and *false positive* rate. More generally, we can say that the misuse-based category comprehends any mean of detection based on *knowledge*, where the network analyst can *characterize* what a malicious behaviour looks like and produce a signature for it. For known malicious behaviours, this approach works really well in terms of performance and false positive rate (depending of course on the quality of the signature). The obvious pitfall is that the system cannot match on unknown malicious behaviour, as no signature is defined for it.

Anomaly-based systems work by trying to understand and remember what is *normal* and alerting about activities that differ from this definition of normality. One consideration could be that a knowledge-based model could be used to achieve this, by defining a rule for normal traffic and letting the system run on the network to find behaviours that do not match the rule. This approach is unfeasible because of the complexity of the resulting signature: there are simply too many

[5] https://suricata-ids.org/

legit behaviours for a system to be able to create and match signatures for all of them. Machine learning approaches, on the other hand, can be trained with real world traffic and extract a model of *normality* that can then be applied on the network to spot anomalies form that model. Training and maintaining a machine learning model is a far easier task than maintaining a handwritten rule that defines what normal traffic looks like.

1.2 Goal and Context

The goal of the internship was to develop *part of* a BDS to gather useful information about the network and providing them to the hunter. The main outcome of this work is MANTIS, a machine learning mechanism to cluster hosts into communities based on their behaviour and unveil the *evolution* of these clusters throughout time. The system is able to spot *singular hosts* as outliers in the network. It was out of scope of this research to provide complete automated alerts for anomalous activities: MANTIS has been designed to be interactively queried by the hunter. Moreover, the system had to be deployed in a real scenario, for customers with different sized networks. The main big advantage was the access to network data from real networks, that allowed to test the system on up-to-date traffic from different customers.

The present work has been entirely done in the context of an internship at Lastline Inc[6]. The Company offers a comprehensive malware protection system that integrates different components, such as an advanced sandbox for dynamic malware analysis and a sophisticated network traffic analysis component for both perimetral intrusion prevention and internal network security monitoring.

The traffic analysis is achieved by means of *ad hoc* hardware appliances, called **sensors**, positioned in strategic points of a network in order to sniff the traffic and perform various IDS activities. The sensors collect a variety of security information out of the network they analyze. Particularly relevant to this work are three data feeds:

[6] https://www.lastline.com/

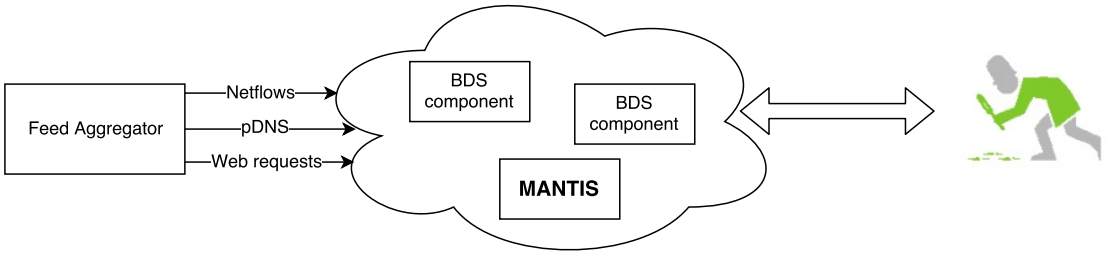

FIGURE 1.3: High level overview of the context

- Netflow records
- Passive DNS records
- Web request records

Whilst the details of these data formats will be given in the following pages, it is important to note that we did not have access to the packet captures themselves (pcap files), but only to these three feeds. Through a set of API calls, it was possible to interface the component with the preexisting system, obtaining real-time records for each of the listed data types.

Figure 1.3 shows the high level scenario in which MANTIS had to be put in place. Note the bi-directional arrow between the system and the hunter, representing the possibility for the latter to query the former, for example during an investigation.

While the inputs of the systems were fixed and well defined, the design phase needed to address not only the internals of the system, but also the outputs to be exposed to the user.

1.3 Why machine learning?

While machine learning techniques are an active research topic since the middle of 1900[7], the market expectation in artificial intelligence is extremely high still at the time of writing, as shown in Figure 1.4 [8]. Companies are applying machine learning to solve new problems, or to find new solutions to known ones.

[7] See https://en.wikipedia.org/wiki/Timeline_of_machine_learning for a complete timeline
[8] Source https://www.gartner.com/

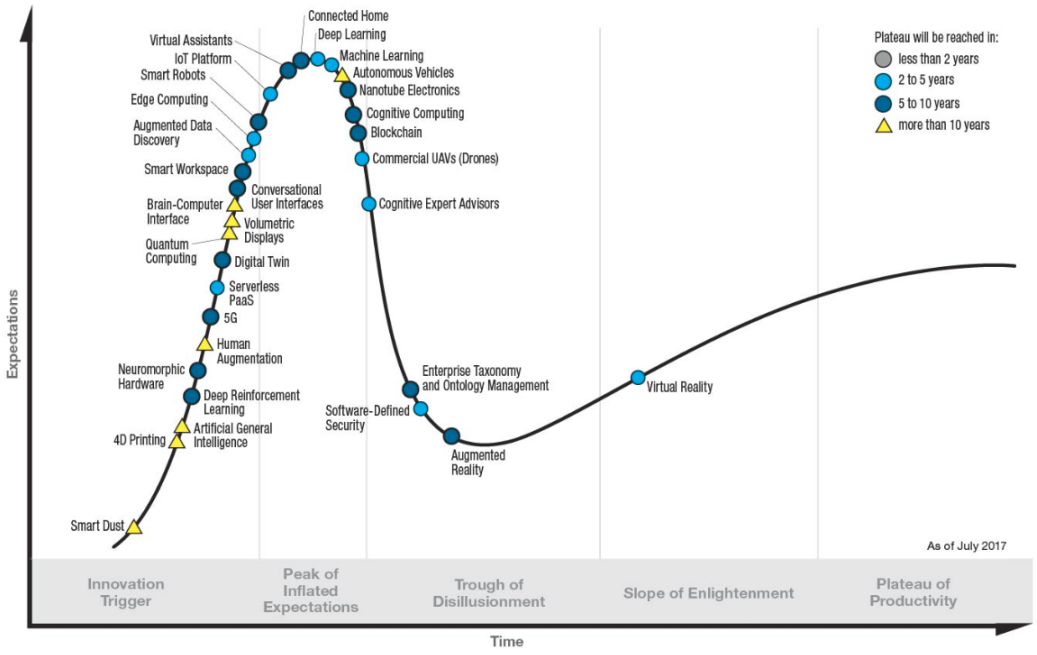

FIGURE 1.4: Gartner top trends in emerging technologies

In the context of network analysis and intrusion detection, machine learning techniques have been widely applied in literature, and their effectiveness has already been argued by Sommer *et al.* [18]: their conclusion is that it is possible to effectively apply machine learning to intrusion detection and, more generally, to traffic analysis, but it is important to define features with a high domain knowledge in order to obtain meaningful results.

Lastline has been active for several years in the field of advanced threat protection and a sophisticated *knowledge-based* system was already in place: from there the need of having a different point of view, that was possible to achieve using machine learning. Machine learning, applied on the huge amount of data that the company has, can give unforeseen insights on potentially anomalous activities on a network. The decision to apply an *unsupervised* model was forced by the absence of a *ground truth* to train the model on, while **DBSCAN** was the chosen clustering algorithm because it is able to identify clusters with arbitrary shapes and effectively spot outliers.

Chapter 2

Input types and machine learning algorithms

In the following pages we will go into the details of the main theoretic aspects of our work, providing useful vocabulary and explaining the basic concepts needed to understand the choices we made with respect to our approach. We will also give an idea of the type of networks we tested MANTIS on, giving some examples of the amount of data that the system processes.

2.1 Input types

As stated in Chapter 1, MANTIS is one of the components that get data from the existing infrastructure in order to monitor a network. We call these components *plugins*, because the operator can decide which of them are enabled and which are not.

We will now go into the details of each data type, i.e. Netflows, Passive DNS, Web requests.

2.1.1 Netflows

Firstly introduced by CISCO around 1996, Netflows are a mean of traffic aggregation now widely adopted in the industry. The protocol has now reached version 9 [4] and has been standardized by the IETF in the IPFIX protocol [2, 5]. The basic idea of the protocol is to aggregate information about an exchange between two hosts in a flow record, which is collected by a central server in the network.

Lastline defined its own proprietary version of Netflows, which contains the following fields:

- Protocol: supports TCP, UDP, ICMP
- Source IP address
- Source port
- Destination IP address
- Destination port
- Timestamp of the first packet of the flow
- Timestamp of the last packet of the flow
- Set of observed TCP flags
- Number of packets out
- Number of packets in
- Number of bytes out
- Number of bytes in
- Sizes of the first 10 packets exchanged

An example is shown in the JSON listing below:

```
{
    "payload_bytes_signature": [
      78,
      10,
      21,
      11,
      20,
      6,
      77,
      14,
      35,
      5
    ],
    "packets_out": 24,
    "bytes_in": 491,
    "packets_in": 14,
    "rsts_out": 0,
    "fins_out": 1,
    "source": "REDACTED(sensor ID)",
    "dst_ip": "192.168.1.1",
    "tags": [],
    "src_ip": "192.168.1.24",
    "src_port": 17828,
    "fins_in": 1,
    "syns_in": 1,
    "bytes_out": 116,
    "syns_out": 1,
    "proto": "TCP",
    "dst_port": 21,
    "rsts_in": 0,
    "ts_start": 1515431230000
 }
```


From Netflow data, we can extract meaningful features to characterize the behaviour of the hosts. For example, the destination port should help in identifying the type of application that has been contacted, the number of bytes helps define the volume of data that has been exchanged, the destination IP and its geolocation should give information on the location of the contacted server and the TCP flags should give insights about the result of the connection attempt and how it has been terminated. We focused mainly on destination ports and destination IPs, as we will discuss while detailing our approach.

2.1.2 Passive DNS

Passive DNS are data structures constructed from the DNS requests extracted from the traffic.

A single passive DNS record is composed of:

- Queried resource
- Type of the queried resource
- Class of the queried resource
- Response code from the server
- Data associated to the response
- IP of the querying host
- Source port used by the host
- Destination IP of the query
- Destination port of the query
- Timestamp of the beginning of the query
- Timestamp of the end of the query

An example can be seen in the JSON listing below:

```
{
    "rrname": "googleapis.l.google.com",
    "rdata": [
    (MODIFIED)
      "1.2.3.4",
      "1.2.3.5",
      "1.2.3.6",
      "1.2.3.7",
      "1.2.3.8"
    ],
    "source": "REDACTED(sensor ID)",
    "rrclass": 1,
    "error": null,
    "ttl": [
```

```
      57,
      57,
      57,
      57,
      57
    ],
    "dst_ip": REDACTED(DNS SERVER IP),
    "n": 1,
    "tags": [],
    "src_ip": 10.0.0.1(MODIFIED),
    "src_port": 0,
    "dst_port": 53,
    "ts_start": 1515433015000,
    "rrtype": 1
}
```

Passive DNS data are heavily filtered in Lastline appliances, therefore we preferred to focus on Netflow records during the design of MANTIS. The integration of the pDNS feed will be considered as part of future work.

2.1.3 Web Requests

This data type is designed to capture the fields of a web request as well as the end points involved in the request itself. It acts as a log of the whole HTTP transaction. A web request record is composed of:

- Hostname of the contacted server
- Path to the resource requested
- HTTP version
- Protocol
- HTTP method
- Referer
- User agent string
- Response code

- Response redirect: Location HTTP header in case of redirect
- Content type: the MIME type of the response body declared in the HTTP headers
- Body type: the MIME type of the response body derived from the analysis of the actual body
- Source IP address
- Source port
- Destination IP address
- Destination port
- Timestamp of the first packet of the request
- Timestamp of the last packet of the request

An example of a Web request record is provided below:


```
{
    "referer": "http://www.example.com/example/path",
    "response_code": 0,
    "method": "GET",
    "http_version": "HTTP/1.1",
    "response_body_type": null,
    "source": "REDACTED(sensor ID)",
    "dst_ip": "1.2.3.4(MODIFIED)",
    "src_ip": "10.0.0.1(MODIFIED)",
    "src_port": 65520,
    "path": "http://example2.com/example/path2",
    "hostname": "example2.com",
    "response_content_type": null,
    "response_body_earlyhash": null,
    "proto": "TCP",
    "resource_path": "/track",
    "dst_port": 8080,
    "response_redirect": null,
    "ts_start": 1515434363000,
    "user_agent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36
        (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36"
}
```

A good candidate for characterizing a host is the user agent string, especially nowadays, because most applications use a custom user agent string to perform web requests. By collecting statistics about the user agent strings used by a host, we should be gathering information about the applications that run on that host: this should intuitively help the behavioural classification that we want to perform. The use of user agents as distributions to further characterize hosts is part of future work and preliminary considerations on this possibility will be given in the final Chapter.

2.2 Traffic amount

Since we focused the present work on features extracted from Netflow data, we will present statistics about three different networks that we worked on, specifically regarding the number of hosts and the number of Netflow data that the system processed.

Network	Hosts/h (peak)	Netflows/h (peak)
Network A	97	32800
Network B	6035	1283050
Network C	25400	600000

TABLE 2.1: Three different network types we worked with

For Network A and B, the data was gathered on different days in November 2017, on a per hour basis. Network Ccontains Netflows captured over one hour of traffic in September 2016. As we can see in Table 2.1, it is possible that a lower number of hosts generate more Netflows than a bigger network.

2.3 Machine learning algorithms

Our entire work is based on a machine learning approach for designing an investigation tool. Although the algorithms and statistical tools that are used have been

long discussed in literature, it is important to know how they work in order to make important considerations afterwards.

A very high level view of the decisions that have to be taken when choosing a machine learning model can be obtained from Figure 2.1, which shows an overview of the model selection process from SAS blog[1].

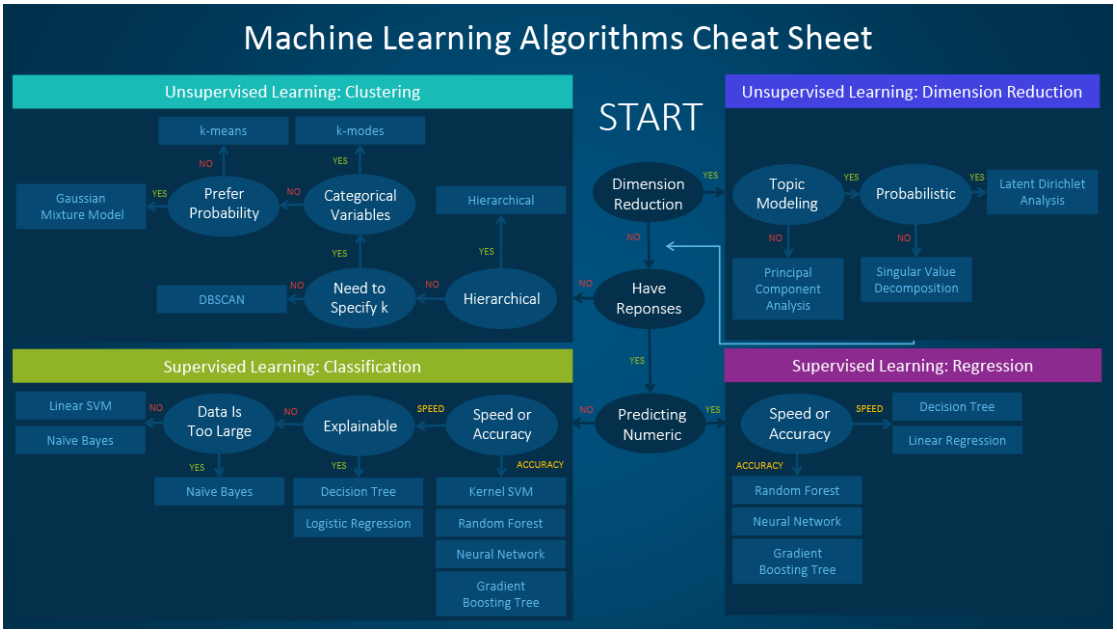

FIGURE 2.1: Machine learning cheat sheet from SAS

Although this is just an overview and not a precise algorithm to follow when choosing a model, it can give an idea about the considerations needed for a proper tool selection. In our case we are dealing with unlabeled data, so this should exclude the supervised learning models. Moreover, we want to model *communities*, which may be well defined by clusters. Finally, we have no knowledge about the number of clusters that we expect a particular network to show, so we cannot specify k. One important aspect that the image does not show is the ability of DBSCAN to identify *outliers* in the data, that we will later start calling *singular* hosts.

[1] https://blogs.sas.com/content/subconsciousmusings/2017/04/12/machine-learning-algorithm-use/

2.3.1 DBSCAN

Density-based spatial clustering of applications with noise, or DBSCAN, is a clustering algorithm discussed by Ester *et al.* [8].

The algorithm is meant to run on datasets that present clusters of arbitrary shape and it is designed to be able to follow the shape of the clusters and identify noise as *outliers*. It gets as input two parameters:

- A distance ε
- A number of points $minPoints$

The algorithm gives the following definitions:

- The ε-*neighbourhood* of a point p, $N_\varepsilon(p)$, is the set of points that are closer than ε from p.
- A point p is said to be *directly density-reachable* from a point q wrt *minPoints*, ε if:
  - $p \in N_\varepsilon(q)$: p has to be in the ε-*neighbourhood* of q
  - $|N_\varepsilon(q)| \geq minPoints$: any point for which this holds is called *core point*
- A point p is *density reachable* from q if there are $p_1, ..., p_n$ points such that $p_1 = q, p_n = p$ and p_{i+1} is directly density-reachable from p_i
- Points p and q are *density-connected* if there is a point o such that p and q are density-reachable from o

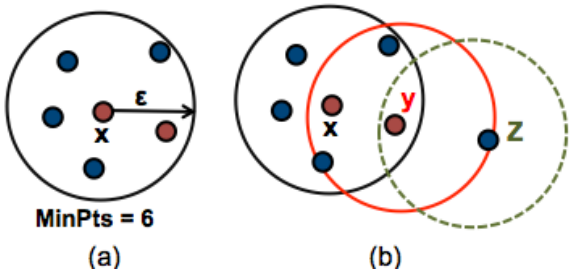

FIGURE 2.2: The three different type of points in the DBSCAN algorithm

Figure 2.2 shows the three types of points identified by the algorithm. In particular, we see that for the represented ε and *minPoints*:

- x is a *core point*
- y is directly density-reachable from x, but $|N_\varepsilon(y)| < 6$. y is then called *border point*
- z is not a core points and is not density-reachable from any core point. z is said to be a *noise point*

Given these definitions, the authors of the algorithm proceed to define a cluster:

A cluster C is a non-empty subset of the dataset for which:

- $\forall p, q$: if $p \in C$ and q is density-reachable from p, then $q \in C$
- $\forall p, q$: p and q are density-connected

The algorithm starts by selecting a random point in the dataset and expanding the cluster according to the definition given above, until the two conditions of *maximality* and *connectivity* are satisfied. In the end, each point of the dataset is identified as core, border or noise point. In the first two cases a label is assigned to the point (such that the same label is assigned to points belonging to the same cluster and different labels are assigned to points belonging to different clusters), while noise points are left without a label.

Chapter 3

Preliminary work: OS and application fingerprinting

As discussed in Chapter 1, we worked on developing part of Lastline's Breach Detection System. The main outcome of this work is MANTIS, an investigation tool based on behaviour-based clustering of hosts in a corporate network.

We use this chapter to discuss some of the preliminary work that has been done before designing MANTIS.

The first attempt to profile a host on the network has been through passive fingerprinting. The proposed solution to OS and application fingerprinting is based on passive DNS data.

The first step was to build a component that acted as a DNS cache and was able to answer to queries based on the traffic that was observed in the network.

An overview of the DNS cache is presented in Figure 3.1.

The DNS cache implements a circular buffer in which we store all the resolutions observed in the network. The time to live of a single entry corresponds to the TTL value of the DNS record. Upon insertion, if the cache is full, we delete the entry which is the closest to its expiration time.

The basic concept behind the passive fingerprinting is that some servers are contacted only by specific applications or Operating Systems. For instance, we argue

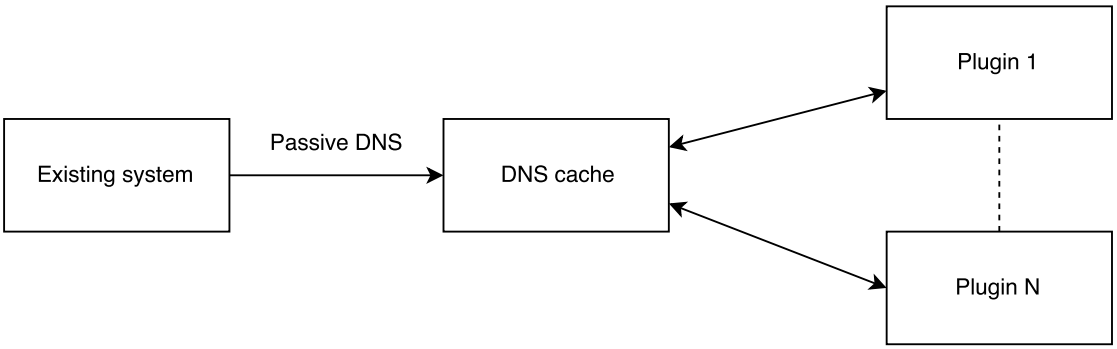

FIGURE 3.1: High level context for the DNS cache development

that the hostname `archive.ubuntu.com` is contacted by hosts running the Ubuntu Operating System, while `update.microsoft.com` can be a good indication of a host running Microsoft Windows.

It is important to notice that while in general there can be many exceptions to this heuristic, the context of a controlled enterprise network environment makes it easier to isolate and ignore these exceptions, if they are benign. For example, a server running several virtual machines can be easily put into a whitelist by the network administrator.

The DNS cache component can be queried for direct and reverse resolutions, and returns an answer based on the records it observed in the network. As Figure 3.1 shows, many *plugins* can interact with it and query for domain names or IP addresses.

We then proceeded to develop two plugins that, like MANTIS, process Netflow records one by one. For each destination IP address observed, the plugins query the DNS cache to obtain the domain name. If the domain name matches one of the servers that are defined in a configuration structure, they mark the host as running that particular Operating System and/or application. A flow chart of the process is shown in Figure 3.2.

Although not directly related to MANTIS, this type of passive fingerprinting is essential to get insights on a network during an investigation. For instance, knowing that some hosts were running a specific Operating System during an attack is helpful to fucus the attention of the incident responder on specific portions of the network.

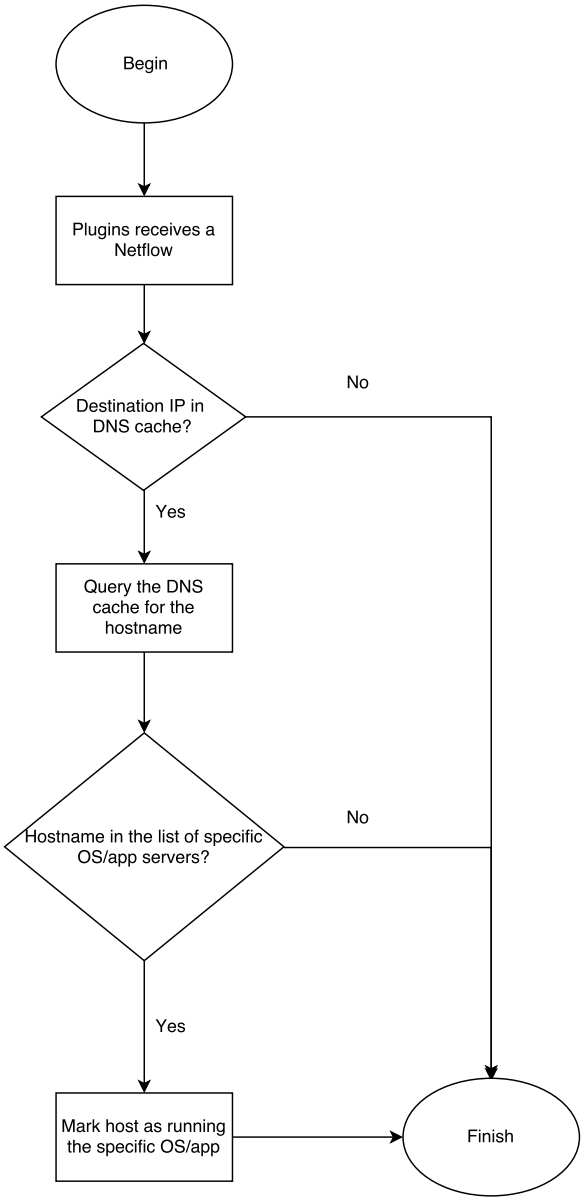

FIGURE 3.2: Flow chart of the OS/app fingerprinting process

Chapter 4

Host clustering

We will now go into the details of the approach that we propose in this document, stressing the intuition behind it, describing our experiments and our setup.

As discussed in Chapter 1, we wanted to design an investigation tool system based on machine learning techniques. It is important to note that Sommer *et al.* already warned us [18] about the use of machine learning with improvised feature selections. Specifically, they state that it is extremely important to design the features for the algorithm such that they can easily be justified with real case network traffic examples.

We will now try to express the rationale behind the choice that we propose for the features, but we stress that this approach can be extended and adapted to any attribute that can be expressed as a frequency distribution.

4.1 Feature Selection

From Chapter 2 we see the set of attributes that we can access in the input data. We focused initially on Netflows, because intuitively they carry the biggest amount of information about the network activity of a host.

Different solutions have been proposed in literature, some of them (such as the entropy of the destination IP addresses and ports, the total volume of data exchanged and the average number of bytes per flow) were tested with our approach,

but a manual inspection of the results did not seem to spot a meaningful clustering of hosts. We then decided to adopt a distribution-based solution, because it is the one that should keep track of the activity without losing granularity, unlike with aggregated measured such as the entropy. Moreover, such measures can easily be computed staring from the distribution themselves, if needed.

We propose here a set of distributions that we decided to use with our approach:

- Distribution of the destination countries
- Distribution of the destination ports, enriched with the PCR value

The rationale behind the choice of the destination country as a feature is simple: we expect the majority of hosts to communicate with servers located in the same country as the client. More generally, we expect more than a single outlier to reach to the same country in a given time bin. This is enough to put these hosts in the same cluster, based on the set of destination countries, and mark the destination country as *commonly visited*.

We then chose the destination ports because it is clear that they play an essential role in defining the type of traffic. Especially in a controlled enterprise environment, it is easy to identify the set of services used in the network and the ports they rely on. Moreover, without the packet payload, we cannot rely on *Deep Packet Inspection* techniques to identify the type of traffic that is captured. We then decided to enrich the destination ports distribution with the *Producer-Consumer Ratio*. The *PCR* is defined as the amount of bytes sent divided by the amount of bytes received in the flow. Again the rationale behind this choice is related to the fact that we want to differentiate traffic with different PCR values on the same port. A possible use case is to differentiate an interactive session (for example on port 22 for ssh with balanced PCR) from a data exfiltration (still port 22, but way more bytes consumed than produced).

Therefore, we defined a method to extract these features from the netflows, simply counting the occurrences of these features in a single time bin (we used 1 hour) while processing netflows online, before normalizing them to a discrete distribution. We then proceeded to concatenate these distributions into a single one,

because we wanted to consider both countries and ports together. Clearly many ssh connections to a host belonging to the company's intranet are less suspicious than many ssh connections to a server located in another country. Concatenating the distributions had of course to yield another distribution, therefore we simply divide each point of the distribution by the number of distributions we are considering, as described in the formula below:

$$D_{final} = \frac{1}{2}D_{countries} \quad | \quad \frac{1}{2}D_{ports+PCR}$$

where | is the concatenation operator.

It is clear that the underlying approach based on distributions is *not* dependent on the particular choice for the attributes and can be generalized and extended to an arbitrary number of distributions, with some performance constraints that will be further discussed in Chapter 5. The generalized formula for N distribution is the following:

$$D_{final} = c_1 D_1 \quad | \quad c_2 D_2 \quad | \quad ... \quad | \quad c_N D_N$$

By tuning the parameters $c_1 ... c_N$ we can give different weights to the single distributions, in case we want to assign different importance to each of them. It is clear that the requirement for the coefficients $c_1 ... c_N$ is that $\sum_i c_i = 1$.

The single resulting distribution constitutes our data point. We can now proceed and apply DBSCAN.

The algorithm is applied as is, with no major modifications to it, if not the definition of a custom metric to use instead of the defined Euclidean distance.

4.2 Distance Metric

Although the Euclidean distance can be used to compute the distance between two arrays, it yields a result that is dependent on the length of the arrays themselves.

As anticipated, we worked with *distributions* of traffic attributes such as destination countries and destination ports. This means that for us a single data point is represented by an array, in which each element represents the frequency count of an observed traffic attribute. A promising distance metric that handles distributions is the *Jensen-Shannon Divergence*. In order to define it we first need to define the *Kullback-Leibler Divergence* or *relative entropy*.

4.2.1 Kullback-Leibler Divergence

Presented in [14], the *Kullback-Leibler divergence* is a metric to measure the distance between two distributions. The formulation is the following:

$$D_{KL}(P||Q) = -\sum_i P(i) \log \frac{Q(i)}{P(i)}$$

The aforementioned definition is valid only if $\forall i, Q(i) = 0 \implies P(i) = 0$.

One big drawback of this metric is its non-symmetry, this means that we cannot use it as is as a distance measure.

4.2.2 Jensen-Shannon Divergence

This distance metric is a symmetric version of the *Kullback-Leibler divergence*. It is defined as follows:

$$D_{JS}(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M), \quad \text{with} \quad M = \frac{1}{2}(P+Q)$$

Which is the sum of the Kullback-Leibler divergence from both distributions to the mean distribution M. One important property is that $0 \leq D_{JS} \leq 1$, which means that it can be used for arbitrary long distributions, yielding a result that varies from 0 to 1 each time. As a result, we can tune the ε parameter of the DBSCAN *a priori*, without requiring further tuning after deployment. This means that the user of this system is required to select a reasonable value just for the *minPoints*

value, without having to run several experiments to optimize the ε parameter. We stress that this is an important achievement, since parameter tuning is one of the most difficult parts of defining a machine learning approach.

4.3 Applying DBSCAN

The application of the DBSCAN algorithm yields two important results:

- Being a noise-aware algorithm, DBSCAN is able to effectively identify *outliers* in the single time bin. We call these outliers *singular hosts*
- Being a clustering algorithm, DBSCAN is able to group *similar* hosts, categorizing them based on their *behaviour*

The first thing to notice is that *singular* hosts are not necessarily a security issue. For instance, in an enterprise network with a single internal file server, this will be categorized as *singular*, without any malicious activity performed by the server.

More interesting are the *migrations* of hosts between clusters. For example, in an enterprise network with a cluster of web servers, if one of them *becomes singular* it may be worth investigating. The system is useful during an investigation after a data breach, when the incident responder can query for hosts that had a behaviour that is similar to the compromised machine.

4.4 Host dynamics

To track the *migrations* of hosts between different hours, we take our system a step further and define a method to map the clusters identified between consecutive independent runs of DBSCAN. This method is based on distances between *centroids* of the clusters. The *centroid* of a cluster is, by definition, the *meta-point* for which every coordinate is the mean of the coordinate of the points belonging to the cluster. In our scenario this translates to computing the *mean distribution* of the cluster and considering it as the cluster's *centroid*. This point becomes the

only representation of the cluster, and we can use it to map the clusters yielded by consecutive runs of the clustering algorithm.

The steps we defined in order to do so are the following:

- Compute DBSCAN in a time bin t
- Compute the centroids of the clusters: $C_1^t, ..., C_N^t$
- Compute DBSCAN in the following time bin $t+1$
- Compute the centroids of the clusters: $C_1^{t+1}, ..., C_M^{t+1}$
- Define C_i^{t+1} as the *evolution* of C_j^t if and only if:
  - C_j^t is the closest centroid to $C_i^{t+1} and$
  - $dist(C_j^t, C_i^{t+1}) \leq \theta$, where θ is the minimum distance between the centroids $C_1^{t+1}, ..., C_M^{t+1}$

If a cluster C_i^{t+1} is associated to C_j^t and there is a host h such that $h \in C_j^t$ in time bin t *and* $h \in C_i^{t+1}$ in time bin *t+1*, we say that h *stayed in the same cluster*, we say that h *migrated to cluster* C_k^{t+1} otherwise.

4.5 Visualizing results

4.5.1 Graphlets

To visualize the behaviour of a host in a single time bin we propose the use of *graphlets*.

Graphlets were first introduced by Karagiannis *et al.* [10]. They are a graph in which nodes are divided into rows (or columns). Nodes belonging to a single row represent different observed attributes of the connections performed by the host.

Figure 4.1 shows an example graphlet. The host that is described by the graphlet is the *source IP*, in this example `10.0.0.1`.

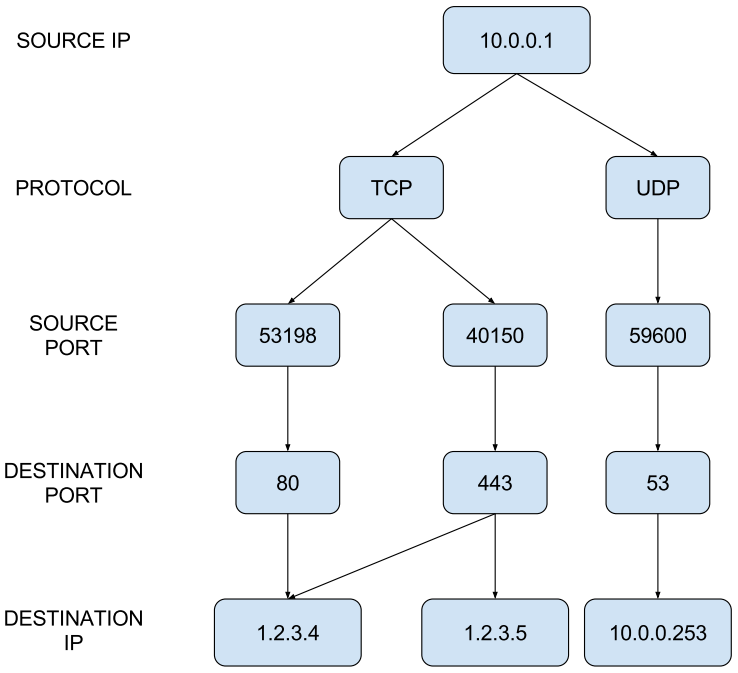

FIGURE 4.1: An example of a graphlet

The figure should be read as follows:

```
Host 10.0.0.1 contacted three different hosts: 1.2.3.4, 1.2.3.5 and
    10.0.0.253.

Host 1.2.3.4 was contacted on both TCP ports 80 and 443, while host 1.2.3.5
    was contacted only on TCP port 443.

Host 10.0.0.253 was contacted on UDP port 53.
```

From the figure, one can get a clear idea of what was the behaviour of host `10.0.0.1`.

One important consideration about graphlets is that, as per the description given by Karagiannis *et al.* in their work, an edge should be considered as *"at least one packet is observed"*; therefore we lose the information about the number of packets that correspond to a single path from the root of the leaf to the tree.

Extracting features directly from the graphlets and performing clustering on these features is possible and has been already done by Himura *et al.* [9]. In our scenario,

the lack of information about the count associated to a single connection would be too heavy when performing the clustering, and it would yield an extremely high number of different clusters, because we would not associate hosts for which the *majority* of connections are similar.

4.5.2 Evolution graph

Having identified a way to model the evolution of the clusters throughout time, we propose a mean of visualization of such evolution, that we call the *evolution graph*.

An example of such graph can be see in Figure 4.2. The presented example has to be read as follows:

```
In time bin number 1 there are two clusters, labelled respectively 1 and 6,
    for which the closest cluster in the previous time bin is the one
    labelled 1.
The distances between these two cluster and the preceding one are 0.005 and
    0.007 respectively.

In time bin number 1, cluster 1 has 120 members, while cluster 6 has 50
    members.

Of cluster 1 in time bin 0, 90 hosts are in cluster 1 of time bin 1, while 5
    hosts are in cluster 6 of time bin 1.
```

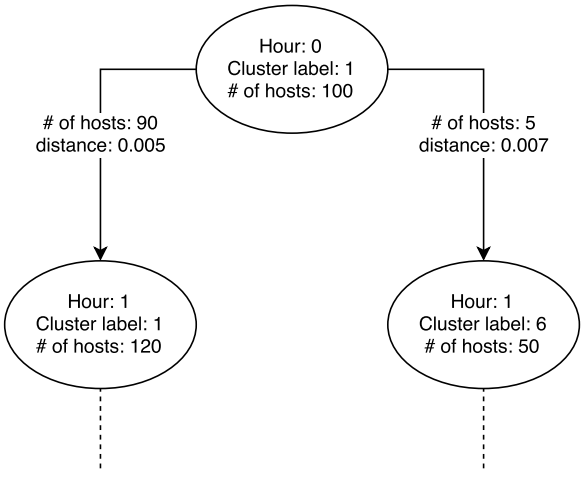

FIGURE 4.2: An example of the evolution graph

In this example, we see that it is possible that a cluster *splits* into two clusters in the following time bin. Some interesting questions that the system can be queried for, given this example, are:

- If 90 hosts of C_1^0 are in C_1^1 and 5 hosts are in C_6^1, what happened to the remaining 5 hosts? Are they still in the network? Did they *migrate* to a cluster C_k^1, *with* $k \neq 1, 6$ (possibly in cluster -1)?
- Cluster C_1^1 has 120 hosts, of which 90 come from C_1^0. Where do the remaining 30 hosts come from? Did they appear in time bin 1 or do they come from C_k^0, *with* $k \neq 1$?

All of these queries can be performed by the *hunter* to the system, because we keep track of the *centroid* of each cluster and the set of hosts that belong to each cluster in a given time bin.

Some of these migrations can be interesting, while others can easily be due to new hosts connecting to the network or normal connection patterns. It is out of scope of this research to define heuristics aiming at discriminating between these categories, as is part of the hunting activity to decide what is interesting and what is not.

Chapter 5

Implementation and evaluation of the algorithm

Since this document is also meant to be the report of an engineering internship, in the following chapter we will go beyond the simple research problem that we analyzed and we will discuss some of the steps that we took in implementing the system, together with some of the issues that we encountered and the considerations that we had to make about the performance of the system before starting to deploy it.

We will follow the steps that led to the final formulation of the solution that we proposed in Chapter 4.

5.1 Host statistics aggregation

Moving towards MANTIS and our final approach, we then developed a component that is able to extract statistics about the hosts by processing Netflow data. The component is implemented in Python, by means of a class that contains counters for each feature value that we are interested in.

The code for this class is shown in Appendix A. The main method that the class exposes is the `process` method, which performs the actions needed to aggregate

the data and construct the graphlet by means of the `networkx` Python package. The class also exposes all the properties needed to extract the distributions.

5.2 Clustering with DBSCAN

Appendix A presents also the Python code that we propose for the actual clustering and the definition of the cluster *evolution* as described in Chapter 4. As the listing shows, we relied on the `sklearn` implementation of DBSCAN, while the Jensen-Shannon Divergence metric is provided by the `js` package. As specified in Chapter 3 and 4, the use of a metric that is bounded between 0 and 1, despite the length of the distribution, allowed us to experiment different values for the ε parameter of the DBSCAN algorithm, and we decided that 0.01 provided a good separation of the distributions, while keeping the number of clusters reasonable.

5.3 Performance evaluation

One of the most important parts of the feasibility study was to evaluate the performance of the system and possibly try to find some means of boosting the performance to achieve an acceptable computation time of the algorithm. As stated in Chapter 4, we picked 1 hour as time unit to aggregate statistics and compute the clustering of the hosts. Although this particular choice was somehow arbitrary, we argue that it is a reasonable compromise between having enough data (picking a smaller time bin could lead to too few Netflows per host to obtain a meaningful distribution) and dynamicity of the hosts (a bin that is too big would fail to provide insights about the *migrations* of the hosts).

Given the choice of 1 hour time bins, the hard constraint on the computation time is equal to the time bin itself: if the computation time is higher than the time bin, the system would accumulate delays and eventually stop working properly. In order to estimate the time taken for a single run of the system, we took data from Network C presented in Chapter 2 and recorded the computation times of

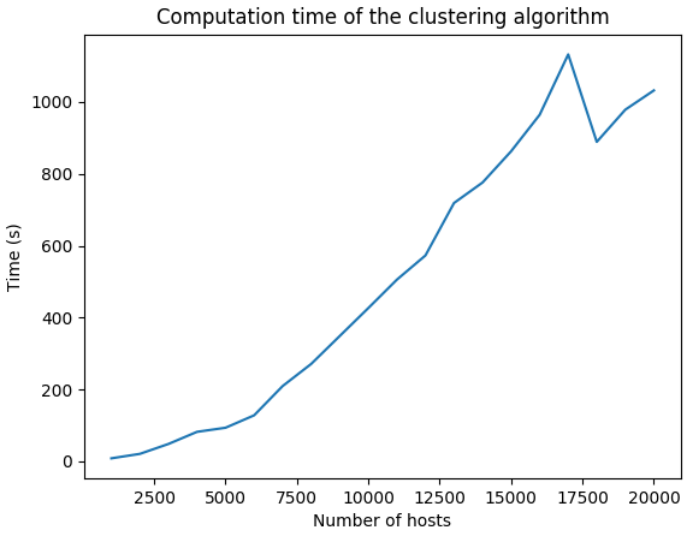

FIGURE 5.1: Computation times as a function of the number of hosts

the DBSCAN algorithm as a function of number of data points and length of the distributions.

Figure 5.1 shows the dependency of the computation time of the DBSCAN algorithm as a function of the number of hosts.

Besides the sudden drop of the computation time between 17000 and 18000 hosts, which is most likely due to some internal optimizations of the DBSCAN implementation, we see that overall we confirm the theoretical $O(n \log n)$ computation time of the DBSCAN algorithm.

We then fixed the number of hosts at 1000 and plotted the computation time as a function of the length of D_{final}. We can clearly infer a linear dependence of the time on it. Figure 5.2 shows the linear dependence that we observed.

All the plotted times come from experiments run on a laptop relying on an *Intel Core i5* processor and 8 GB of RAM.

By considering the worst case scenario of a network with around 25000 hosts as Network *C*, we clearly need to find a way to reduce the length of the distribution, because the worst case scenario for the number of different destination ports observed can reach 393210 values. This upper bound is computed as follows: there are 65535 ports for the TCP protocol and the same amount for the UDP protocol;

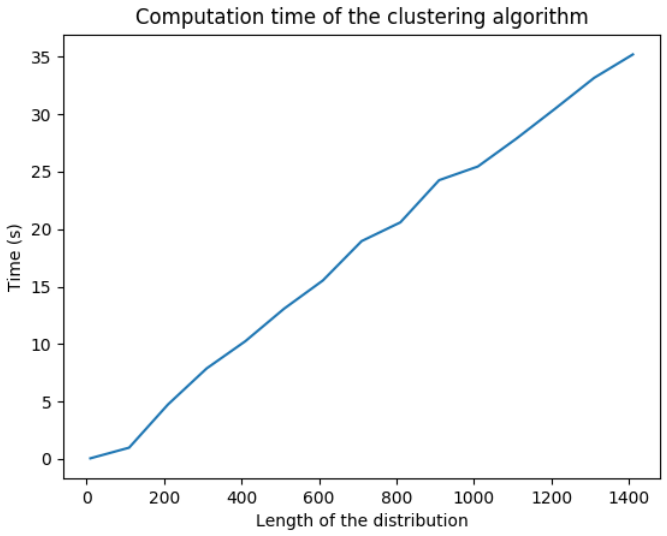

FIGURE 5.2: Computation times as a function of the number of hosts

as stated in Chapter 4, we enrich the single value of the destination port with the PCR value, which itself can assume the values *low, balanced* and *high*.

The strategy that we propose as a solution to the possible explosion of the destination ports values is to identify two set of ports as follows:

- We call *significant ports* the ports that we observe more frequently in the network overall.
- We call *other ports* the ports that we observe less frequently in the network overall.

All the ports observed for a single host that belong to the *other ports* bin will be grouped together in a single point of D_{final}.

Here follows an example of this strategy, applied on the values of the ports without the PCR values, for readability reasons. Figure 5.3 shows all the observed destination ports in Network B in a single time bin.

We can see that the ports follow a clear *heavy-tail* distribution, where the first value, which represents the aggregation of the ports *80* and *443*, dominates the statistic. Given a threshold θ, that we set after running some some experiments, we consider all the ports for which the distribution value is less than θ as *other*

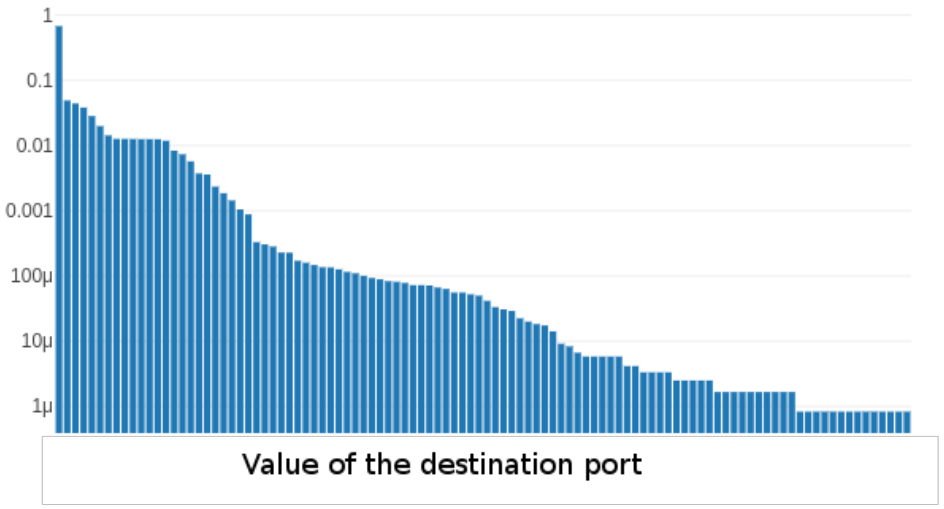

FIGURE 5.3: Distribution of the overall observed destination ports in a medium network

ports, aggregating them and adding the value of the distribution for each of them to obtain a single point in the distribution. While the tuning of the θ parameter requires some action from the user of the system, it guarantees that the length of D_{final} does not make the computation time explode. Moreover, with more computation power, we could loosen this constraint and allow a longer D_{final}. Note that we did **not** make the same consideration for the distribution of the countries, because the number of different countries has an upper bound of 195 at the time of writing, which is a reasonable amount and does not justify any aggregation of country values.

5.4 Cluster evaluation

Finally, we proceeded in evaluating the quality of the clusters by applying two different quality measures: the two different applied measures are detailed below.

5.4.1 Silhouette coefficient

The *silhouette coefficient* [17] measures how much a data point is similar to the data points in the same cluster, compared to the data points belonging to the closest different cluster.

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

Where $a(i)$ is the average distance from data point i to all the points of the cluster it belongs to, while $b(i)$ is the distance from data point i to the closest data point in a different cluster.

It is clear that $-1 \leq s(i) \leq 1$, where 1 is to be intended as the best possible score and -1 as the worst possible score.

5.4.2 Calinski-Harabasz score

The *CH* score [3] is another metric to define cluster quality. It compares the *between cluster variance* to the *within cluster variance*.

The formulation is the following:

$$CH = \frac{SS_B}{SS_W} \times \frac{N - k}{k - 1}$$

In the formula, SS_W, or *within cluster variance*, is calculated as:

$$SS_W = \sum_{i=0}^{k} \sum_{x \in C_i} ||x - m_i||^2$$

And represents the sum over all the clusters of the sum of squares inside the cluster.

In order to compute SS_B, we need to first compute the *total sum of squares* TSS, which is defined as

$$TSS = \sum_{x} ||x - M||^2$$

Where M represent the mean point over the entire dataset. Now $SS_B = TSS - SS_W$.

This metric presents a global maximum when there is a balance between the number of clusters and the variance inside the clusters themselves.

With these definitions in mind, we proceeded to evaluate the quality of the clusters over the medium sized network already discussed in the previous pages.

Figure 5.4 shows the plot of the scores over three days of captured traffic on Network B: the peaks in number of different clusters and the drop of the quality measures correspond to peaks of activity on the network, where because of the higher number of observations, we have smoother transitions between the points, which brings the algorithm to join clusters that were well separated with less hosts on the network. Part of the future work is to try to implement an adaptive clustering, where the distance threshold depends on the density of the data points.

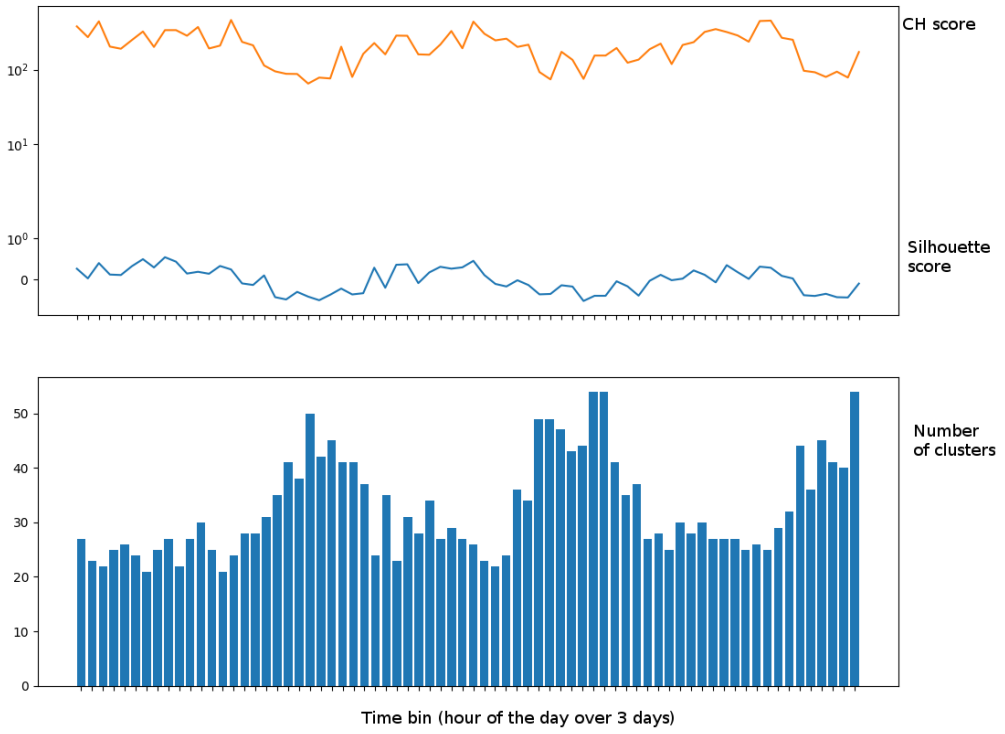

FIGURE 5.4: The silhouette score (blue) and CH-score (orange) over three days, along with the number of different clusters

It is important to notice that in our scenario these results cannot be taken in an absolute manner. For example, we often see the silhouette score drop under 0, which means that some data points may be closer to the closest point in a different cluster w.r.t. the centroid of the cluster they belong to. Lets us take as an example the situation depicted in Figure 5.5. DBSCAN successfully separates the clusters, but some points of the red cluster are indeed closer to the clue points compared to the centroid of the red clusters (represented by the black square); same thing happens for the blue cluster, where the centroid is represented by a black circle. This would make the overall silhouette score drop, even if the algorithm has correctly separated the clusters.

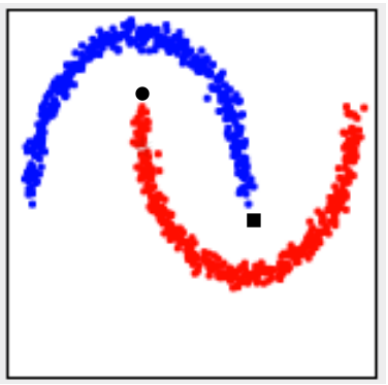

FIGURE 5.5: An example of failure of the silhouette score

These misevaluation cases, which are more common during peak activities of the network, happen because of the higher density of points in the dataset, which brings to more widespread clusters. One possible solution to this problem is to find a local optimum of the CH score by modifying the distance threshold ε of the DBSCAN algorithm in an automated manner. Decreasing ε and re-computing the algorithm has some important consequences:

- The number of different cluster would increase even more with a lower ε
- We would need to recompute the algorithm, which is an expensive procedure

It is part of future work to find a solution to this issue. For our concept of cluster, which should represent a community of similar hosts, it is enough to follow the density-based definition of clusters obtained by running DBSCAN.

Having described some of the implementation aspects that we had to consider while working on MANTIS, we can now provide a use case example of use of the system during an investigation process.

Chapter 6

An example scenario

We now proceed to present a use case of MANTIS during an investigation. We took data from *Network B* of our dataset during 6 hours of activity of November 2017. Instead of looking for attacks already present in the data, which is a task that would have required many different experiments on all the historic data that Lastline keeps, we decided to *inject* a possible breach scenario that summarizes the breach steps shown in Figure 1.2.

Let us examine the behaviours that we injected in the network:

- Host A is downloading data over HTTP from a server in the US
- In the subsequent time bin, it starts downloading data from a server in China (initial compromise)
- It then starts brute forcing ssh credentials over the internal network (lateral movement)
- It then downloads many GB of data from an internal server over scp (asset access)
- It uploads the data over HTTPS to the same server located in China (exfiltration)
- It goes back to downloading data over HTTP from the US server

This toy example might be slightly unrealistic, but it shows us different stages of a breach and how the system classifies the host over time.

Figure 6.1 shows the labels assigned to Host *A* during the whole attack scenario. As expected, in time bin 1 the host has a defined label, which means that its behaviour is common in the network.

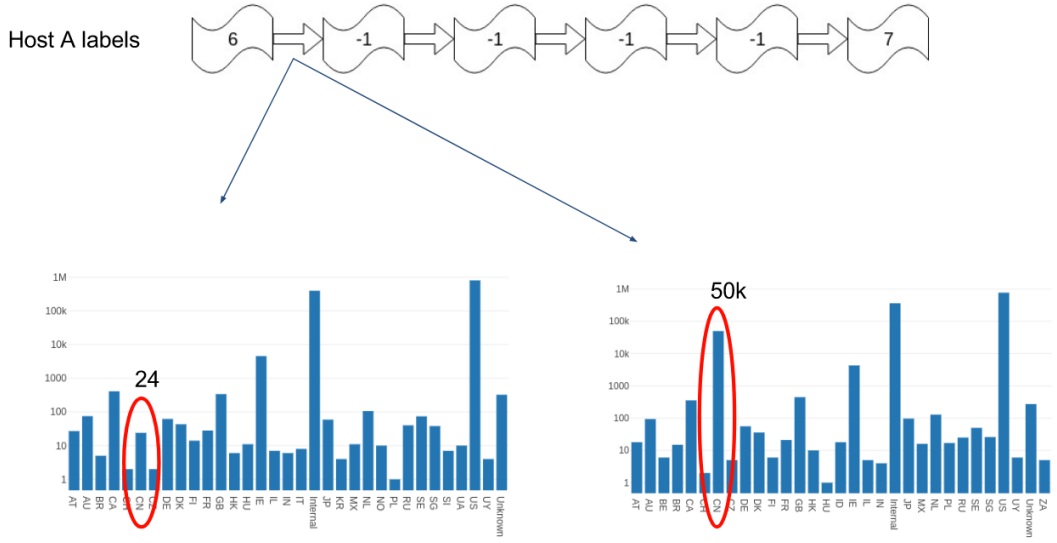

FIGURE 6.1: First compromise and how it affects the overall network traffic

In Figure 6.1 we also show the impact of the compromise phase on the overall traffic of the network. We see a sudden growth of Netflows towards China w.r.t. the ones seen in time bin 1. This corresponds to the *migration* of Host *A* from a defined cluster to cluster -1, which represents *singular hosts*.

We see in Figure 6.2 how the lateral movement phase affects the overall network traffic on the ports side, showing a sudden increase of the connections on port 22.

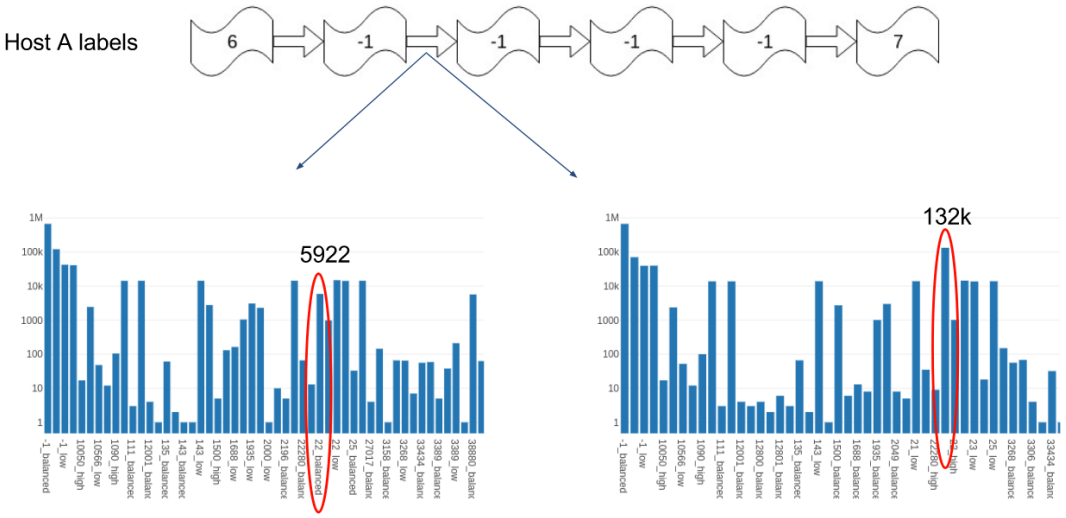

FIGURE 6.2: Lateral movement and how it affects the overall network traffic

The following two steps do not have a visible impact on the cumulative traffic distribution for the whole network, but we see from the previous pictures that the system still succeeds at labelling the host as singular.

Let us examine now the *evolution graph* for cluster 6 during the 6 examined time bins (from 10 to 15) in Figure 6.3. We see that cluster 6, that represents the hosts downloading over HTTP from US servers, stayed quite stable in terms of position in the hyperspace over the 6 time bins. Moreover, the system changed the labels of the cluster multiple times, ending up with label 7, which is coherent with the label assigned to Host A when it went back to its normal activity.

Finally, we gathered the following statistics during the experiments we made:

- Average number of active hosts per time bin: 2043.5
- Singular hosts in time bin 10: 797
- Average cluster size in time bin 10: 34.3
- Number of hosts that *became* singular in time bin 11: 78
- Number of hosts with the same set of labels as Host A over all time bins: 0

With these statistics in mind, here are some use case of usage of the system during an investigation:

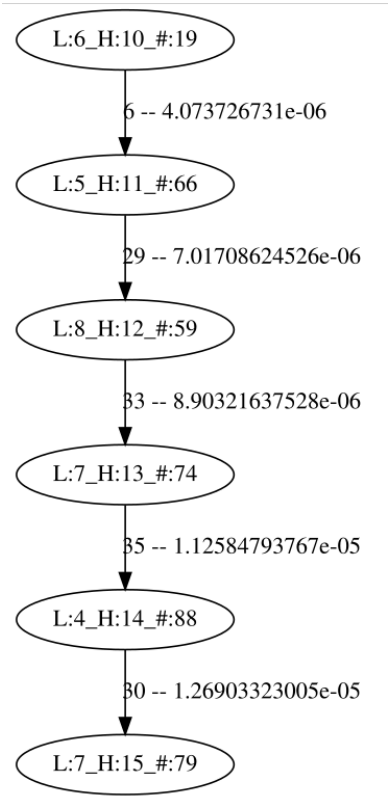

FIGURE 6.3: The evolution of Cluster 6

- Query for all the hosts that were labelled as belonging to cluster 6 in time bin 10: 19 hosts
  - These 19 hosts have been more in danger of the same compromise given that they were performing the same activity as Host A
- Query for all the hosts in other clusters that stayed singular over time bin 11, 12, 13, 14: 3 hosts
  - 2 other hosts may have been performing the same anomalous activities (compromise, lateral movement, asset access, exfiltration) as Host A during these time bins
- Query for all the hosts that had the *same behavioural path* as Host A: 0
  - No other host has been compromised in the same way of Host A, performed the same malicious activities *and* came back to cluster 7 over the examined time frame

We strongly believe that by answering to these question, the hunter could have a huge advantage in investigating the incident. For example, in this case, the hunter could focus her attention to the 3 hosts that started from a defined cluster, stayed singular for the following 4 time bins, and came back to a defined cluster in the last time bin, in order to see if the same threat actor compromised also the other 2 hosts.

We can proceed now to analyze the related work and underline the differences with the present research.

Chapter 7

Related work

In this chapter we proceed to analyze the existing academic work related to traffic analysis based on machine learning techniques, underlining the differences with the present work.

Li *et al.* [16] used two different *supervised* machine learning approaches to classify network traffic: On-Line Support Vector Machines and Decision Trees. The model was trained with data crawled from a *Microsoft Active Directory* server, which constituted a *ground truth* for the supervised models. Similarly, Shubair *et al.* [1] discussed an SVM based method to detect the network scan behaviour of worms.

Both these works used supervised models, while it is clear that in our scenario the system has to be deployed in a network where no ground truth is provided, therefore a supervised model is not applicable. Instead we apply a clustering algorithm to separate hosts into different categories, without prior knowledge about how many different categories are present in a network.

Wei *et al.* [19] discussed the possibility to use a *bottom-up clustering* strategy to separate the hosts into different categories. The use of an *unsupervised* model brings this research closer to our scenario. The main differences between [19] and the present work are:

- **Clustering strategy:** we chose a clustering algorithm that is *noise-aware*, i.e. that is able to effectively identify outliers in our dataset. The necessity of

being able to spot *singular* hosts in a network is essential in an investigation after an incident.

- **Feature selection:** although our approach was tested with aggregate metrics such as the number of sent and received bytes over a defined time bin, we argue that the use of *distributions* provides a more comprehensive overview of the host behaviour, maintaining a high sensitivity to behavioural changes.

For the aforementioned reasons, we claim that our approach does not overlap with the work of Wei *et al.*

Xu *et al.* [21] discussed about the opportunity to model *host communities* by means of *bipartite graphs* and aggregate them into *one-mode projection graphs*. The resulting weighted adjacency matrix gives an overview of communities, where by community the authors meant a series of hosts that happen to contact the same set of destination hosts. They then applied a *spectral clustering* algorithm to categorize the hosts, by using the weights of the adjacency matrix as a measure of distance between two data points. We argue that our approach, based on *distributions* of selected attributes of the traffic data, is capable of capturing more information than counting the common destinations between hosts. This is because our approach considers the distribution as a whole, allowing the distance metric to consider also the differences between the feature values.

Himura *et al.* [9] described a clustering approach based on *graphlets*. Since the use of graphlets is also part of our work (although mostly for visualization purposes), we will go into the details of how a graphlet is constructed in Chapter 3. By extracting features from graphlets, the authors applied clustering techniques to group similar hosts together, extracting a mean of visualization which they called *synoptic graphlet*.

Similar work has been done by Dewaele *et al.* [7], where they again use aggregated features such as entropy calculated on specific bytes of the destination IP addresses to categorize hosts. They applied *MST clustering* to divide hosts into categories.

Lakhina *et al.* [15] discussed the possibility of mining anomalies based on entropy values. They applied two different clustering techniques: *k-means* and *bottom-up*

clustering to categorize the hosts. They argue that their system is capable of spotting traffic anomalies even when they constitute a small portion of the data, differently from volume-based anomaly detection.

In the aforementioned works, aggregated measures have been used. After attempting the same type of measures with our DBSCAN algorithm, we concluded that a feature set based on distributions is more reliable in providing a meaningful grouping of hosts.

Winter *et al.* [20] discussed about an *inductive IDS*, i.e. a model trained exclusively with malicious data. They trained a One class Support Vector Machine to separate benign traffic from malicious one. One class SVMs are capable of dividing the data in two classes: either similar to the training set or different from it. In our scenario, where the categorization of hosts is important for network security monitoring purposes, the possibility of clustering hosts into more than two categories is essential.

Kruegel *et al.* [13] suggested the use of *distributions* to perform detection. They studied the character distribution of the *user agent* string in web requests to detect suspicious activities. The distribution they derived was though sorted in descending order, they considered the *shape* of the curve, while our distributions consider also the value of the feature that we are using.

Finally, Kind *et al.* [12] described an *histogram based* approach to anomaly detection. The use of an histogram based model was the main starting point of our work. While their suggestion was to consider the *hamming distance* between two bitmaps that represented the shape of the histograms, we chose to use the *Jensen-Shannon Divergence* as a distance metric to cluster hosts. More details about this metric will be given in Chapter 3.

To the best of our knowledge, the use of distributions as features and the Jensen-Shannon Divergence as metric to perform clustering of data points has never been applied in the field of network anomaly detection to categorize hosts, and this motivated the continuation of our work in this direction.

Having discussed the related work, we can now move to the conclusion of the document, summarizing the contribution of this thesis and providing insights on future work.

Chapter 8

Conclusions and future work

8.1 Conclusions

This work envisions and describes MANTIS, a machine learning approach to the problem of host categorization in a network. MANTIS is a system that is applicable on medium sized corporate networks and is capable not only of providing useful insights to the network operator under normal condition, but also helping an investigator during or after a so-called *data breach*.

MANTIS relies on an *unsupervised* machine learning technique, which is a density-based clustering, exploiting the *distribution* of features as data points and the *Jensen-Shannon divergence* as distance metric. We tested our method on real world data and concluded that it is indeed possible to use MANTIS in a production system for real customers for helping the so called *threat hunting* process, which still relies on human intervention in any known system in the market.

8.2 Future work

Due to the lack of time, we had to compromise on the number of different experiments that we did. We left as part of our future work to take this system one step further and effectively apply it as an automated alerting system.

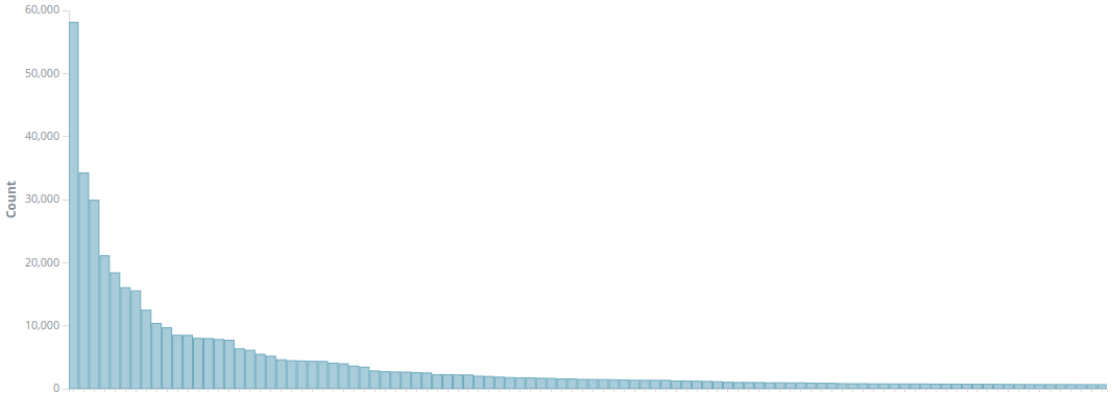

FIGURE 8.1: Distribution of the user agent strings

8.2.1 Integration of other data feeds

One important aspect that we underlined in the previous Chapters is the amount of data that Lastline has, which translates in an enormous amount of possibilities for feature selection. The first data feed we plan to integrate is *Web request records*, specifically extracting the *user agent* string. As stated before, we claim that the user agent string plays an essential role in the characterization of the applications running on a system. We extract the distribution of user-agent strings as we did for countries and port with PCR value, subsequently extending the definition of D_{final} given in Chapter 4.

As shown in Figure 8.1, the distribution follows a heavy-tailed shape, as happens with the destination ports. We expect to be applying the same aggregation technique we did with for the ports for performance reasons.

8.2.2 MANTIS as an alerting system

Provided that the considerations in the previous Chapters are still valid and we do not plan on substituting the existing system entirely with MANTIS, we are convinced that our approach can be improved in order to alert on some of the interesting *migrations* that we discussed in Chapter 4. This will require to:

- Define a way to rank these *migrations* by importance
- Apply a rate limiting algorithm on the number of alerts that can be generated and alert only on the most important ones

On the cluster quality side, we plan to improve our existing algorithm in order to adapt to situations in which the disposition of the points in the $len(D_{final}) - dimensional$ space makes the quality measures drop. This can be done by simply reducing the parameter ε of the DBSCAN algorithm in an adaptive way, finding a (possibly only local) maximum of the Calinski-Harabasz score for that particular time bin.

Appendix A

Listings

```
from __future__ import division

import collections
import geoip2.database
import geoip2.errors
import geopy.distance
import ipaddr
import networkx as nx
import scipy.stats


class HostStats(object):
    """
    Class containing the statistics for a specific host.

    Store all the informations needed to calculate the entropies and to
    construct the graphlet of a host.
    """

    def __init__(self, host):

        self._host = host

        # Store the graph
        self._graph = nx.DiGraph()
        self._graph.add_node(self._host)

        # Store all the different protocols used by the host
        self._protocols = collections.defaultdict(int)

        # Store all the different source ports used by the host
        self._src_ports = collections.defaultdict(int)
```

```
        # Store all the different destination ports contacted by the host
        self._dst_ports = collections.defaultdict(int)

        # Store all the different IPs contacted by the host
        self._dst_ips = collections.defaultdict(int)

        # Store all the different IP networks contacted by the host
        self._dst_nets = collections.defaultdict(int)
        self._dst_countries = collections.defaultdict(int)

        # Store the minimum timestamp
        self._min_ts = None
        # Store the maximum timestamp
        self._max_ts = None

        # Store the number of flows
        self._flows = 0

        # Store the number of established flows
        self._established_flows = 0

        # Store the total number of bytes
        self._total_bytes = 0

    @property
    def protocols(self):
        return self._protocols

    @property
    def src_ports(self):
        return self._src_ports

    @property
    def dst_ports(self):
        return self._dst_ports

    @property
    def dst_ips(self):
        return self._dst_ips

    @property
    def dst_countries(self):
        return self._dst_countries

    @property
    def dst_nets(self):
        return self._dst_nets
```

```
@property
def dst_ports_entropy(self):
    return self._entropy(self._dst_ports)

@property
def dst_ips_entropy(self):
    return self._entropy(self._dst_ips)

@property
def number_of_flows(self):
    return self._flows

@property
def failed(self):
    return 1 - (self._established_flows / self._flows)

@property
def avg_bytes(self):
    return self._total_bytes / self._flows

@property
def non_web(self):
    return sum([v for k, v in self._dst_ports.iteritems() if k not in {80,
        443}]) / self._flows

@property
def dst_nets_entropy(self):
    return self._entropy(self._dst_nets)

@property
def activity_time(self):
    return 0 if not self._min_ts or not self._max_ts else (
        self._max_ts - self._min_ts).total_seconds()

def in_degree(self, nodes):
    return self._graph.in_degree(nodes).values()

def out_degree(self, nodes):
    return self._graph.out_degree(nodes).values()

def _entropy(self, items_numbers):
    tot_number = sum(items_numbers.values())
    items_distr = {}

    for item, number in items_numbers.iteritems():
        items_distr[item] = number / tot_number
```

```
        return scipy.stats.entropy(items_distr.values())

    def process(self,
                protocol,
                src_port,
                dst_port,
                dst_ip,
                packets,
                data,
                established,
                dst_country,
                start_ts):
        """
        Process the stats of a packet/netflow and construct the graphlet.
        """
        # add the protocol and connect
        self._graph.add_node(protocol)
        self._graph.add_edge(self._host, protocol)

        # add the source port and connect
        self._graph.add_node(src_port)
        self._graph.add_edge(protocol, src_port)

        # add the destination port and connect
        self._graph.add_node(dst_port)
        self._graph.add_edge(src_port, dst_port)
        # self._graph.add_edge(protocol, dst_port)

        # add the destination country
        dst_net = ipaddr.IPNetwork((dst_ip, 16)).masked()
        if dst_ip.is_private:
            # for private addresses we want a higher granularity
            self._graph.add_node(dst_ip)
            self._graph.add_edge(dst_port, dst_ip)
        else:
            self._graph.add_node(dst_net)
            self._graph.add_edge(dst_port, dst_net)

        # gather statistics on the node:
        # i.e. how many times did we see that particular item on the network
        # number of flows++
        self._flows += 1

        # is the flow established?
        if established:
            self._established_flows += 1

        # log the protocol
```

```
        self._protocols[protocol] += 1

        # log the source port
        self._src_ports[src_port] += 1

        # log the destination port
        self._dst_ports[dst_port] += 1

        # log the destination ip
        self._dst_ips[dst_ip] += 1

        # log the destination network
        self._dst_nets[dst_net] += 1

        self._dst_countries[dst_country] += 1

        # sum the bytes transferred to the total number of bytes
        self._total_bytes += data[0] + data[1]

        if not self._min_ts or self._min_ts > start_ts:
            self._min_ts = start_ts
        if not self._max_ts or self._max_ts < start_ts:
            self._max_ts = start_ts

    def save_graph(self, path):
        try:
            nx.drawing.nx_agraph.write_dot(self._graph, path + str(self._host))
        except TypeError:
            print "OUCH!, {} made me break!".format(str(self._host))
```

host_statistics.py

```
from __future__ import division

import collections
import js
import numpy as np
import simplejson
import sys

from sklearn.cluster import DBSCAN
from sklearn.metrics import calinski_harabaz_score
from sklearn.metrics import silhouette_score
from sklearn.metrics.pairwise import pairwise_distances

DIST_THRESHOLD = 0.1


def main():
    iteration = int(sys.argv[1])

    with open("stats/stats", "r") as f:
        data = simplejson.loads(f.readline())
        columns = simplejson.loads(f.readline())

    X = np.array(
        [list(np.concatenate((np.array(hist["countries"]) / 2,
            np.array(hist["ports"]) / 2)))
            for hist in data.values()])

    silouhette = -1
    ch_score = -1
    eps = 0.01

    db = DBSCAN(
        eps=eps,
        min_samples=5,
        metric=js.js_div,
        algorithm="ball_tree",
        leaf_size=1,
        n_jobs=4).fit(X)

    labels = db.labels_

    n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
    sys.stdout.write("Hour: {}\nEstimated number of clusters: {}\n".format(
        iteration, n_clusters_))

    silouhette = silhouette_score(X[labels != -1], labels[labels != -1],
        js.js_div)
```

```
    ch_score = calinski_harabaz_score(X[labels != -1], labels[labels != -1])
    sys.stdout.write("Silhouette coefficient: {}\n".format(silouhette))
    sys.stdout.write("CH score: {}\n".format(ch_score))

    host_map = {host: label for host, label in zip(data.keys(), labels)}

    cluster_map = {}
    for label in (set(labels) - set([-1])):
        class_member_mask = (labels == label)
        dists = X[class_member_mask]
        cluster_map[label] = {str(column): value for column, value in zip(
            columns, list(np.mean(dists, 0)))}

    if iteration > 0:
        # Let's map the new clusters to the previous ones
        associations = collections.defaultdict(dict)

        distance_threshold = find_threshold(cluster_map.values())

        for label, mean in cluster_map.iteritems():
            hour = 1
            closest_label, previous_hosts = find_closest(mean, iteration - 1,
                distance_threshold)
            if not closest_label and iteration > 1:
                closest_label, previous_hosts = find_closest(
                    mean, iteration - 2, distance_threshold)
                hour = 2
            if closest_label:
                associations[label]["label"] = closest_label
                associations[label]["hosts"] = list(previous_hosts)
                associations[label]["hour"] = hour

        with open("stats/associations_{}".format(iteration), "w") as f:
            f.write(simplejson.dumps(associations))

    with open("stats/clusters_{}".format(iteration), "w") as f:
        f.write(simplejson.dumps(host_map))
        f.write("\n")
        f.write(simplejson.dumps(cluster_map))
    return 0


def find_closest(mean, hour, threshold):
    closest_label = None
    min_dist = 1

    with open("stats/clusters_{}".format(hour), "r") as f:
        previous_host_map = simplejson.loads(f.readline())
```

```
        previous_cluster_map = simplejson.loads(f.readline())

    for prev_label, prev_mean in previous_cluster_map.iteritems():
        joint_columns = set(mean.keys() + prev_mean.keys())
        mean_distr = [mean.get(key, 0.0000001) for key in joint_columns]
        prev_mean_distr = [prev_mean.get(key, 0.000001) for key in joint_columns]
        dist = js.js_div(np.array(mean_distr), np.array(prev_mean_distr))
        if dist < min_dist:
            closest_label = prev_label
            min_dist = dist

    if min_dist > threshold:
        return None, None

    previous_hosts = None
    if closest_label:
        previous_hosts = set(
            [host for host, label in previous_host_map.iteritems() if str(label)
                == str(
                closest_label)])
    return closest_label, previous_hosts


def find_threshold(means):
    """
    Define a threshold based on the current status of the clusters.

    Instead of defining a minimum threshold distance which is fixed, it is
    better to inspect the current status of the clusters (i.e. the distances
    between the current centroids) and define a distance based on that.
    The threshold distance should be smaller than the minimum distance observed
    in the current time bin. Clusters closer than this distance should be
    considered associated, clusters further that this threshold should not be
    associated.
    """

    centroid_means = [x.values() for x in means]
    X = np.array(centroid_means)
    distances = pairwise_distances(X)
    np.place(distances, distances == 0, 1)
    return np.amin(distances, axis=(0, 1))


if __name__ == "__main__":
    sys.exit(main())
```

cluster.py

Bibliography

[1] S. A. Abdulla, S. Ramadass, A. Altaher, and A. A. Nassiri. Setting a worm attack warning by using machine learning to classify netflow data. *International Journal of Computer Applications*, 36(2):49–56, December 2011.

[2] P. Aitken, B. Claise, and B. Trammell. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, Sept. 2013.

[3] T. Caliński and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics-Simulation and Computation*, 3(1):1–27, 1974.

[4] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, Oct. 2004.

[5] B. Claise and B. Trammell. Information Model for IP Flow Information Export (IPFIX). RFC 7012, Sept. 2013.

[6] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805 – 822, 1999.

[7] G. Dewaele, Y. Himura, P. Borgnat, K. Fukuda, P. Abry, O. Michel, R. Fontugne, K. Cho, and H. Esaki. Unsupervised host behavior classification from connection patterns. *International Journal of Network Management*, 20(5):317–337, Sept. 2010.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[9] Y. Himura, K. Fukuda, K. Cho, P. Borgnat, P. Abry, and H. Esaki. Synoptic graphlet: Bridging the gap between supervised and unsupervised profiling of host-level network traffic. *IEEE/ACM Transactions of Networking*, 21(4):1284–1297, Aug. 2013.

[10] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. *SIGCOMM Computer Commununication Review*, 35(4):229–240, Aug. 2005.

[11] R. A. Kemmerer and G. Vigna. Intrusion detection: A brief history and overview. *Computer*, 35(4):27–30, Apr. 2002.

[12] A. Kind, M. P. Stoecklin, and X. Dimitropoulos. Histogram-based traffic anomaly detection. *IEEE Transactions on Netwetworking*, 6(2):110–121, June 2009.

[13] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 251–261, 2003.

[14] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematics*, 22(1):79–86, 03 1951.

[15] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. *SIGCOMM Computer Communication Review*, 35(4):217–228, Aug. 2005.

[16] B. Li, M. H. Gunes, G. Bebis, and J. Springer. A supervised machine learning approach to classify host roles on line using sflow. In *Proceedings of the First Edition Workshop on High Performance and Programmable Networking*, pages 53–60, 2013.

[17] P. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, Nov. 1987.

[18] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[19] S. Wei, J. Mirkovic, and E. Kissel. Profiling and clustering internet hosts. In *Proceedings of the SIAM International Conference on Data Mining*, 2006.

[20] P. Winter, E. Hermann, and M. Zeilinger. Inductive intrusion detection in flow-based network data using one-class support vector machines. In *NTMS*, pages 1–5, 2011.

[21] K. Xu, F. Wang, and L. Gu. Network-aware behavior clustering of internet end hosts. In *Proceedings INFOCOM*, pages 2078–2086, 2011.