POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Thesis

Smart intrusion detection systems based on machine learning

Supervisors
Prof. Antonio Lioy
Ing. Daniele Canavese
Ing. Leonardo Regano

Candidate
Matteo Coscia

Academic year 2020-2021

Contents

1 Introduction 6

2 Network traffic statistics 9

2.1 Wireshark . . . 9

2.2 Tstat . . . 11

2.2.1 Extracted statistics . . . 13

2.3 CICFlowMeter . . . 14

2.4 Tools comparison . . . 14

3 Network attacks 17

3.1 Bruteforce attack . . . 17

3.1.1 Attack description . . . 17

3.1.2 Attack tools . . . 19

3.2 Denial of Service . . . 20

3.2.1 Attack description . . . 21

3.2.2 Attack tools . . . 23

3.3 Web attack . . . 24

3.3.1 Attack description . . . 24

3.3.2 Attack tools . . . 28

3.4 Botnet . . . 28

3.4.1 Attack description . . . 29

3.4.2 Attack tools . . . 30

4 Intrusion Detection Systems 31
4.1 Detection methodologies . . . 31
4.1.1 Signature-based Detection . . . 31
4.1.2 Anomaly-based Detection . . . 32
4.2 Detection approaches . . . 34
4.3 Technology types . . . 35
4.3.1 Host-based IDS . . . 35
4.3.2 Network-based IDS . . . 36
4.3.3 Wireless-based IDS . . . 37
4.4 Popular IDSes . . . 38
4.4.1 Suricata . . . 39
4.4.2 Snort . . . 45
4.4.3 Zeek . . . 46

5 Machine Learning and classification techniques 48
5.1 Machine Learning . . . 48
5.1.1 Dataset analysis and features selection . . . 49
5.1.2 Dataset preprocessing . . . 49
5.1.3 Model selection . . . 50
5.1.4 Training, validation and testing . . . 52
5.1.5 Evaluation metrics . . . 54
5.2 Classification techniques . . . 56
5.2.1 Decision tree . . . 56
5.2.2 Random forest . . . 57
5.2.3 Support Vector Machines . . . 58
5.2.4 Neural Networks . . . 59

6 Solution design 63
6.1 Creation and selection of classifiers . . . 63
6.1.1 Extraction of statistics . . . 64
6.1.2 Features selection . . . 64
6.1.3 Training and Validation . . . 66
6.1.4 Testing . . . 68
6.2 IDS integration . . . 69
6.2.1 Suricata initialisation . . . 70
6.2.2 Packet decoding . . . 71
6.2.3 Packet classification . . . 71

7 Dataset analysis 73
7.1 Dataset composition . . . 73
7.1.1 CSE-CIC-IDS2018 . . . 73
7.1.2 Torsec dataset . . . 74
7.2 Traffic analysis . . . 75

8 Experimental results 77
8.1 Classification results . . . 77
8.1.1 Random Forest . . . 77
8.1.2 Support Vector Machine . . . 79
8.1.3 Neural Network . . . 79
8.1.4 Comparison between algorithms . . . 80
8.2 Classification time . . . 80
8.3 Suricata integration . . . 81

9 Related works 84
9.1 Machine Learning and IDS . . . 84
9.2 Machine Learning and traffic analysis . . . 88

10 Conclusions 92

A User manual 94
A.1 Requirements . . . 94
A.2 Installation . . . 96
A.3 Usage . . . 96
A.3.1 Configuration . . . 96
A.3.2 Creation of new signatures . . . 98

B Developer manual 100
B.1 Requirements . . . 100
B.2 Installation . . . 100
B.3 Code usage . . . 101
B.3.1 Dataset scripts . . . 101
B.3.2 Creation and selection of classifiers . . . 102
B.3.3 Suricata integration code . . . 104
B.3.4 Selection of Tstat statistics . . . 105

Bibliography 106

Chapter 1

Introduction

The network threats landscape changes at an increasingly faster pace and the failure in mitigating such threats has a twofold consequence: an economic loss for the victims and an infrastructure cost, in terms of bandwidth and throughput, for the whole network. A report of Netscout[1] about the first half of 2020 highlights a 15% increase of cyber-attacks with respect to 2019, for a total of 4.83 millions of DDoS attacks observed by the company and a total 10 millions at the end of 2020[2]. Moreover, the same Netscout report states that the attackers shifted their targets towards fundamental services during the COVID-19 pandemic, such as E-commerce, healthcare and educational services.
However, not only the average number of attacks increased during the first pandemic lock-down, with a spike in the number of attacks between April and May 2020, but the type of attacks has changed and adapted to the new targets. The average attack duration has reduced by around 50%, with an increase in complexity and strength of the attack: DDoS attacks that exploits a single attack vector, i.e. the exploitation of a flaw in a network protocol or software, has decreased by 43% with respect to 2019, while the number of attacks exploiting more than 15 vectors at the same time has increased by more than 2.800% in the last couple of years. The same report shows that the preferred attack vector is the DNS protocol, counting more than 1 million of attacks alone and an amplification factor of 160:1 for DDoS amplification attacks.

However, DDoS attacks are not the only popular cyber-attack type: web application attacks like SQL injection or cross-site scripting (XSS) are as much popular and widespread among attackers, if not more. In fact, the European Union Agency For Cybersecurity (ENISA) threat landscape report[3] of 2020 lists web-based and web application attacks in the top four positions of the most recurrent and dangerous attacks, while DDoS attacks and botnets can be found at the sixth and tenth position respectively. Moreover, the Akamai company provides a periodic report[4]

[1] https://www.netscout.com/threatreport

[2] https://www.netscout.com/blog/asert/crossing-10-million-mark-ddos-attacks-2020

[3] https://www.enisa.europa.eu/news/enisa-news/enisa-threat-landscape-2020

[4] https://www.akamai.com/uk/en/resources/our-thinking/state-of-the-internet-report/web-attack-visualization.jsp

of the web attacks it observes inside networks monitored by its systems. The report that covers the week between February 7 2021 and February 14 2021 states that over 145 millions of web attacks attempts have been observed. The vast majority of these attacks is composed by SQL injection, accounting for 60% of the total, while XSS attacks reach almost the 18%.

Hence, the typical countermeasures for these types of cyber-attacks have to rapidly adapt to this ever-changing landscape. Intrusion Detection and Prevention Systems (IDS, IPS or IDPS) are one of the possible tools that are used, together with firewalls, to mitigate the effectiveness of many cyber-attacks. An Intrusion Detection System (IDS) is the tool, software or hardware, that allows the automated detection of intrusions inside a monitored network, while the same tool is called Intrusion Prevention System (IPS) in case it also automates the countermeasures against these intrusions. These tools can operate in different positions of the network and are typically split into Network-based IDS (NIDS) and Host-based IDS (HIDS): a NIDS monitors a portion of the network by analysing the passing traffic, while a HIDS monitors the activities of the single host where it is deployed. Moreover, traditional network-based IDS solutions, like Snort[5] or Suricata[6] are typically based on the concept of "signature" to identify threats occurring in the network they are monitoring. Signatures can be described as sets of characteristics of a network packet or aggregates of packets (e.g. TCP flows) that signal the presence of an ongoing attack. However, signatures are typically hand-crafted by experts, which base the patterns of a signature on the knowledge of existing attacks and known exploits. For example, the Emerging Threats[7] list contains a set of publicly available signatures compatible with the syntax used by Snort and Suricata. Even if the signature-based type of approach yields very good results in the detection of known attacks, the rapidly-changing threats landscape makes difficult to constantly adapt and create new signatures to detect cyber-attacks. Furthermore, the variation of some patterns inside a known attack may be enough to avoid the detection of an IDS: a Malwarebytes 2019 report[8] describes the possibility of a malware capable of dynamically adapting its attack pattern to avoid detection, by using artificial intelligence and machine learning techniques.

A possible solution to solve the shortcomings of classic signature-based detection is to integrate their detection capabilities with automatically adapting detection engines, based on machine learning classifiers. By using machine learning based techniques, it is possible to create classifiers able to take advantage of patterns inside the malicious traffic, to detect intrusions without manually creating a signature for each new attack type. Moreover, the resulting IDS gains the ability of generalising the characteristics of a family of threats, leading to an easier detection of previously unknown attacks or modified patterns.

[5] https://www.snort.org/

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

[7] https://rules.emergingthreats.net/

[8] https://resources.malwarebytes.com/files/2019/06/Labs-Report-AI-gone-awry.pdf

With the improvement of signature-based detection as objective, this work proposes a pipeline for the creation, selection and integration of machine learning classifiers inside an existing IDS, Suricata. The creation and selection of these ML models has been performed using a dataset containing some of the most widespread cyber-attacks: DoS and DDoS attacks, botnets, bruteforce and web-based attacks. Due to the objective of integrating the classifiers inside a working IDS, different algorithms have been compared both in terms of classification capabilities and in terms of time performances, to obtain an optimal solution for each different type of attack. The adopted ML algorithms have been random forests, support vector machines and neural networks. Then, the best models obtained have been integrated inside Suricata, modifying the existing signature syntax with the addition of ad-hoc keywords to perform the ML-based classification. Moreover, in order to obtain the networks statistics of interest to train the classifiers and performs the predictions, the Tstat[9] tool has been employed and integrated inside Suricata and its functionalities have been expanded as necessary.

Finally, the following chapters cover both the theoretical aspects of this approach and the analysis of the adopted solution and the obtained results. More in detail, Chapter 2 introduces the concept of network traffic statistics and the possible techniques used to obtained them; Chapter 3 presents an overview of the network attacks used for this work's experiments, together with the main tools to performs these attacks; Chapter 4 proposes a taxonomy of intrusion detection systems, the different approaches existing and an overview of the actual software solutions that can be found, including Suricata; finally, Chapter 5 ends the theoretical section of this work with the description of a typical machine learning workflow and the technical explanation of the classification algorithms used.
Then, Chapter 6 illustrates the design of the proposed solution, composed by the creation and selection of machine learning models and the subsequent integration inside the Suricata IDS; Chapter 7 analyses the characteristics of the used dataset, while Chapter 8 lists the results obtained during the experiments with the proposed solution. At last, Chapter 9 performs a comparison of other existing similar solutions, by describing them and highlighting differences and similarities, while Chapter 10 sums up the workflow of this work and the obtained result, pointing out some limitations of the proposed solution and suggesting some future improvements. Appendix A and Appendix B contain the manuals to use the provided solution and to expand it, from the perspective of a final user or a developer, respectively.

[9] http://tstat.polito.it/

Chapter 2

Network traffic statistics

This chapter contains a description of the statistics that can be extracted analysing network traffic and the tools that can be used to perform such analysis. In general, network traffic measurement and monitoring is a methodology aimed at understanding packet traffic on the Internet. It serves as the basis for a wide range of IP network operations and engineering tasks such as trouble shooting, accounting and usage profiling, routing weight configuration, load balancing, capacity planning, and more. Two approaches are possible: active or passive [1]. The active approach aims at interfering with the network to induce a measurable effect. An example of such interference could be the alteration of network state by the enforcement of artificial packet loss. On the other hand, the passive approach does not interfere with the network: a pure observation is performed by means of dedicated tools, commonly named "sniffers".

There are several tools widely available that can capture packets at various levels of the OSI model[1], performing a passive analysis. The most common probably are Tcpdump[2] or Wireshark[3], while other tools like Tcptrace[4] are able to accept already captured packets as input, in various formats.
The following sections describe some of the candidate tools to be used for this work, while the last section contains a comparison between these tools and the explanation for the final choice.

2.1 Wireshark

Wireshark is a network packet analyser. Its functionalities can be used, for example, to troubleshoot network problems, examine security problems or debug protocol implementations. It is available for both UNIX and Windows systems and it can capture live packet data from a network interface, import packets from text files

[1] https://www.iso.org/standard/20269.html

[2] https://www.tcpdump.org/

[3] https://www.wireshark.org/

[4] https://linux.die.net/man/1/tcptrace

containing hex dumps of packet data, save packet data captured, filter packets on many criteria and create various statistics. Since the scope of this chapter concerns network traffic statistics, a detailed overview of the ones produced by Wireshark is provided.

The statistics section[5] of Wireshark is split into general ones, containing information about the capture files, or into protocol-specific statistics. The general section can be further split into:

Capture File Properties various properties of the capture file;

Protocol Hierarchy details about the protocol hierarchy of the captured packets;

Conversations statistics about traffic between specific IP addresses;

Endpoints statistics about traffic to and from an IP addresses;

I/O Graphs various graphs to visualise packets statistics over time.

The "Capture File Properties" section contains information about a specific capture file, like the name, the length, the duration and the hardware used to perform the capture. Moreover, it provides some general statistics like the number of captured packets, the average number of packets per second, the average packet size, the total number of bytes and the average number of bytes per second.

Instead, the "Protocol Hierarchy" section provides the hierarchy structure of the protocols found in the analysed packets, together with some additional statistics. The hierarchy stars from the outermost layer, a frame of the data link layer, and proceeds with the inner protocols of the packet. A typical example of a hierarchy, starting from a frame, could be the Ethernet protocol, then IPv4, TCP and HTTP. For each one of the protocols in the hierarchy, Wireshark lists the percentage of packets with respect to other protocols at the same level (e.g. 60% IPv4, 40% IPv6), the absolute number of packets, the number of bytes and the average number of bits per second.

Then, the "Conversations" section contains statistics about traffic between pairs of IP addresses, while the "Endpoints" section contains statistics about traffic to and from a specific endpoint. The statistics provided for both sections are very similar and include the total number of packets to and from the selected addresses, the number of bytes the average number of bits per second and, only for the "Conversations", the duration of the flow between the two addresses. Wireshark supports many types of endpoints, like Bluetooth, Ethernet, WiFi, IPv4, IPv6, TCP, UDP, Token Ring and others.

Finally, the "I/O Graphs" section allows to plot packet and protocol data in a variety of ways. It contains a chart drawing area along with a customizable list of graphs, divided into time intervals. Moreover, clicking on the graph takes the user to the associated packet in the packet list.

[5] https://www.wireshark.org/docs/wsug_html/#ChStatistics

2.2 Tstat

TCP Statistics and Analysis Tool[6] (Tstat) is an automated tool for passive monitoring. It has been developed by the networking research group at Politecnico di Torino since 2000 and it offers live and scalable traffic monitoring up to gigabits per second. Tstat started as an evolution of Tcptrace, by automating the collection of TCP statistics adding real-time traffic monitoring features, but it evolved over the years, offering more statistics and functionalities. Since it is a passive tool, its typical usage scenario is the monitoring of Internet links, in which all flowing packets can be observed. An example of this setup is shown in Figure 2.1: Tstat monitors a link that connects the network of interest with the rest of the Internet, to see all the passing packets.

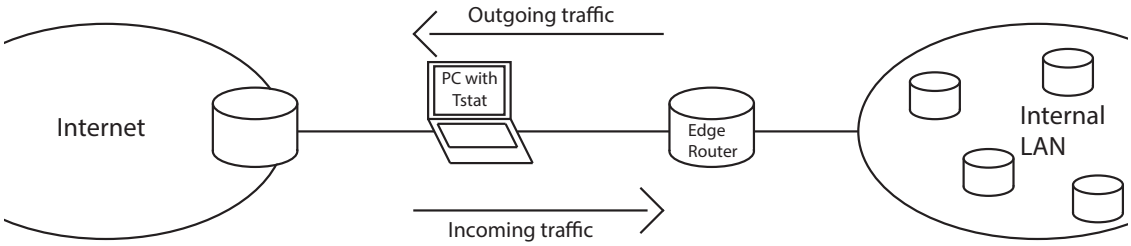

Figure 2.1: An example of Tstat setup.

The basic objects monitored by Tstat are IP packets transmitted on the monitored link. Flows are defined according to some rules: all packets identified by the same "flowID" that have been observed in a given time interval are grouped together. A common choice for the "flowID" is the tuple (ipProtoType, ipSrcAddr, srcPort, ipDstAddr, dstPort) so that TCP and UDP flows are considered [1]. For example, in the case of TCP, the start of a new flow is commonly identified when the TCP three-way handshake is observed; similarly, its end is triggered when either a proper TCP connection teardown is seen, or no packets have been observed for some time. While for UDP, a new flow is identified when the first packet is observed, and it is ended after an idle time. Moreover, opposite symmetric unidirectional flows are typically grouped together to compute separate statistics for "client-to-server" and "server-to-client" flows.

Furthermore, Tstat supports both real-time and non real-time analysis of packets. Real-time monitoring is obtained with the usage of libpcap, a library created by the same developers of the Tcpdump tool that is widely used to capture packets from Ethernet links under several operating systems. Aside from libpcap sniffing, Tstat supports common hardware solutions like Endace DAG[7] cards. Instead, non real-time monitoring is performed by analysing packets previously captured with other tools. The trace formats currently supported by Tstat are:

tcpdump public domain program from LBL;

[6] http://tstat.polito.it/

[7] https://www.endace.com/dag-10x2-s-datasheet.pdf

snoop distributed with Solaris;

etherpeek Mac sniffer program;

netmetrix commercial program from HP;

ns network simulator from LBL;

netscout NetScout Manager format;

erf Endace Extensible Record Format;

DPMI Distributed Passive Measurement Interface (DPMI) format;

tcpdump live Live capture using pcap/tcpdump library.

Moreover, also the produced output is customizable. Statistics are available at different granularities: per-packet, per-flow or aggregated. At the lowest level, packet traces can be dumped into trace files for further processing and packets from different applications can be dumped in different files. At an intermediate level, flow-level logs are generated and provide detailed information for each monitored flow. These logs file are organized as tables, where each column is associated with specific information and each row reports the two unidirectional flows of a connection. Several flow-level logs are available and their content and structure is better described in Section 2.2.1.
Finally, at the highest level of granularity, two formats are available: histograms and Round Robin Databases[8] (RRD). Histograms are frequency distributions of collected statistics over a set of flows: for example the distribution of the VoIP calls duration is computed by considering the VoIP flows observed during a certain interval. Instead, RRD allows to build a database that spans several years, by limiting the used disk space. RRD handles older data differently from newer samples. Newest data is stored at higher frequencies, while older data are averaged in coarser timescales. Moreover, the RRD tool allows for a graphical inspection of collected results.

Finally, a brief overview of the mechanisms behind Tstat functionalities are shown in Figure 2.2, which represents a block diagram of the trace analyser: these are the steps through which the program moves for each analysed packet [2].

Each packet is first analysed by the IP module, which takes care of all the statistics at the packet layer. Then, the next module distinguishes between TCP and UDP segments to compute per-segment statistics and the flow continues with the respective analyser. The TCP module decides if the segment belongs to an already identified flow, using the tuple described above. If not, then a new flow is identified only if the SYN flag is set, as Tstat processes only complete flows. Then, the flow statistics are updated with the new segment and in case the segment correctly closes a flow, the flow-layer output is produced and the flow data structure is released.

[8] https://oss.oetiker.ch/rrdtool/

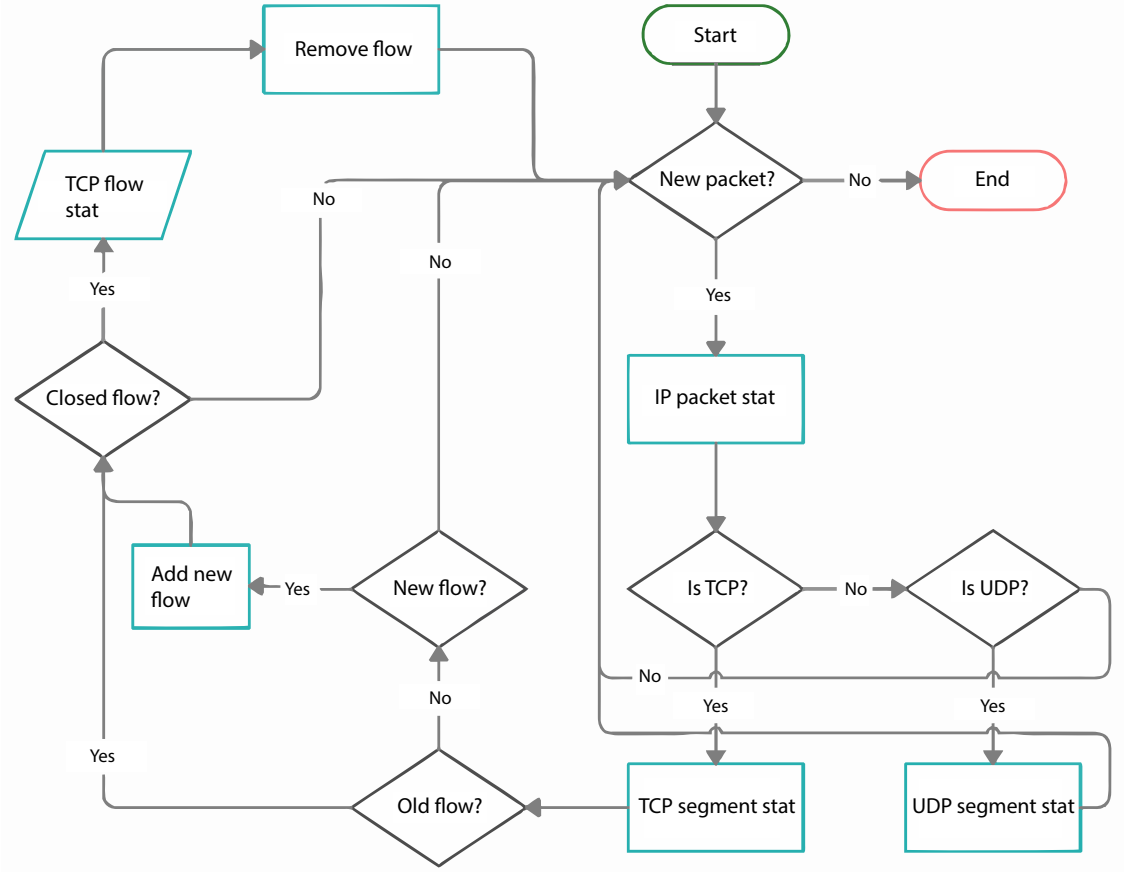

Figure 2.2: A schematic block diagram of Tstat trace analyser. Only the TCP analyser is detailed.

2.2.1 Extracted statistics

Tstat creates a set of TXT files where each row corresponds to a different flow and each column is associated to a specific measure. When it is useful, the columns are grouped according to "Client-to-Server" (C2S) and "Server-to-Client" (S2C) traffic directions. For most logs, the first row contains a summary with the description of all columns. The generated logs are "log_tcp_complete", "log_tcp_nocomplete", "log_udp_complete", "log_video_complete", "log_http_complete", "log_mm_complete", "log_skype_complete", "log_chat_complete" and "log_chat_messages".

The first two, log_tcp_complete and log_tcp_nocomplete, report every TCP connection that has been tracked by Tstat. A TCP connection is identified when the first SYN segment is observed and is ended when FIN/ACK or RST segments are observed or no data packet has been observed, from both sides, for a default timeout of 10s after the opening SYN segment, or 5min after the last data packet. Both timeout values can be customised. Tstat discards all the connections for which the three way handshake is not properly seen.
Both files have similar format with values separated by spaces. Each log is made by the composition of different measurements sets, which order is hard-coded as follows:

- Core TCP set;

- TCP End to End set (Optional);
- TCP P2P set (Optional);
- TCP Options set (Optional);
- TCP Layer7 set (Optional);
- TCP Advanced set (Optional).

However, the log_tcp_nocomplete file always contains only the Core TCP set. This set contains the basic information for all TCP flows and its content is described in Table 2.1. The first two columns represent the position inside the log of the client-to-server and server-to-client information respectively.

The other sets include specific measures, which usefulness is more situational. For example the TCP End to End set includes measures about round-trip-time (RTT) and time-to-live (TTL) for TCP connections. The complete list can be found in the Tstat website[9]. Finally, the detailed description of the steps to use Tstat can be found in Appendix B.

2.3 CICFlowMeter

CICFlowMeter[10] is a network traffic flow generator and analyser. It can be used to generate bidirectional flows, where the first packet determines the forward and backward directions. Moreover, it can compute more than 80 network traffic features, such as flow duration, number of packets, number of bytes and average length of packets. The complete list of extracted features can be found on the website of one of the datasets created with this tool, the CSE-CIC-IDS2018[11] dataset. Furthermore, this tool allows the addition of more features or the reduction of the existing ones. The output is produced in the CSV format. Finally, the tool is publicly available in a Git repository[12] and it is written in Java.

2.4 Tools comparison

The tool chosen for this work's experiments should have some specific characteristics: it should be able to produce a high number of network features that will be used to create the machine learning models, it should be able to provide statistics in real-time and its architecture should allow for an integration with an IDS, which is one of the objectives of this work.
Wireshark has been excluded because it does not satisfy any of these pre-requisites.

[9] http://tstat.polito.it/measure.shtml

[10] https://www.unb.ca/cic/research/applications.html#CICFlowMeter

[11] https://www.unb.ca/cic/datasets/ids-2018.html

[12] https://github.com/CanadianInstituteForCybersecurity/CICFlowMeter

As explained in Section 2.1, it can produce a very limited amount of statistics for bi-directional flows; moreover, it does not provide any API to retrieve statistics in real-time and the only way to obtain a CSV file of the produced ones is through the GUI or through the terminal-based version of Wireshark, Tshark[13].
Instead, the other two candidate tools, Tstat and CICFlowMeter, produce a considerable amount of network traffic statistics. The highest number of features is provided by the CICFlowMeter, that can compute around 80 features for bi-directional flows, while Tstat can produce almost 40 features for TCP flows. Moreover, the CICFlowMeter tool is the same tool that has been used to create the dataset chosen for this works's experiments, as explained in Chapter 7, and the authors of the dataset already provides the CSV files with the extracted statistics. On the other hand, the usage of Tstat would require the extraction ex-novo of the traffic statistics, starting from the provided packets captures.
However, the final choice has been the usage of Tstat, due to the last requirement of the list explained above: the integration with an IDS. The fact that Tstat is written in C, the same programming language used by the IDS chosen for this work, as explained in Section 6.2, allows for a seamless integration of the required functionalities inside the IDS.

[13] https://www.wireshark.org/docs/man-pages/tshark.html

C2S	S2C	Unit	Description
1	15	-	IP addresses of the client/server
2	16	-	TCP port addresses for the client/server
3	17	-	total number of packets observed form the client/server
4	18	0/1	0 = no RST segment has been sent by the client/server
5	19	-	number of segments with the ACK field set to 1
6	20	-	number of segments with ACK field set to 1 and no data
7	21	bytes	number of bytes sent in the payload
8	22	-	number of segments with payload
9	23	bytes	number of bytes transmitted in the payload, including rtx
10	24	-	number of retransmitted segments
11	25	bytes	number of retransmitted bytes
12	26	-	number of segments observed out of sequence
13	27	-	number of SYN segments observed, including rtx
14	28	-	number of FIN segments observed, including rtx
29		ms	flow first packet absolute time (epoch)
30		ms	flow last segment absolute time (epoch)
31		ms	flow duration since first packet to last packet
32		ms	client first segment with payload since the first flow segment
33		ms	server first segment with payload since the first flow segment
34		ms	client last segment with payload since the first flow segment
35		ms	server last segment with payload since the first flow segment
36		ms	client first ACK segment since the first flow segment
37		ms	server first ACK segment since the first flow segment
38		0/1	1 = client has internal IP, 0 = client has external IP
39		0/1	1 = server has internal IP, 0 = server has external IP
40		0/1	1 = client IP is CryptoPAn anonymized
41		0/1	1 = server IP is CryptoPAn anonymized
42		-	Bitmap stating the connection type, as identified by TCPL7 inspection engine
43		-	Type of P2P protocol, as identified by the IPP2P engine
44		-	For HTTP flows, the identified Web2.0 content

Table 2.1: Tstat Core TCP set.

Chapter 3

Network attacks

This chapter focuses on some of the most popular network attacks, with a particular emphasis on the typologies encountered during this work's experiments and further described in Section 7.1. It also presents a sum of the main tools used to perform such attacks.

3.1 Bruteforce attack

The bruteforce attack is one of the oldest and simplest attacks. It consists in multiple login attempts to a server, trying to find the correct credentials. Usually the username is already known or found through other methods.

3.1.1 Attack description

There are different techniques to execute this type of attack: the exhaustive attack is the simplest one, but other popular techniques are dictionary attacks and rainbow tables.
Exhaustive attack is quite straightforward: given a set of characters and a length range, it tries all the possible combinations until the right one is found. The set of characters can include lowercase and/or uppercase letters, digits and special symbols. A possible variation of this method consists in trying only specific words or variations of such words, starting from commonly used terms and names: for this reason, this technique is called dictionary attack.
NordPass, a password manager software, shows the most common passwords of 2020[1] and they are easily cracked in less than a second with this simple type of attack: the most used one seems to be "123456", followed by the way more secure "123456789", probably used for forms that require longer passwords. On the fourth place there is the evergreen "password", but scrolling down the list it is possible to find many popular names like "qwerty", "iloveyou", "unknown", "pokemon" or "michael".

[1] https://nordpass.com/most-common-passwords-list/

A smarter variation to bruteforce attacks, still used to guess credentials, starts from the hash of the password: given a known hash algorithm, the hash of each word in a set is pre-computed and stored with the hash as searching index. The set can be an exhaustive one or composed by a subset of common passwords, like above. Once an hashed password is captured on the network, a lookup function searches for that hash inside the dictionary and obtains the corresponding password associated. This type of attack is not properly bruteforce, since it does not try all the possible combinations, it simply stores them for a later use.

Given the same set of characters for both methods, the number of combinations is the same, of course. However, the bruteforce one has to compute and try during the attack the whole set of combinations, in the worst case. The second type is much faster at attack time, because it has only to lookup a data structure, given the index. The downside is that this pre-computed table has to be stored in memory.
The rainbow table method [3] is a trade-off between time and memory requirements: a chain of hashes is computed for a subset of the possible combinations. Starting from an element of this subset, the hash function is computed, followed by the computation of a reduction function, that brings the result back into the password domain (e.g. into fixed length alphanumeric words). This reduction function is not the reverse of the hash. The chain continues computing the hash over the reduction and so on, for a fixed amount of times. Only the beginning and the end of each chain are stored: in this way the table is much smaller.
The lookup of an hash is done in the following way: the reduction function is computed over the hash and then the hash is computed again over the reduction, similarly to the creation of the table process. This is repeated for a maximum number of times equal to the length of chains: if at any time during the process a match is found with an end of a chain in the table, the beginning of the chain is taken and the chain is reconstructed to find the hash and the corresponding password.
A simple example is shown in Figure 3.1 to better clarify this algorithm. The starting set is composed by three words: “123456”, “password” and “security”. The hash and reduction functions are computed and only the beginning and the ending of the chains are stored. Once the hash “67d23ry” is sniffed from the network, the reduction and hash functions are alternated until the end of a chain is found, the word “linux”. The beginning of the corresponding chain, “security”, is used to reconstruct the chain and the sniffed hash is found: the password that originated it is on the previous step, “paperino”.

Actually, using the same reduction function at each step, causes the collision and fusion of multiple chains, in case at different steps both chains produce the same output. To overcome this limitation and allow even longer chains, a different reduction function is used for each link in a chain, but keeping the same order for different chains. In this way, to have a complete chain fusion, the collision should happen at the exact same number of link in two different chains. If it happens, one of the colliding chains is pruned off.
Often, pre-computed rainbow tables for different hash functions and different sets of characters are sold on the black market, making this type of attack much easier.

However, in case of an offline attack, the attacker has all the time needed to

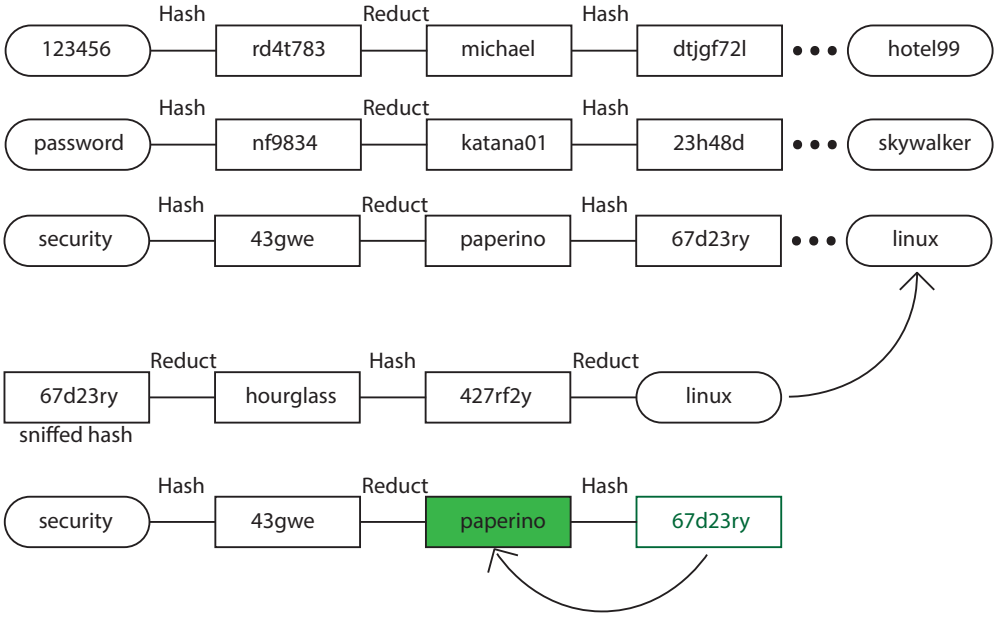

Figure 3.1: An example of rainbow table with singular reduction function.

bruteforce the credentials; instead, in an online attack, there are some basic countermeasures that can be taken. Against both techniques, it is possible to limit the number of attempts to login over a certain time, to increase the complexity of login or to introduce a timed two factor authentication (2FA). A typical example of a timed 2FA is the Timed One Time Password [4] (TOTP): in this case, after the insertion of the correct credentials, the server requires the insertion of a pseudo-random password, generated by an ad-hoc system. Possible examples of such generators are OTPs received by SMS or specific dedicated applications like Google Authenticator.
Furthermore, against the techniques based on hash tables, it is possible to add a salt to the password in order to force the re-computation of the hash. This simple technique can make impossible the usage of rainbow tables: the salt is a large sized word, possibly not the same for all the passwords of a server, that is added after the user's password. The hash is computed on the concatenation of the password and the salt, forcing the attacker to create a table for each possible salt. While with a 12 bit salt it would still be possible to compute and store 4096 different tables, a 128 bit salt would require too many years to compute all the possible tables (2^{128}).

3.1.2 Attack tools

Patator[2] is one of the many tools available to perform brute force attacks. It is not among the most common[3] and it is written in Python. It is able, to brute force SSH, FTP, SMTP, HTTP, POP3, DNS and various SQL servers.

[2] https://tools.kali.org/password-attacks/patator

[3] https://resources.infosecinstitute.com/topic/10-popular-password-cracking-tools/

For instance, Listing 3.2 shows a possible output of the execution of this tool, to find a FTP password for the user "root". In this case, a file containing a list of passwords is passed to the tool, that attempts to perform the login into the target FTP server by trying the provided passwords: the correct one is "ftp".

```
$ ftp_login host=10.0.0.1 user=root 0=logins.txt password=FILE0
   -x ignore:mesg='Login incorrect.' -x
   ignore,reset,retry:code=500
19:36:06 patator INFO - Starting Patator v0.9
19:36:06 patator INFO -
19:36:06 patator INFO - code size time | candidate | num | mesg
19:36:06 patator INFO - -----------------------------------------
19:36:07 patator INFO - 530 18 0.002 | anonymous | 7 |
   Permission denied.
19:36:07 patator INFO - 230 17 0.001 | ftp | 10 | Login
   successful.
19:36:08 patator INFO - 530 18 1.000 | root | 1 | Permission
   denied.
19:36:17 patator INFO - 530 18 1.000 | michael | 50 | Permission
   denied.
19:36:36 patator INFO - 530 18 1.000 | robert | 93 | Permission
   denied.
```

Figure 3.2: An example of patator output.

Other more common similar tools are THC Hydra, NCrack, John the Ripper, Rainbow Crack and many others. Rainbow Crack[4] is a software able to both create and lookup rainbow tables. It is available for both Windows and Linux and includes hardware acceleration capabilities, for both NVIDIA and AMD GPUs. Moreover, for less than 1000$, it is possible to buy the full set of MD5 rainbow tables from its website, shipped within a 2TB hard drive. The table containing alphanumeric characters from the set [a-z], [0-9] with plaintext long from 1 to 9 characters needs 65GB, while it increases to 690GB for the alphanumeric set [a-z], [A-Z], [0-9].

3.2 Denial of Service

The Denial of Service attack (DoS) is maybe one of the most known attacks. Its aim, rather than being that of accessing information, is to disrupt its availability. The underlying concept is to saturate the server's resources with packets, to negate the connection to legitimate users.

[4] https://project-rainbowcrack.com/

3.2.1 Attack description

First of all, Denial of Service attacks exist in two variations: standard DoS attacks or Distributed DoS attacks (DDoS). In this last case, the attacker infrastructure is distributed across several machines, that can be intentional attackers or not. In case this infrastructure is composed by a multitude of infected computers, this is called a botnet: it can execute commands received from the attacker, even at a delayed time, in order to attack the victim in different ways, as better explained in Section 3.4.
However, DoS attacks can fall under two groups: direct and reflection attacks [5]. Direct ones involve traffic sent to the victim directly from the attacker infrastructure. This can be a single machine or a set of servers. To avoid the attribution of the attack, usually the attackers employ random spoofing, a technique used to fake the IP address of packets. Instead, in reflection attacks, third party servers are used to reflect the traffic towards the victim. Moreover, many protocols that allow for reflection, also add amplification, causing the generated traffic volume to become several times bigger. These two groups of attacks are also referred to as volumetric, since are based on the volume of the requests to exhaust resources of the victim. Another possible type of DoS are semantic attack, that must be crafted ad hoc, because they exploit specific flaws in the victim infrastructure, for example to crash a server.

Some common direct DoS types are UDP and TCP flooding attacks. The UDP one is based on the fact that a host replies with an "ICMP Destination Unreachable" message in case it receives an UDP message towards a port where no service is listening. Sending a large amount of such UDP packets forces the host to keep replying with ICMP packets, thus depleting resources. This type of attack is quite easy to counter, by limiting the rate at which ICMP replies are sent.
Another simple, but effective, direct attack is SYN flooding [6]: it exploits a flaw in the three way handshake protocol for TCP connections. This protocol is composed by three simple steps:

1. the client sends a SYN message to the server;
2. the server answers with a SYN-ACK message;
3. the client answers with an ACK message, followed by data.

If the client does not answer with the last ACK message, the server awaits some time before freeing the resources held for that possible connection. So, if the server receives too many SYN messages without the last ACK, maybe from fake IP sources through IP spoofing, it runs out of resources, slowing down or crashing directly.
The most basic method to fight this type of DoS attacks is to shorten the timeout for the half-opened connections or to dynamically dimension the queue for new connections.

On the other hand, reflection with amplification attacks are easier to perform since they require less resources from the attacker. Typical examples of protocols that allow amplification are NTP, IGMP and also newer ones like DNSSEC.

Network Time Protocol [7] (NTP) is a protocol widely used to synchronize computer clocks in the Internet: NTP amplification is based on the existence of a specific command "monlist" that answers a UDP message with the list of the last 600 hosts that connected to the queried server. By spoofing the IP address and sending repeated monlist requests, the victim is flooded with UDP packets. IGMP DoS is based on a bad-configured network: usually IGMP messages are limited to the local network; however some routers reply to the "AskNeighbors2" query from arbitrary Internet hosts with a unicast "Neigbors2" response. The normal and legitimate use of these queries is to obtain a list of the local neighbours inside a network: a unicast "AskNeighbors2" query has to be sent to a router, that should answer with a unicast "Neigbors2" response. However, with a moderate botnet of 2000 bots, it is possible to flood a 10Gbps link, with an amplification of almost 20 times [8], by exploiting the existence of this bad-configured routers, used to amplify the attack. For DNSSEC amplification, the starting point is the same as for IGMP: a misconfiguration. Recursive DNS servers should accept DNS queries only from authorized clients, but sometimes this does not happen. Through IP spoofing, the victim can be flooded with UDP packets. The implementation of DNSSEC, a security extension of DNS, increased the amplification rate from 4.5 times to 45 times the size of the request[5].
Another possible method is called Smurfing: in this case it is exploited the "ICMP Echo Request" and a bad-configured network gateway. Sending such request to the broadcast address of this network and spoofing the source IP as the victim IP address, the victim will be flooded by the "ICMP Echo Reply" messages. A correctly configured gateway would simply not answer to ICMP requests sent to its broadcast address.

Furthermore, there are DoS attacks that do not involve layer 4 protocols, but use layer 7 protocols to damage the victim [9]. In this case HTTP requests with GET, POST or HEAD methods are used. The underlying concept is the same for all three methods: an incomplete request is sent to the server, sending subsequent headers at regular interval to keep the connection alive. The full header is never sent and the attack keeps going until the server resources are depleted.

Interestingly, a 2017 study shows that more than 30.000 DoS attacks per day are performed [10]. Moreover, TCP is the preferred protocol, accounting for around 80% of the total direct attacks, with UDP and ICMP following at 16% and 4.5%. Instead, reflection and amplification attacks are mostly performed with NTP, with 40% of the total, followed by DNS and CharGen at 26% and 22% respectively. Finally, an other interesting statistic about DoS attacks is that in TCP floods the preferred target ports are HTTP and HTTPS ones, with 50% and 20% of the total, while for UDP floods the victim ports are mostly associated with various on-line multiplayer games.

[5] https://blog.nexusguard.com/dnssec-fuels-new-wave-of-dns-amplification

3.2.2 Attack tools

HTTP Unbearable Load King (HULK[6]) is a popular DoS tool. It is designed to generate volumes of unique and obfuscated traffic, bypassing caching mechanism. The principle behind HULK is that a unique pattern is generated at each new request, thus evading intrusion detection and prevention systems. For instance, HULK will rotate both User-Agent and Referer fields of the requests.
Originally it was written in Python, but has been recently ported to Go. The main difference from the Python version lays in the concurrency architecture of Go language. The "goroutines" allow for a better usage of resources and consequently a much higher connection pool on the same hardware: a goroutine can be described as a lightweight thread, handled by the Go runtime and its scheduler; moreover, a goroutine is created with a very small amount of memory, that is expanded if needed. Once installed, the usage of HULK is quite simple:
`$ hulk -site http://example.com/test/ 2>/dev/null` is enough to start a DoS attack.

Goldeneye[7] is an HTTP DoS test tool, written in Python and created to test if a site is susceptible to DoS attacks. It exploits the "HTTP Keep Alive + NoCache" attack vector. This type of attack vector . Similarly to HULK, it changes User-Agent and Referer fields. Its usage is still simple: `$ ./goldeneye.py <url> [OPTIONS]` is the command to launch it; possible options include the usage of a custom file of User-Agents or the possibility to choose between GET or POST requests. This tool has not been updated since 2018.

Slowloris[8] is another HTTP DoS tool, still written in Python. It differentiates itself from previous tools using a different exploit. Its mechanism is based on keeping the connections alive sending header periodically (every 15 seconds). If the server closes a connection, the tool opens a new one. Also this tool can be easily launched, once installed, with `$ slowloris example.com [OPTIONS]`. Some options include the destination port selection, the usage of HTTPS or the possibility to set a different sleep timer between each header sent. Moreover, it can be used through Slowhttptest[9], a configurable tool that implements the most common Application Layer DoS attacks. Some of the modes supported by this other tool are Slowloris, RUDY, Apache range attack and Slow Read attack. Slowhttptest is written in C++.

Low Orbit Ion Cannon[10] (LOIC) and High Orbit Ion Cannon (HOIC) are two DoS application. The oldest one, LOIC, can flood the target server with TCP, UDP or HTTP packets. In order to generate enough traffic to damage the target, thousands of user need to coordinate the attack. Low-scale attacks can be easily mitigated with well written firewall rules. Since it needs a high number of coordinated users to perform a successful attack, the tool includes a so-called "Hivemind"

[6] https://github.com/grafov/hulk

[7] https://github.com/jseidl/GoldenEye

[8] https://github.com/gkbrk/slowloris

[9] https://github.com/shekyan/slowhttptest

[10] https://github.com/NewEraCracker/LOIC

mode to allow the remote control of the client: a sort of voluntary botnet. It is written in C#, but a JavaScript and a web-based versions have been released too. Interestingly, LOIC has been used in some high profile DDoS attacks, like "Project Chanology"[11] in 2008, when the Anonymous group used it to protest against the actions of the Church of Scientology, in the United States.
Its successor, HOIC, operates slightly differently and has been developed by the Anonymous group. It floods the victim with HTTP POST or GET requests and can target as many as 256 sites simultaneously. In this case, fewer people are needed to perform a low-scale attack, as few as 50. It also contains some additional features with respect to LOIC, like some scripts that let the attackers hide their geolocation. A famous attack perpetrated by the same Anonymous group happened in 2012 with "Operation Megaupload", one of the largest DDoS attacks at that time. It was launched to protest against the shutting down of Megaupload[12], a filesharing website, by the United States Department of Justice. In this case the targeted websites belonged to the U.S. Department of Justice, the Recording Industry Association of America, the Motion Picture Association of America and Broadcast Music, Inc.

3.3 Web attack

This type of attack is quite common. It consists in a set of methods that exploit vulnerabilities in dynamic web sites. The two main typologies are Cross Site Scripting (XSS) and SQL injection. In both cases, the objective is to grab private information from the victim who is visiting the web site.

3.3.1 Attack description

XSS attacks consist, typically, in a JavaScript code injection that allows the execution of this code in the victim's web browser. This can happen in different ways, leading to the definition of three main groups of XSS attacks [11]: server side ones are called Persistent and Non-Persistent, while on the client side there are DOM based XSS attacks. Moreover, there is also another client side attack; it exploits plug-ins to inject the malicious code.

Persistent attacks are maybe the most dangerous, because any legitimate user that will visit the infected website will be potentially affected by the XSS attack. Instead, to carry out a Non-Persistent attack, each single victim must be tricked into performing specific actions, as described later. Once a vulnerable website is found, the steps of the attack are the following:

1. one of the web application forms is used to inject the script, that is stored into the website repository;

[11] https://en.wikipedia.org/wiki/Project_Chanology

[12] https://en.wikipedia.org/wiki/Megaupload

2. the victim browses the vulnerable website and requests a web page;
3. the HTTP response of the server now contains the requested HTML web page incorporated with the stored script;
4. the malicious code is executed on the victim's web browser, transferring, for example, private credentials to the attacker.

The injected script can contain many different commands. Listing 3.3 shows a very trivial example of a script that sends the cookie of the victim to the attacker's website. Normally, the code does not contain only the script: for example if the vulnerable form is in a comment section on a website, the script is injected with a benign text, but only the text is shown when the page is loaded, while the script is executed.

```
<script>
       window.location = "http://attacker/?cookie=" +
          document.cookie
</script>
```

Figure 3.3: A trivial example of a script injected into a form.

Non-Persistent attacks work in a similar way, but in this case the script is not saved into the server repository. An ad hoc URL is crafted by the attacker and it contains the malicious code inside. When the victim uses such URL, the script is executed, with the same results as before. A possible example of a malicious URL is shown in Listing 3.4.

```
http://vulnerable.com/search?keyword=<script>...</script>
```

Figure 3.4: An example of a script injected into an URL.

In this case, a big difference with respect to the previous attack is that the victim must be induced to click on such URL, because simply visiting the vulnerable web site as usual, will not trigger any script. A common way to trick people to use these URLs are emails that try to mimic a real website email, but contain malicious links.

The third and last type of XSS described is the DOM based attack. The first steps are exactly the same as in Non-Persistent attack, the difference is purely about the type of HTTP response of the vulnerable website. While in the case of Non-Persistent attack, the server incorporates the malicious script directly inside the response message, in DOM based attacks the response contains a benign script that, using the same example of Listing 3.4, prints the requested keyword on the webpage. The difference is subtle, but in this second case the malicious script is executed on the victim's client once a benign script makes it possible. Still using the same example, the benign script of the client expects to print a keyword on

the HTML page; instead, since it does not perform any validation mechanism, it executes an unexpected script on the client's browser.

However, XSS is not the only way to attack vulnerable web sites. SQL injection attacks often have the same aim, i.e. obtaining private credentials of the victim, but can perform more dangerous operation and instead of working through JavaScript injection, they use SQL. In this case too, there are different possibilities: Piggy Backed and Union queries are the most simple and popular, but more advanced ones do exist [12].

Piggy Backed injections consist in the execution of additional queries that were not intended by the developer of the website. The most common example is made with a vulnerable login form, where username and password must be inserted. Let's assume that the server executes the query shown in Listing 3.5, once credentials have been retrieved from the client. The parameters `usr` and `psw` are the retrieved credentials. A Piggy Backed attack consists in the addition of another query after this one.

```
SELECT * FROM Users WHERE username = usr AND password = psw;
```

Figure 3.5: An example of a query vulnerable to SQL injection.

For example, if the attacker inserts in the password field a malicious input, the query becomes as shown in Listing 3.6, causing the deletion of the Users table.

```
usr = "Paperino"
psw = "0000; DROP TABLE Users"

SELECT * FROM Users WHERE username = "Paperino" AND password =
   "0000"; DROP TABLE Users;
```

Figure 3.6: An example of a Piggy Backed SQL injection attack.

Instead, the Union query method exploits the `UNION` keyword in SQL. The way to perform this injection is through a vulnerable form, as usual, and the resulting query could be the one shown in Listing 3.7. In this case the `--` at the end of the malicious username starts a comment in the SQL query, so the password field is not considered during the execution. In a real case there would be some differences from this example, since the attacker needs to be sure that the number and type of columns in the results of both queries are compatible, in order to use the `UNION` keyword.

Finally, there are more advanced strategies that combine SQL injection with other attacks. For instance, SQL + DDoS attack is a possibility. Exploiting specific SQL commands that allow to encode and decode strings plus forms vulnerable to previous attacks like `UNION` queries, it is possible to create SQL queries that take a lot of time to be executed. Encoding and decoding many times a set of columns

```
usr = "Paperino" UNION SELECT * FROM Credit_card WHERE username
    = "admin"; --
psw = "0000"

SELECT * FROM Users WHERE username = "Paperino" UNION SELECT *
    FROM Credit_card WHERE username = "admin"; -- AND password =
    "0000";
```

Figure 3.7: An example of a UNION SQL injection attack.

or a whole table can drain all the server resources, causing it to crash or to not be able to answer to further requests.

However, there are some good practices that can strongly mitigate the effects of these types of attacks and combined with more sophisticated techniques can detect them. A general advice is to always add validation steps to all the input fields, removing or escaping unwanted characters that can be used to perform XSS or SQL attacks. Character escaping can be split into several different categories[13]: the main ones are output escaping, JavaScript and Event Handler escaping and various specific tags escaping.
Output escaping is about HTML text that is directly printed on the page; in this case the five XML significant characters that need to be escaped are `&`, `<`, `>`, `"`, `'` and they respectively becomes `&`, `<`, `>`, `&quot;`, `&#x27;`. Other non-alphanumerical characters with ASCII code lesser than 256 should be escaped with the format `&#xHH;`, where `HH` is the hexadecimal ASCII code. Also HTML attributes like `attr` are subject to these escaping rules. JavaScript and Event Handlers (e.g. `onmouseover`) sensible characters are `'`, `"`, `\`, `/`, `<`, `>` (and others) and are escaped following the format `\xHH`, where `HH` is the ASCII hexadecimal code. Specific tags like CSS `<style>` require the escaping format `\HH`, while others like URL `href` require `%HH` format. Fortunately, each language usually offers library functions to automatically perform this kind of transformations.
Moreover, specifically for SQL injection, it is possible to use, beside specific character escaping, prepared statements or stored procedures to automatically parse the query variables[14]. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application, while prepared statements are written directly inside the application. An example of a Java prepared statement is shown in Listing 3.8. All these techniques can counter a large amount of basic attacks, but are not enough for more advanced ones, so specific countermeasures have to be taken.

[13] https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[14] https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE
   user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Figure 3.8: An example of a Java prepared statement.

3.3.2 Attack tools

Heartleech[15] is a tool that looks for a specific vulnerability inside the system. The OpenSSL library introduced a bug called Heartbleed (hence the name of the tool) between 2012 and 2014, due to an erroneous validation of inputs. The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users. This bug affected Operating Systems, websites and common software. Some popular websites that were affected are Yahoo!, Stack Overflow, GitHub, Reddit and Wikipedia. Its official record number inside the Common Vulnerabilities and Exposure (CVE) list is CVE-2014-0160[16], while the name comes from the TLS/DTLS extension's implementation which contained the bug, named Heartbeat.

Among other functionalities, the Heartleech tool is capable of obtaining private keys directly from a vulnerable website. The way it works, on a high level, is to look through the Hearthbleed memory buffer, which is accessible due to the vulnerability, one byte at a time and it constructs a `BIGNUM` variable. Then it obtains the public key of the server and tries to divide it with the constructed variable, if it is a prime number. If the result has no remainder, the other prime number needed to build the private key has been found. Moreover, it has been specifically designed to avoid being detected by IDS like Suricata[17], since their signatures trigger when the TCP payload starts with a specific pattern similar to `18 03 02 00 03 01 40 00`.

3.4 Botnet

Botnets are not properly cyber-attacks, but can be defined as the tools used to perpetrate specific attacks. Specifically, botnets are distributed computing platforms

[15] https://github.com/robertdavidgraham/heartleech

[16] https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

[17] https://blog.inliniac.net/2014/04/08/detecting-openssl-heartbleed-with-suricata/

used for illegal activities like launching DDoS attacks or sending spam, trojan and phishing emails [13].

3.4.1 Attack description

The first thing needed by an attacker to build a botnet is a vulnerability to exploit in the target systems. Typically, a single vulnerability is not enough to compromise a system, so the attacker has to combine several vulnerabilities to gain control of a computer [14]. An example of this kind of vulnerability is the buffer overflow, an exploit aimed at accessing a portion of memory not allocated to the process. For example, an old vulnerability present in the RPC protocol for Microsoft[18] allowed remote attackers to execute arbitrary code via a malformed message.
Moreover, attackers typically select specific networks to use as botnets: for example, some "always-on" broadband connections can provide a high bandwidth capacity to the attacker, speeding up the process. An example of such networks is the educational address space (.edu), that is often poorly secured and offers large storage capacities and a fast network connection [14].

However, the peculiar characteristic of a botnet with respect to other types of tool is the presence of a Command and Control (CNC) channel that allows to update and direct the systems that compose the botnet. The CNC architecture can vary between botnets: some examples of such architectures are IRC-based, HTTP-based, DNS-based or P2P-based [15]. The difference between these architectures mostly lays in the way the "botnet master" communicates with the bots and which network protocols are used. The most prevalent type of botnets are the one based on the Internet Relay Chat [16] (IRC) protocol, which was originally designed for large social chat rooms to allow for several forms of communication among large number of hosts. Instead, Peer-to-Peer (P2P) based botnets are not centralised like the other types, leading to an harder detection or monitoring [13].

Furthermore, some specific steps can be identified in the life-cycle of a botnet. First of all, the attacker scans a network for known vulnerabilities and infects the victim machines through the related exploits. At this point, the infected hosts execute ad-hoc scripts that download and install the bot binary file, typically through FTP, HTTP or P2P protocols. The next step to actually become part of the botnet is the connection of the infected machines to the CNC server: in this way the bots are ready to receive commands from the "botnet master", to perform the various types of attacks towards other victims. The final steps that compose the life-cycle of a botnet are the maintenance and update of the bots: the bots can download more binary files in the infected machines to update themselves. The possible reasons for this update are to avoid the detection of monitoring systems or to add new functionalities [13]. Finally, the "botnet master" can decide to move its control server towards another location (e.g. to avoid detection) and communicate to the bots the new location of the CNC server.

[18] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0352

3.4.2 Attack tools

Ares[19] is a remote access tool written in Python, composed by two main parts: a Command and Control (CNC) server to administrate agents, and the agent program, which runs on infected hosts and communicates with the CNC. Common usages of this tool are keylogging and file download, but its full capabilities include the execution of any shell command, the upload of a file to the server and the ability to take screenshots. A quite recent case involving this tool is from 2019, when it has been discovered that many Android set-top boxes and some smart TVs were vulnerable, due to a configuration service that has been left enabled on these devices. These IoT botnets have been used to trigger crypto-mining and dictionary attacks.

Nmap[20] is an open source software for network discovery. It is not a proper attack tool, but it is able to determine what hosts are available on the network, what services they are offering, the Operating Systems they are using and also what type of filters/firewalls are active on the network. These functionalities are useful to prepare actual attacks.

[19] https://github.com/sweetsoftware/Ares

[20] https://nmap.org/

Chapter 4

Intrusion Detection Systems

Intrusion detection is the process of monitoring the events occurring in a computer system or network, and analysing them for signs of intrusions [17]. With this definition, an Intrusion Detection System (IDS) is simply the tool, software or hardware, that allows the automated detection of intrusions. Moreover, it is also possible to automate countermeasures against these intrusions: in this case, the tool is called Intrusion Prevention System (IPS), or Intrusion Detection and Prevention System (IDPS), since it performs both functionalities.
This chapter describes the different typologies of IDS, that are classified based on the method, the approach and the technology they use. There are many other fine-grained differences between various IDSes, but the ones listed in the following sections are the main ones. Then, an overview of the main IDS solutions is presented, both commercial and open-source. At last, the Suricata IDS functionalities are described, since it is the IDS chosen for the integration with the Machine Learning classifiers, as better detailed in Section 6.2.

4.1 Detection methodologies

The two major categories of methodologies are Signature-based Detection (SD) and Anomaly-based Detection (AD) [17]. Each one of them has some advantages and disadvantages, as explained in the following sections. A hybrid approach can be taken, typically leading to an overall better result.

4.1.1 Signature-based Detection

A signature is a pattern that corresponds to a specific intrusion or threat. A database of signatures is kept and captured events are compared with them to find a match. Since previous knowledge is required in order to create such database, this method is also called knowledge-based.
The main advantage of this method is its simplicity; moreover, it is usually effective against known attacks. However, it has some major drawbacks: it is ineffective in detecting unknown attacks or even variants of known attacks. Moreover, it is quite difficult and time-consuming to keep the database up to date with new threats.

Finally, it has no understandings of network state or protocols: for a signature, there is no difference between a specific protocol or another, it simply looks for a match with the packet it is examining.

For example, a signature may try to match against Teardrop attacks, a type of DoS attack that involves sending fragmented packets to the victim and exploits an old bug present in the TCP fragmentation reassembly[1]. In this case, the signature has the objective of matching all the packets with an overlapping fragment offset. Another possible DoS signature is the one against the "ping of death"[2] DoS attack, presented in Section 3.2. In this case, the aim of the signature is to match all the ICMP packets with header and fragment offset that results in greater than 65535 bytes [18, chapter 10]. Instead, a signature that tries to match a possible TCP exploit is the one against the TCP NULL scan: this is a particular port scan technique that exploits the fact that a host that receives a TCP packet with all the flags set to 0 has to answer with a packet with the RST flag enabled in case of closed port and has to ignore the packet in case of active port, as defined in RFC-793 [19]. In this case, the signature would try to match all the packets without any flag set. Finally, another example of a signature against a TCP attack is the one for the Local Area Network Denial (LAND) attack[3], a type of DoS that exploits IP spoofing, as explained in Section 3.2, and sends a TCP packet with source and destination addresses and ports with the same value, causing the victim to continuously try to reply to itself. The signature simply looks for a match for this type of port/address pair.
Moreover, the signatures used in a signature-based IDS are typically obtained from publicly available sources[4], but the more advanced sets of pre-made rules must be bought.

4.1.2 Anomaly-based Detection

Instead, AD methods leverage the concept of anomaly, i.e. a deviation from normal behaviour. The core of this type of systems consists of two modules: the first one is in charge of defining a model of the normal behaviour, and can be also called Training module; the second one uses the model to perform the actual detection, measuring the deviation of an event in relation to the model [20].
The creation of the model can happen in different ways, but the training data used to define such model must be the most complete possible. Moreover, this model has to evolve with time, as the system behaviour evolves. Figure 4.1 shows a generic architecture of an anomaly-based IDS: the "sensor subsystem" has the role of monitoring the network activity and capture traffic. The "activity monitors" capture the traffic from the network in different ways, as explained in Section 4.3. Then, the pre-processing sub-module has the objective of converting the captured traffic into specific objects (e.g. C struct or Java object) using internal representations.

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0015

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0128

[3] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0016

[4] https://rules.emergingthreats.net/

Depending on the network layers and protocols monitored by the IDS, there could be an internal IP packet representation, a generic flow representation or others. These internal objects are then used by the detection subsystem to find anomalies and report them or take the chosen countermeasures.

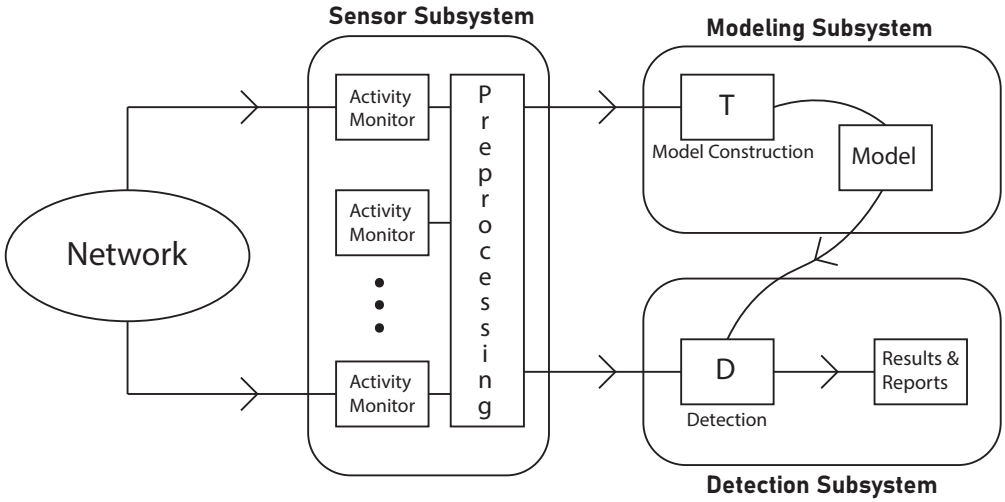

Figure 4.1: The generic architecture of an AD IDS.

It is possible to classify AD methods following three aspects of the workflow: the network features analysed, the type of behavioural model or the scale of the analysis [21]. Considering network features, it is possible to analyse different types of network traffic; for example, the detection could be focused on flow analysis (e.g. TCP flows) or involve a broader set of protocols, like FTP, TLS or ICMP. Typical examples of measures in a TCP/IP environment are:

- the number of bytes sent/received during a fixed time interval by a given final system;
- the number of IP/TCP/UDP/ICMP packets sent/received by a given final system during a fixed time interval;
- the number of TCP/UDP connections initiated during a fixed time interval;

On the other hand, also a distinction about the considered protocols can be done, classifying AD methods according to the network layers modelled (i.e. Data Link, Network, Transport or Application).
However, the core choice concerns the creation of the model, which is performed by the "modelling subsystem". In this case, there are two possible approaches: learnt models and specification-based models. Learnt models are based on various learning techniques, such as rule-based systems, statistical algorithms or various machine learning algorithms. The common aspect of all these approaches is the fact that the model is not manually specified by a human, but each learning technique uses some algorithm to extract knowledge from the data provided: more details on this type of approach is given in Section 4.2. On the other hand, in specification-based detectors, the model is constructed by an expert human. Such model is

composed of a set of specifications that capture legitimate system behaviour. These specifications can be provided by using any kind of formal tool, such as finite state machines. Also these approaches are further described in Section 4.2.

Finally, the scale of the analysis is often implicit in every detection method. A typical example of a scale is the time dimension: it is well known that traffic related measures exhibit, under normal conditions, periodic patterns that can be easily visualized [21]. These patterns can be identified over hours, days or even weeks of analysis. For example, counting the number of packets received by a service during intervals of one hour and plotting the results, it is possible to identify diurnal or nocturnal regimes or even weekly or seasonal characteristics. It is important to consider different scales because certain anomalies are only observable at specific scales: for example DDoS attacks usually need some sort of correlation among connections, since the inspection of individual packets may not reveal any sign of anomaly.

The principal advantage with respect to SD is the ability to detect new and unknown vulnerabilities. On the other hand, their response time to trigger alerts is generally slower and the models can be not very accurate due to the constant mutation of observed behaviours and events [20].

4.2 Detection approaches

Regarding the detection approaches, IDSes can be grouped into five subfamilies, each one containing algorithms and implementations with some common aspects. These groups are statistic-based, pattern-based, rule-based, state-based and heuristic-based.

Statistics-based IDSes mostly belong to AD methods and use predefined thresholds, statistical measures (e.g. mean, standard deviation) and probabilities to identify possible intrusions. A typical threshold approach consists of the count of events that occur over a period of time, raising the alarm if this count is lesser or greater than the specified threshold. The difficulty of this approach is determining the correct thresholds. The usage of statistical measures instead of fixed thresholds improves the effectiveness of these methods. Measures like mean or standard deviation can be used to determine if an event is anomalous or not, with a certain confidence. Moreover, this kind of approach allows for the usage of multivariate models, that consider the correlation between two or more metrics. Finally, a probability model consists of keeping track of the state of the system at fixed intervals, having a probability for each state at a given time interval. Once the state changes, if the probability of occurrence of that state is low, the alarm is triggered.
Pattern-based techniques all come from the signature-based group. These methods are simple and do not have great flexibility: they compare strings of bytes with a database based on known attacks in order to find intrusions.
Rule-based approaches belong mainly to the knowledge-based family of IDSes, but also some AD methods are possible. This type of detection uses rule-based languages to model the knowledge that experts have collected about attacks. This approach allows a systematic browsing of the network activity in search of evidence

of attempts to exploit known vulnerabilities. Sometimes they are also used to verify the proper application of the security policy of an organization [22].
State-based methodologies adopt the finite state machine concept: the model behaviour is captured in states, transitions and actions. Like other AD methods, a deviation from the expected behaviour, i.e. a transition to an unexpected state, will trigger the alarm.
Finally, heuristic-based IDS mostly belong to the AD family and include various techniques: some machine learning algorithms like Neural Networks or techniques like Genetic algorithms [23]. Genetic algorithms are a search technique used to find approximate solutions to optimization problems. The major advantage of genetic algorithms is their flexibility and robustness and are based on probabilistic rules instead of deterministic ones. They can be used to directly derive classification rules or as a feature selection tool, followed by other data mining algorithms to acquire the rules.

4.3 Technology types

The technology classification is based on the position inside the network where the IDS is deployed. It can be Host-based (HIDS), Network-based (NIDS) or Wireless-based (WIDS).

4.3.1 Host-based IDS

Host-based IDSes monitor the events occurring within a single host for suspicious activity. Examples of monitored activities are wired and wireless network traffic, system logs, running processes, file access and modification and system configuration changes. The detection software running on a host is also known as agent and it may also perform prevention actions. HIDS are typically deployed to critical hosts, such as servers containing sensitive information, although they can be installed also on other hosts. They are used primarily to analyse activity that can not be monitored in other ways. For example, since they are installed on end-points, they can analyse data that is normally encrypted for network-based IDS, but it is decrypted once it reaches the end-point.
Moreover, HIDS offer several specific techniques to monitor the host activity: code analysis, network traffic analysis and filtering, filesystem monitoring and log analysis [24]. Code analysis consists in monitoring attempts to execute malicious code on the host: one technique is to execute code in a virtual environment to compare its behaviour with known good or bad behaviour; another technique consists in checking signs of typical stack and heap overflow exploits aimed at accessing a portion of memory not allocated to the process [25].
Agents also perform network traffic analysis and sometimes can also monitor wireless communications. An agent can extract files sent by an application (e.g. email o peer-to-peer file sharing) to check for the presence of malware. The filtering action can restrict incoming and outgoing traffic for each application.
Instead, filesystem monitoring can be performed using different techniques, like file integrity checking or file attribute checking. File integrity checking consists of the

periodical generation of checksums (e.g. with MD5) of critical files, then comparing them to reference values. File attribute checking is used for less vital files and monitors files attributes like dimension, last modified date and write permissions. Both techniques are reactive, i.e. they detect attacks only after they happened, but some agents have more proactive abilities, such as monitoring file access attempts and preventing malicious ones. Finally, some agents can monitor system and application logs to identify malicious activity.

Moreover, HIDS usually require considerable customization: for example, they may rely on observing host activity to develop profiles of expected behaviour, but some need to be configured with detailed policies that define how each application should behave. Without this configuration, an agent could not distinguish, for example, if the installation of a new application is due to malicious activity or it is done as part of normal host operations. A big limitation of host-based IDSes is the considerable resource consumption on the host machine. For this reason, most of the techniques described above can be applied only periodically, such as every hour, leading to delays in detecting intrusions and consequent actions.

4.3.2 Network-based IDS

Differently from HIDS, NIDS are positioned over the network, with one or more sensors, and monitor and analyse network traffic for particular segments or devices, to identify suspicious activities. The NIDS network interface card is placed in promiscuous mode[5], to analyse all the packets they see, regardless of their destination [24]. Typically, they perform the analysis at the application layer, for example with HTTP, SMTP or DNS protocols. However, they also monitor the activity at the transport layer (e.g. TCP/UDP) and network layer (e.g. IP). Sometimes, they also perform limited analysis of the hardware layer protocols, like ARP.

Network sensors can be deployed in two ways: in-line or passive. In-line sensors are placed so that the traffic it has to monitor passes through it. For this reason, it may happen that in-line sensors are used as hybrid firewall/IDPS devices. The primary reason to deploy an in-line sensor is to stop attacks by blocking traffic.
On the other hand, passive sensors monitor a copy of the actual traffic: no traffic passes through it. Passive sensors can analyse traffic with various methods: with port mirroring, to send a copy of the traffic passing through one or more switch ports towards another port monitored by the sensor; with a network tap, i.e. a direct connection between the sensor and the physical network medium itself; or with an IDS load balancer[6], which is a device that aggregates and directs mirrored traffic towards the correct IDS, in case there are multiple IDS devices deployed in the network since all the packets of a flow must go to the same IDS to allow a correct analysis. Typically, passive sensors do not provide a reliable way to block the traffic, differently from in-line ones, but in some cases can try to do so: for example, by sending specific packets to interrupt a TCP connection towards both

[5]With promiscuous mode, the network interface controller accepts all the passing traffic, rather than accepting only the traffic that is destined to it.

[6]https://lwn.net/Articles/145406/

end-points.
Furthermore, IP addresses are normally not assigned to the network interface used to monitor traffic, except for interfaces used to manage the IDS. A sensor with no IP address on its monitoring interfaces is known as "stealth mode" and it provides some security benefits such as concealing it and preventing other hosts to initialise a connection with it. However, an attacker could be able to determine the existence of a sensor and its characteristics by analysing its prevention actions triggered in response to specific attack patterns.

Finally, the main limitation of NIDSes is the inability to analyse encrypted traffic (e.g. HTTPS, SSH or VPN connections). Moreover, NIDSes may be unable to perform full analysis under high loads: the delays in processing packets could cause unacceptable latency. To avoid this, some IDS sensors can recognize high load conditions and skip certain types of traffic without performing a full analysis; in other cases, it may be necessary to perform manual adjustments to the IDS configurations. For example, Suricata, a popular open source IDS, automatically skips some packets if it can not handle all the traffic, but it also provides some guidelines to solve this issue, if possible[7]. Furthermore, the presence of a firewall must also be considered to decide the best layout, since it can change the characteristics of the traffic. NIDS sensors are susceptible to some network attacks: the most popular are DDoS attacks, that can exhaust a sensor's resources, as described in Section 3.2. The most popular open source NIDS available are: Snort[8], Suricata[9] and Zeek[10], which are described in Section 4.4.

4.3.3 Wireless-based IDS

WIDSes are very similar to NIDSes, but their focus is wireless protocols and activities: they are most often used for monitoring wireless local area networks (WLANs). The typical components of a WIDS are the same as a NIDS, but their sensors are very different. Unlike a network-based IDS, which can see all the packets on the network it monitors, a wireless-based IDS works by sampling traffic from the two frequencies bands used by the IEEE 802.11 family of protocols [26], 2.4GHz and 5GHz; moreover, each band is split into several channels. Usually, a WIDS sensor is passive, i.e. it does not pass the traffic from source to destination, and its deployment can be in a fixed position or mobile [24].
Moreover, wireless IDS typically do not examine communications at higher network layers, like IP addresses or application payloads. Besides being able to detect DoS attacks, misconfigurations and policy violations, WIDSes can detect physical attacks, like the emission of electromagnetic energy on the WLAN frequency to make it unusable. Also, most wireless sensors can identify the position of a wireless device by using triangulation, if multiple sensors are available. Similarly to NIDSes passive sensors, the only countermeasure to an ongoing attack is the termination of

[7] https://suricata.readthedocs.io/en/latest/performance/analysis.html

[8] https://www.snort.org/

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

[10] https://zeek.org/

connections, in this case through the air: this happens by sending messages to the interested end-points, telling them to de-associate the current session and refusing new connections to be established.
However, also WIDSes have some limitations: for example, they are subject to some evasion techniques: an attacker could launch the attack on more channels at the same time since a sensor can not monitor them all at once [24]. Moreover, they are also subject to some attacks like DoS or specific WLAN attacks like jamming. The only way for a sensor to avoid the disruption of its radio transmissions by jamming is to establish a physical perimeter so that attackers can not get close enough to perform it. Finally, some types of attack can not be detected at all by WIDSes: for example, an attacker can passively monitor wireless traffic without being detected, to process it off-line at a later time.

4.4 Popular IDSes

On the market, there are various IDSes available, both free or paid. Some of them are only available for specific operating systems, while others are compatible with all the main ones. Moreover, a great part of them is specialized in either HIDS or NIDS functionalities, while some can perform both activities. It is common that IDS products are integrated with firewall functionalities. Starting from paid ones, the most popular are described. Unfortunately, paid IDSes vendors typically do not publish too many technical details about their product.

SolarWinds Security Event Manager[11] is a log manager that provides various features. For this reason, it can be classified as a host-based IDS. It provides a centralized log collection and normalization and an automated threat detection and response. It monitors user activity and file integrity, but it can also detect DDoS attacks. Finally, it can analyse and manage firewall logs. Some of the active responses it can trigger include: SNMP alerts, screen messages and emails; USB device isolation; account suspension or user expulsion; IP address blocking; process killing and system shutdown or restart. An annual subscription for 30 nodes costs 2000$.
CrowdStrike Falcon[12] is a host-based IDS that offers end-point protection. Also in this case there are no technical details available, but it offers proactive threat hunting with continuous raw events capture. It also offers typical antivirus capabilities and firewall management. The price starts from 16$/month per end-point, for a minimum of five end-points and a maximum of 250.
ManageEngine EventLog Analyzer[13] is a HIDS focused on analysing log files. The website offers a detailed list of features logged[14], including event, system, server and application logging. Moreover, it can analyse logs of Microsoft SQL Server and Oracle. Apart from the paid plans, it offers a free tier that includes monitoring of

[11] https://www.solarwinds.com/security-event-manager

[12] https://www.crowdstrike.com/

[13] https://www.manageengine.com/products/eventlog/

[14] https://www.manageengine.com/products/eventlog/features.html

up to five sources.
TrendMicro Tipping Point[15] is a stand-alone IDPS, designed to identify and block malicious traffic. It can be deployed into the network with no IP address or MAC address to immediately filter out malicious and unwanted traffic. Its filters leverage machine learning and statistical data modelling. TippingPoint can also detect domain name system (DNS) requests from malware-infected hosts attempting to contact their command and control (CnC) hosts. Moreover, it supports IPv4 and IPv6 payload inspection, also with VPN or MPLS tags and GRE traffic [27]. It is suited for large and very large enterprises, including banking, telecom, healthcare and transportation. The base model price starts at 6000$.
Hillstone IDPS[16] is a network-based IDPS that offers intrusion prevention, antivirus, application control, advanced threat detection, abnormal behaviour detection, a cloud sandbox and a cloud-based security management and analytics platform. It operates in-line, performing deep packet inspection, and assembling inspection of all network traffic. It also applies protocol anomaly analysis and signature analysis to block threats. It is suited for enterprise customers who need a standalone IDPS solution. The price starts from 18000$.

4.4.1 Suricata

Developed in C language by the Open Information Security Foundation[17] (OISF), Suricata is a multi-platform network threat detection engine. It is capable of both detection and prevention functionalities, so it can be classified as an IDPS. It is one of the most widely used open source IDSes and it supports a powerful rule and signature language, plus the ability to use Lua[18] scripting to detect complex threats. Some examples of other features are:

- offline analysis of PCAP files;
- traffic recording using PCAP logger (i.e. the ability to export the captured traffic into PCAP files);
- YAML[19] configuration file, which is easier to read for humans with respect to other formats, like XML;
- full IPv6 support;
- TCP stream engine, capable of tracking and reassembling sessions;
- support for decoding of IPv4, IPv6, TCP, UDP, SCTP, ICMPv4, ICMPv6, GRE, Ethernet, PPP and VLAN packets;

[15] https://www.trendmicro.com/it_it/business/products/network/intrusion-prevention/tipping-point-threat-protection-system.html

[16] https://www.hillstonenet.com/products/network-intrusion-prevention-system-s-series/

[17] https://oisf.net/

[18] https://suricata.readthedocs.io/en/latest/lua/index.html

[19] https://suricata.readthedocs.io/en/latest/configuration/suricata-yaml.html

- support for decoding of HTTP, SSL, TLS, SMTP, FTP, SSH, DNS and DHCP app layer protocols;
- FTP, HTTP and SMTP engine[20] capable of transaction logging and file identification, extraction and logging; the file extraction works on top of other protocol parsers and it is capable of storing on disk the re-assembled file extracted from the request/response data;
- live rule reloading, without restarting Suricata;
- EVE and JSON logging, plus the ability to generate personal output formats with Lua scripts;
- event filtering, based on rules or thresholds;
- multi threading support;
- IP reputation[21] mechanism, integrated into the rules language.

The purpose of the IP reputation component is the ranking of IP Addresses within the Suricata Engine. This separate module most often runs on a central database that all sensors of the IDS already have communication with. This module is able to subscribe to one or more external feeds: each one regarding a vast number of IP addresses and containing positive or negative intelligence classified into a number of categories.

Listing 4.2 presents an example of a packet signature, with the parts that compose it in different colours. The signature syntax used for the example belongs to Suricata[22], but it is almost fully compatible with other popular IDSes, like Snort.

```
drop tcp $HOME_NET any -> $EXTERNAL_NET any
(msg:"ET TROJAN Likely Bot Nick in IRC (USA +..)";
flow:established,to_server;
   flowbits:isset,is_proto_irc;
content:"NICK "; pcre:"/NICK .*USA.*[0-9]{3,}/i";
reference:url,doc.emergingthreats.net/2008124;
classtype:trojan-activity; sid:2008124; rev:2;)
```

Figure 4.2: An example of a Suricata signature

The keywords `tcp`, `$HOME_NET`, `$EXTERNAL_NET`, `any` and `->` are part of the header of the signature, while the remaining part consists of options. Instead, the `drop` keyword is the action: it determines what happens when the signature matches. Possible actions are:

[20] https://suricata.readthedocs.io/en/latest/file-extraction/file-extraction.html

[21] https://suricata.readthedocs.io/en/latest/reputation/ipreputation/ip-reputation.html

[22] https://suricata.readthedocs.io/en/latest/rules/intro.html

- alert - generate an alert;
- pass - stop further inspection of the packet;
- drop - drop packet and generate alert;
- reject - send RST/ICMP unreach error to the sender of the matching packet;
- rejectsrc - same as just reject;
- rejectdst - send RST/ICMP error packet to receiver of the matching packet;
- rejectboth - send RST/ICMP error packets to both sides of the conversation.

`tcp` is the protocol. In this case, the list of supported protocols is much longer, but a few examples are TCP, UDP, ICMP, HTTP, FTP, TLS, DNS, SSH, SMTP, DHCP.
`$HOME_NET, $EXTERNAL_NET` represent source and destination of the traffic. Both IPv4 and IPv6 addresses are supported and can be combined with some operators. A couple of examples could be the ones presented in Table 4.1.

Example	**Meaning**
!1.1.1.1	Every IP address but 1.1.1.1
![1.1.1.1, 1.1.1.2]	Every IP address but 1.1.1.1 and 1.1.1.2
$HOME_NET	HOME_NET setting in configuration file
[$EXTERNAL_NET, !$HOME_NET]	EXTERNAL_NET and not HOME_NET
[10.0.0.0/24, !10.0.0.5]	10.0.0.0/24 except for 10.0.0.5

Table 4.1: Examples of syntax of possible IP addresses.

The two `any` keywords indicate the source and destination ports. Port numbers can be combined in a way similar to addresses.
Finally, the `->` symbol tells in which way the signature has to match. Nearly every signature has an arrow to the right. This means that only packets with the same direction can match. However, it is also possible to have a rule match both ways, with the `<>` symbol. The `<-` direction does not exist.

The remaining part of the signature consists of options. There are lots of possible options for each signature: these are enclosed by parenthesis and separated by semicolons. Some options have settings (such as `msg`), which are specified by the keyword of the option, followed by a colon, followed by the settings (e.g. `flow:established,to_server;`). Others have no settings and are composed simply by the keyword. Rule options have a specific ordering and changing their order would change the meaning of the rule.

A high level overview of the Suricata IDS pipeline can be seen in Figure 4.3, which begins with the capture of packets from the network and ends with the

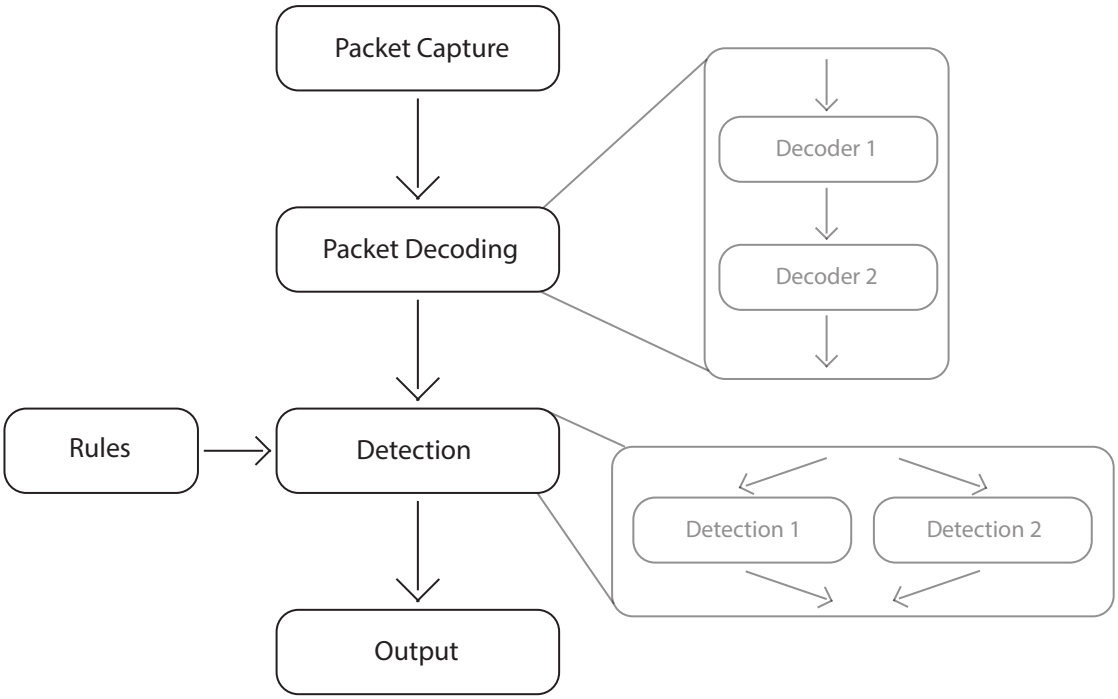

Figure 4.3: The high-level Suricata workflow.

creation of log outputs. The IPS workflow is slightly different since it also includes eventual countermeasures to take after a signature match.

Captured packets pass through the Decoding module first. The decoding pipeline starts from the lower layers of the stack and proceeds with the higher ones. It is also possible to create a custom decoding functionality and to place it into the pipeline. Decoded data is stored as an internal representation that is then used by the following modules.

The Detection module is the core one. Differently from the Decoding one, it operates in a parallel way. Each Detection function checks one rule against each packet, and in case a rule is triggered, the action specified in the rule itself is taken. Rules can be composed of one or more patterns or signatures, as explained in Section 4.1.1. Moreover, Suricata provides a pre-filter engine: since there are too many rules to be checked individually, it is possible to check only a specific pattern for each rule before checking the whole rule. The most common example is the Multi Pattern Matcher (MPM), where only the rules that have at least one match in the MPM stage are completely checked. This algorithm works in a simple way: of each signature, one pattern is used by the MPM. That way Suricata can exclude many signatures from being examined because a signature can only match when all its patterns match. Finally, it is also possible to add custom detection functions to the module or to use Lua scripts to achieve more complex tasks.

The output produced is one or more log files, where the content can be highly customized in order to show specific alerts or warnings in different formats. The most common way of logging in Suricata is through the Extensible Event Format,

nicknamed as EVE[23]. EVE is a specific configuration file format, used to select the alerts, anomalies, metadata and files info that will be stored in the actual log file. An example of an EVE configuration file is shown in Listing 4.4: in this specific example, the logging is enabled and will produce a file called "eve-ids.json". This file will contain details about the `alert` and the `drop` actions caused by signature matches.

```
outputs:
       - eve-log:
              enabled: yes
              type: file
              filename: eve-ids.json
              types:
                     - alert
                     - drop
```

Figure 4.4: An example of EVE configuration file.

However, the characteristics of the produced output can be deeply customised: for example, the possible types of `alert` can be specified. In this case, the user can choose to log only alerts produced by some specific options of the rule, instead of logging all the triggered alerts. Some examples of sub-sets of alerts are `http, tls, ssh, smtp, dnp3, flow, vars`; moreover a `dns` type can be used, with options for DNS queries and answers. Instead, the produced log file uses the JSON format: an example of output is shown in Listing 4.5 and contains information about a single alert. In this example, the source and destination IP addresses are shown (`"src_ip": "192.168.2.7"` and `"dest_ip": "x.x.250.50"`) and the specific signature that raised the alert is also listed, using a signature id among the other fields (`"signature_id" :2001999`), used to uniquely identify a signature inside the used database.

Finally, a more in-depth description of some of the Suricata modules is provided. Due to the complexity of the tool, only some of the logical blocks are shown in Figure 4.6, mostly the ones of interest for this work, as detailed in Section 6.2. Since Suricata is a multi-thread software that performs in parallel many different operations, the logical blocks have been divided across the different modules that have been already described in Figure 4.3.

The initialisation block is run at the beginning, as soon as Suricata is launched. The first major step is the reading of the "suricata.yaml" configuration file, to obtain details about the user choices (e.g. output folder, signatures to use, enable/disable specific modules). Then, the command line arguments are parsed to eventually override some configurations or add different ones. The next major logical step is the parsing of the signatures provided by the user. In fact, to manage Suricata rules it is possible to proceed in two ways: automatically download from

[23] https://suricata.readthedocs.io/en/latest/output/eve/index.html

```
{
        "timestamp": "2009-11-24T21:27:09.534255",
        "event_type": "alert",
        "src_ip": "192.168.2.7",
        "src_port": 1041,
        "dest_ip": "x.x.250.50",
        "dest_port": 80,
        "proto": "TCP",
        "alert": {
                "action": "allowed",
                "gid": 1,
                "signature_id" :2001999,
                "rev": 9,
                "signature": "ET MALWARE BTGrab.com Spyware
                   Downloading Ads",
                "category": "A Network Trojan was detected",
                "severity": 1
        }
}
```

Figure 4.5: An example of Suricata JSON output.

the web pre-made signatures for different types of attack or manually write and add signatures to Suricata. All the provided rules are parsed and a syntactic check is performed; then the rules are loaded into internal structures to be used from other modules. Once all the initialisation has been completed, the threads deputed to perform decoding or detection are created and started.

A single thread of the decoding module works in the following way: it extracts a packet from the packets pool, a sort of list structure where packets are inserted after being captured on the network interface. Then, each protocol that composes the packet is decoded, calling the appropriate function. Each protocol information is parsed and used to update internal structures and statistics. The inner protocol information is then passed to the next decoding function until all the supported parts of the packet have been analysed.

Finally, the detection module has the objective of finding matches between decoded packets and signatures. Each detection thread extracts a "packet" from an internal structure. As explained above, this is no more a network packet, but a C struct containing all the details about a specific packet, the related flow and eventual other higher layer protocols. At this point a first pre-filtering is performed: for example, using the MPM algorithm described above. Then, each remaining signature is checked individually. Furthermore, to reduce the time needed to check all the signatures, its header is checked first (i.e. the protocol, the source and destination IP addresses and ports and the direction of the packet). If the header part of a specific signature matches, then the remaining options are checked. If also all the rest of the signature matches, the related action is performed.

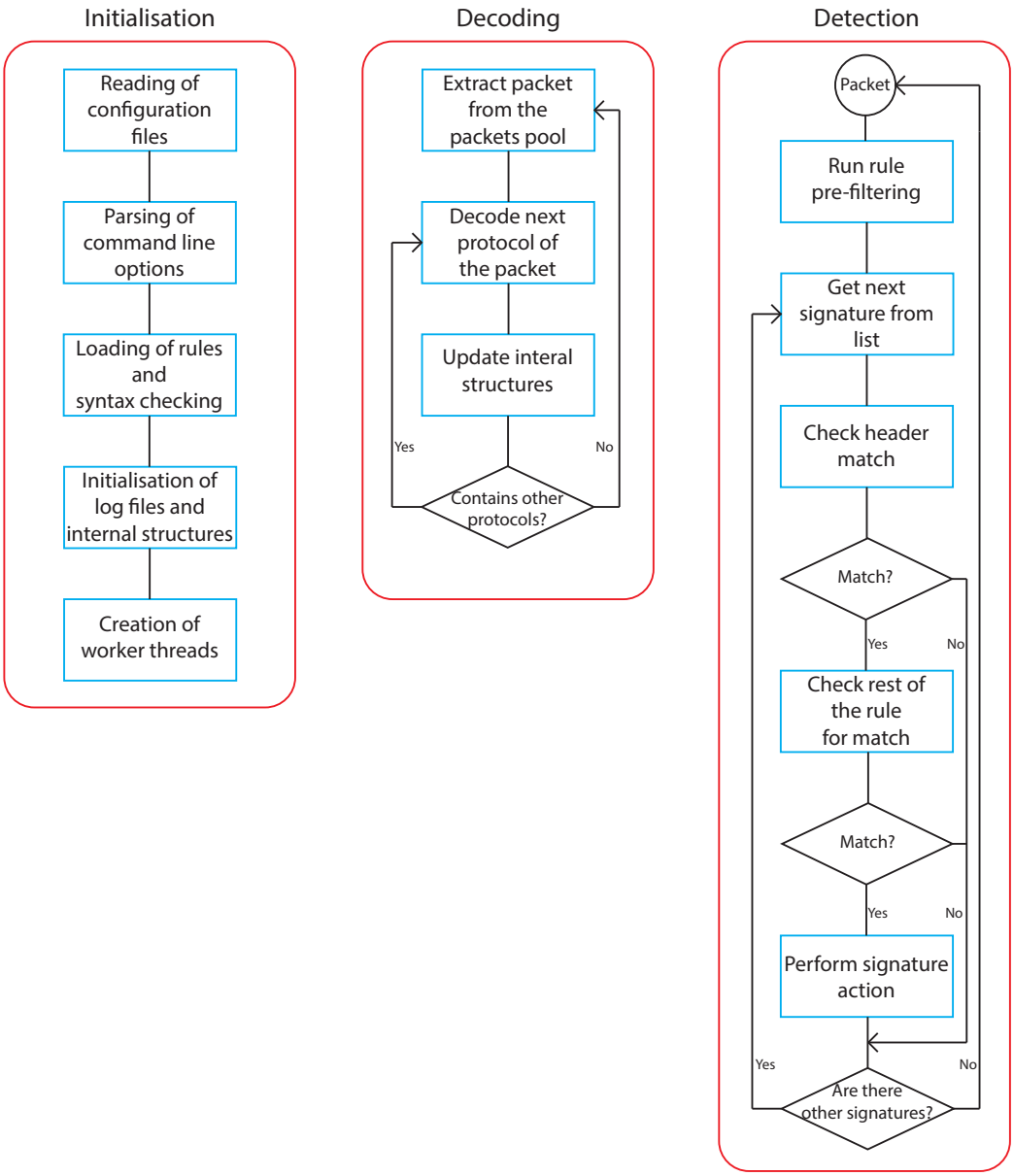

Figure 4.6: The logical blocks of a part of the Suricata workflow.

4.4.2 Snort

Snort[24] is an IDPS that performs packet inspection using pattern matching. This matching is implemented in the form of rules, which syntax is almost totally compatible with the one of Suricata rules, that has been described in Section 4.4.1. Some minor differences exists, but are mostly about advanced features introduced by the two IDSes to add more functionalities to each tool[25]. The Snort architecture contains some modules called pre-processors, that read the captured packets

[24] https://www.snort.org/

[25] https://suricata.readthedocs.io/en/latest/rules/differences-from-snort.html

before rule evaluation, serially in the order specified by Snort's configuration. Pre-processors allow the implementation of functionalities more complicated than pattern matching, such as anomaly detection. A few examples of pre-processors [28, chapter 6] included in Snort are the fragmentation, the stream and the reputation pre-processors.
The fragmentation pre-processor has the objective of managing the re-assembling of packets that it is analysing. This pre-processor is of fundamental importance because different hosts with different operating systems re-assemble packets in different orders. This means that if the Snort pre-processor would re-assemble fragmented packets in an order specific to a single OS, it may not detect exploits based on fragmentation for another OS. For this reason, the fragmentation pre-processor uses seven different policies to re-assemble packets and keeps track of a high number of fragments (8192 by default). The seven policies are based on the conventions used by most of the operating systems like BSD[26], Linux, Windows and Solaris[27].
The stream pre-processor is a target-based TCP reassembly module for Snort. It is capable of tracking sessions for both TCP and UDP. TCP protocol anomalies, such as data on SYN packets or data received outside the TCP window, are configured via the `detect_anomalies` option in the TCP configuration. Some of these anomalies are detected on a per-target basis. For example, a few operating systems allow data in TCP SYN packets, while others do not.
Instead, the reputation pre-processor provides basic IP blacklist/whitelist capabilities, to block/drop/pass traffic from IP addresses listed. In the past, the standard Snort rules were used to implement reputation-based IP blocking, but this pre-processor will address the performance issue and make the IP reputation management easier. This pre-processor runs before other pre-processors. The IP lists are loaded from external files and multiple blacklists or whitelists are supported. One of the biggest limitations of Snort is that it is a single-threaded application, even if it is possible to obtain multithreading by splitting the flow into multiple parts and letting a different instance of Snort analyse each part.

4.4.3 Zeek

Zeek[28] is primarily not rule-driven, differently from Snort, but it implements its own scripting environment, with its own programming language. It is an interpreted, typed language and what makes it interesting is the existence of domain-specific types. For example, the "addr" type holds an IP address. Two types of collections are present: sets and tables. The loops are available in the form of iteration through collections.
The Zeek scripting language[29] is event-driven: an example of an event is the extraction of a file from the network traffic. In this case, the Zeek scripting language

[26] https://www.freebsd.org/

[27] https://www.oracle.com/it/solaris/solaris11/

[28] https://zeek.org/

[29] https://docs.zeek.org/en/current/examples/scripting/

allows to create a function that handles this specific event to perform certain analysis. For example, Listing 4.7 shows an event handler that is triggered when the "Files framework" of Zeek, deputed to extracting and hashing files, extracts a file from a flow. The `file_hash` event allows scripts to access the information associated with a file: in this example the script checks that the hash function is SHA1 and then performs a lookup of the hashed file in a database containing hashed malware. In case of match, the "Notice framework" will create the corresponding log. The events handled by Zeek are contained in an "Event Queue" and are processed on a first-come-first-serve basis.

```
event file_hash(f: fa_file, kind: string, hash: string)
{
        if ( kind == "sha1" && f?$info && f$info?$mime_type &&
        match_file_types in f$info$mime_type )
        do_mhr_lookup(hash, Notice::create_file_info(f));
}
```

Figure 4.7: An example of Zeek event.

Differently from Snort, Zeek acts only as an IDS, rather than an IDPS: it monitors the traffic and produces log files with alerts, to be checked manually or through other systems. Some of the items monitored by Zeek are bidirectional flows, DNS queries and responses, HTTP requests, port scans, email headers from SMTP traffic, successful and unsuccessful SSH connections, SSL certificates and traffic tunnels. The pre-installed scripts usually expose an API that can be used by users to extend the default functionalities. Finally, the scripts are organised in modules, that can be broken in several files. A module can define types, variables, functions, and event handlers. These entities can be either local to the module or globally accessible from other modules. Zeek can be run both as a single-threaded application and as a multithreaded application.

Chapter 5

Machine Learning and classification techniques

This chapter contains an introduction to Machine Learning (ML) and a technical explanation of the classification techniques used in this work's experiments. The machine learning section presents the typical workflow of ML models, from the analysis of the dataset until the evaluation of the model, detailing some of the most common issues. Then, in the second section, algorithms of decision trees, random forests and support vector machines are described, followed by an overview of neural networks.

5.1 Machine Learning

Machine learning is a sub-category of Artificial Intelligence (AI), which is the study of "intelligent agents" or "rational agents", i.e. something that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. From the broad field of AI, machine learning can be defined as the sub-category focused on improving the agent's chance of reaching its goal through experience. More precisely:

Definition A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [29].

For example, a computer program that learns to play checkers might improve its performance as measured by its ability to win (P) at the class of tasks involving playing checkers games (T), through experience obtained by playing games against itself (E).

Moreover, machine learning techniques can be split into several groups, the main ones being "supervised" and "unsupervised" families. Supervised learning includes all methods that assume the presence of labelled data, i.e. usually a human decided value that indicates the expected output, the target. On the other hand, other approaches that do not require labelled data and do not have a target are

called unsupervised, e.g. clustering or dimensionality reduction algorithms.
Furthermore, supervised models can be further split into two groups: classification and regression. While regression models have as output a continuous value, classification ones have a discrete value (a class). Due to the nature of the data used in this work's experiments, as discussed in Section 7.1, only classification algorithms have been utilised. The description of the used algorithms can be found in Section 5.2.
Finally, the learning process of every ML model goes through some common steps, that include some best practices to avoid possible issues.

5.1.1 Dataset analysis and features selection

First of all, the source dataset must be analysed. The number of features and the nature of each of them has to be inspected, to eventually perform some modifications. It is common to use graphs of different types to better visualise the distribution of each class, for example, histograms, scatter diagrams or box-plots. A possible problem that may be detected visualising the dataset is to have a high unbalance between the classes: for example, in a binary classification problem, one of the two classes may have a number of samples that is 10000 times higher with respect to the other class. The next steps will have to act accordingly to this information, for example with a specific sampling technique or by completely ignoring that class.
Then, the first selection of features is performed. Sometimes the source dataset contains features that are not useful for that particular work, maybe because the data has been collected for other purposes, or because different experiments have to be performed with the same dataset. In this case, the useless features are simply discarded, as they would add noise to the model learning. Once these preliminary steps are performed, the preprocessing of the dataset can start.

5.1.2 Dataset preprocessing

Dataset preprocessing is a set of techniques that aim to prepare the dataset to the actual training. Most of the times the dataset is not ready "as-is", but must be modified in various ways. Some of the most common techniques involve the cleaning of the dataset, the transformation of certain features and the scaling of values.

Dataset cleaning techniques are needed when some samples have missing or invalid values or, for any reason, are duplicate inside the dataset. Usually, the algorithms do not tolerate missing values: in this case, the possible choices are the removal of the sample from the dataset or the insertion of an ad-hoc value in the missing spot. The removal of the defective sample is a straightforward way to avoid the problem, but may not be possible in all cases: for example if the dataset is really small, removing samples is not a good idea. In this case, there are methods to find a value to assign; regression algorithms can be used to predict the missing value, but a faster way is to replace it with the average of the existing values. Finally,

duplicate samples are usually removed from the dataset, since they do not carry any useful information.

Feature transformation techniques are used to change the type of specific features. For example, some algorithms like Neural Networks only accept numerical attributes, but sometimes datasets contain categorical values. In this case, it is necessary to transform the feature values domain with a process called "encoding". The opposite process, "decoding", must be performed at the end of the learning phase, to bring back the values to the original domain.
A specific encoding technique, called "one-hot encoding", consists in the creation of a column for each category of the original domain, as shown in Figure 5.1, assigning the value one when the category is the same as the original, zero otherwise.

Age	Salary	Gender
25	24000	F
49	32000	M
36	40000	F
41	35000	F
58	50000	M

Encoding →
← Decoding

Age	Salary	Male	Female
25	24000	0	1
49	32000	1	0
36	40000	0	1
41	35000	0	1
58	50000	1	0

Figure 5.1: An example of feature encoding.

Finally, scaling the values of features is fundamental, since some algorithms are highly influenced by the absolute value of features and by their variances, like Support Vector Machines. It may happen that a feature completely dominates over the other, making also difficult a visual comparison between distributions: for example, a feature may have very small values in the range 0-1, while another feature (e.g. a time measure or a distance) may have values in the range 0-100000. A common way to perform this scaling is known as "standardisation", its objective is to scale each feature so that its distribution has zero mean and unit variance. The formula to perform such scaling is:

$$x_{norm} = \frac{x - \mu}{\sigma}$$

where μ and σ are respectively the mean and the variance of a feature. Once these operations have been performed, the model to train must be selected.

5.1.3 Model selection

Choosing the right model for a specific problem can make the difference between good and bad results. There are lots of different ML algorithms and each one of them is based on different assumptions and can be used with more or less good results depending on the context. It must be kept in mind that, as stated by the popular no-free-lunch theorem [30, 31], none of these methods has proven to be successful to all type of problems, so the space of solutions has to be explored to find the most suitable one to the specific situation.

For example, one of the simplest classification algorithms is K-Nearest-Neighbours (K-NN): it is based on the concept that samples that belong to the same class are positioned one near the other inside the features space; to perform a classification of a point, the K nearest samples are considered and a majority voting assigns the class. This simple algorithm has some advantages like the absence of hypothesis about the data or the presence of a single parameter to optimise (K), but has the disadvantage of requiring a lot of time at inference phase and it needs to store all the training samples to work. Moreover, as all the algorithms that involve the computation of a distance, it is subject to the curse of dimensionality, which is explained slightly below. On the other hand, an algorithm like decision trees has different advantages, like the ability to work with categorical data and the easier interpretability of the results with respect to other models. However, as explained below, it is often subject to the phenomenon of "overfitting". For these reasons, if possible, different models are used to solve a problem and the results are then compared to pick the best one.

Curse of dimensionality is the name given to a phenomenon that happens when a dataset has too many features, with respect to the number of samples. As an example, imagine a fixed number of samples within a hyper-cube of increasing dimensions. For simplicity 1D, 2D and 3D examples are shown in Figure 5.2. As the number of dimensions increases, there are fewer samples for each region of the hyper-cube: the density decreases. For algorithms based on the computation of distances between samples to determine similarities and perform the classification, a low density of the samples leads to a situation where the points identified as most similar by the algorithm are not truly similar since they may be far from each other. As the number of dimensions increases, the number of samples needed to keep a high density increases exponentially, leading to a prohibitive number of samples: the only way to avoid the explosion of the number of needed samples to still have good performances is to limit the number of features.

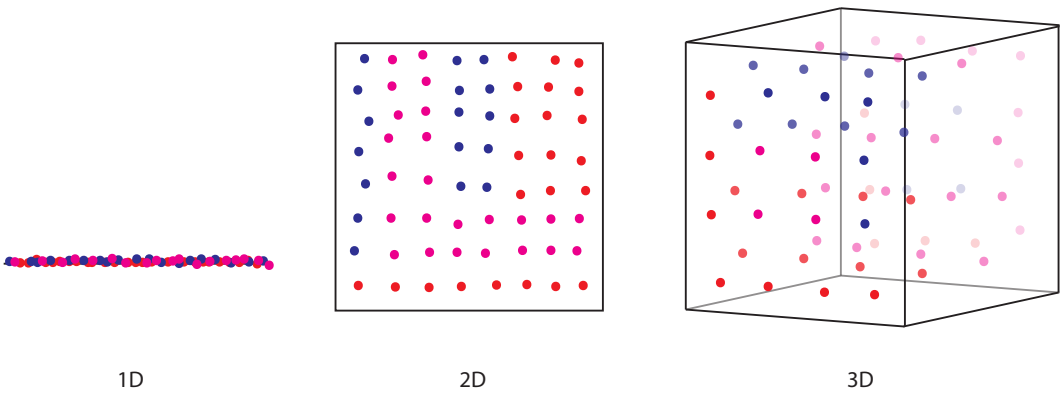

Figure 5.2: An example of samples distribution inside an increasing dimensional hyper-cube. 64 samples in each hyper-cube have decreasing density.

5.1.4 Training, validation and testing

The actual learning phase is typically split in training, validation and testing. These three phases consist in the usage of different sub-sets of the dataset to create a machine learning model that is able to predict the class (in case of classification algorithms) of a generic sample, being it new or previously seen by the model. The training phase is the core one inside this step: it contains the actual execution of the algorithm chosen to create the model. Instead, the validation phase has the objective to choose among different trained models the best one to use for the final test phase: in fact, multiple training phases are typically performed during a machine learning pipeline, leading to the creation of multiple models. This is caused by the fact that each ML algorithm can be executed using different configurations of the base model: a trivial example is the number of estimators used by a random forest algorithm (i.e. the number of trees inside the forest), as explained in Section 5.2.2. Since there is not a fixed number that works well for all the datasets, the training phase must be repeated with different combinations of these numbers, called "hyper-parameters". Finally, the test phase evaluates the performance of the best model chosen by the validation with a set of unseen samples, i.e. samples not used to train the model.

There are multiple reasons for the choice of splitting the creation of a model in three phases, the main one being the presence of "hyper-parameters" inside the model, that have to be tuned to obtain the best results. To take a shortcut, one may think to remove the validation set and perform this tuning evaluating the results of the different trainings on the test set: this is always a bad idea. In fact, the test set must be used only at the end of the process, to evaluate the goodness of the final model, once all the hyper-parameters have been fixed. Using the test set instead of the validation set would introduce a form of bias that typically leads to overfitting.

Overfitting, and its opposite underfitting, are two phenomena that involve the way a model adapts to the training data. Using a regression problem as an example, as shown in Figure 5.3, it is possible to imagine the underfitting phenomenon with a model that is unable to adapt to the training data, leading to low accuracy. This may be caused by a too simple model. On the other hand, overfitting is the opposite: the model is too much precise in fitting the training dataset, leaving no space for samples coming from a different distribution. Even if the training accuracy is quite high, overfitting usually leads to a lower accuracy at test time.

Hence, for these reasons, it is recommended to split the dataset into training, validation and test sets. However, this split can happen in many different ways, from more straightforward ones to more elaborated. Starting from the proportions of the split, there is not a standard percentage for each set. Typically, the training set is the bigger of the three, but the proportions may vary and can depend on the size of the dataset. For example, a smaller one may need a higher cut for the training, to have enough training samples, while a bigger dataset may allow for a bigger test set. Also, the validation set dimension can vary greatly, reaching as few as one element in case of a very small dataset, as explained later with the different validation techniques.
However, the main choices to split the dataset are the following: simple random

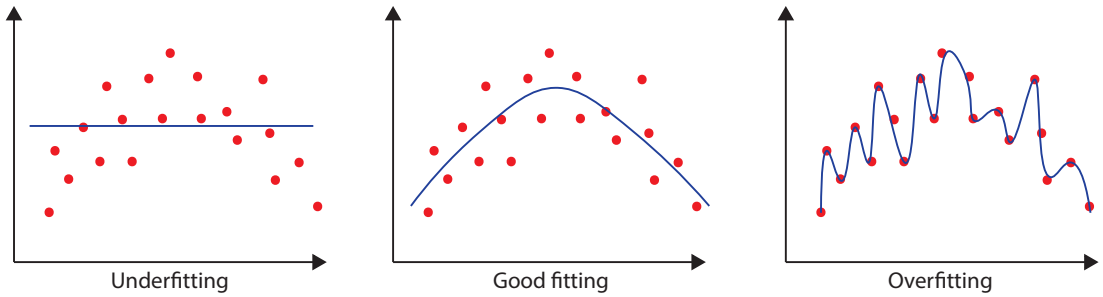

Figure 5.3: Examples of underfitting and overfitting in a regression problem.

sampling, stratified sampling and downsampling [32]. Simple random sampling does exactly what the name says: each sample is picked with the same probability, independently of its class. However, this may lead to situations where a class is not represented at all in the training set: for example, Caltech 256[1] is a computer vision dataset with 30607 images belonging to 256 different classes; performing a random sampling has a small chance of creating a training dataset that does not contain any image of a particular class, or maybe very few. This would lead to a model unable to correctly classify that class. To avoid this problem, usually stratified sampling is performed: the dataset is partitioned in several splits, e.g. one for each of the 256 Caltech classes, and then the selected percentage is sampled from each split. This leads to a training set where each class has the same proportion with respect to the other classes that it had in the original dataset. This is usually not a problem, but if the classes distribution is highly unbalanced, the model could incur in a phenomenon called classification bias. Take as an example a two class dataset with 100000 elements, of which 99000 belonging to the first class and 1000 to the second class. Most of the algorithms will be biased to assign the first class label at evaluation time, simply because it appeared many more times during training. A way to overcome this issue consists in performing downsampling: instead of picking samples with the same proportion from both classes, the sampling favours the smaller class. However, another possible solution is to use different weights for samples belonging to different classes: some algorithms allow to specify a weight for each class, in this way it is possible to give more importance to the class that contains less samples. Finally, another technique that is used not very often is resampling: each time an element is sampled from the dataset, it is put back again and it is possible to sample it another time; this voluntarily introduces duplicates.

Then, once the three sets have been created, the training phase can start. At this point, depending on the type of validation that has been chosen, different things can happen. A possibility is to simply use the validation set to evaluate each one of the training sessions and pick the best one: this process is known as grid-search[2], due to the creation of a sort of grid to compare all the possible combinations of hyper-parameters chosen. However, it is also possible to perform validation with

[1] http://www.vision.caltech.edu/Image_Datasets/Caltech256/

[2] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

a different technique: "K-Fold Cross Validation" [33] (KCV). This method helps to reduce overfitting, at the cost of a much higher computational time. Longer algorithms or huge datasets typically do not use it, since the training time would be incredibly high. This validation technique works as follows: instead of using a single validation set, the split between training and validation are performed multiple times, creating "K folds", one for each different split. Then, the training and validation is performed separately on each fold, using the respective training and validation set. Finally, the results of all the folds are averaged and compared with the other training sessions. This means that, if the standard validation optimized 20 combinations of hyper-parameters, a K-Fold Cross Validation with 4 folds has to perform $20 \cdot 4 = 80$ separate training sessions. However, this technique is especially useful for small or very small datasets, since it allows to perform a meaningful validation even with few samples. Moreover, there is a variant of this algorithm, called "Leave-One-Out Cross Validation": it still follows the general rule of the KCV algorithm, but in this case the validation set has size equal to one; a single element is picked each time and used to perform the validation. This allows to assign more samples to the other two sub-sets, the training and test ones: this can be helpful for very small datasets.

5.1.5 Evaluation metrics

Finally, once the validation phase has ended, the best model is typically re-trained over the whole training and validation sets together. Then, after the test has been performed, the results must be collected and analysed. There are a lot of possible measures to collect from a machine learning algorithm and some depend on the specific model that has been used. However, some of the most popular and used ones are the accuracy, the precision and the recall, which are derived from other measures [34].
At first, the confusion matrix is created, which is a plot that contains for each class the number of correctly assigned samples or wrongly predicted ones. An example of such matrix can be seen in Figure 5.4: each element of the matrix has a specific name, using a binary classification problem as example.

The class containing the samples of interest (e.g. malicious traffic, cancer images, ...) is called "positive" class, while the other is called "negative". The cell of the confusion matrix containing the number of positive samples that are correctly predicted as positive represents the "True Positive" measure, while the cell containing the negative samples that are correctly classified as negative represents the "True Negative". The other two cells contain the "False Positive" and "False Negative" measures, that represent respectively the samples that are erroneously labelled as positive and the samples erroneously labelled as negative. At this point, the accuracy can be defined as:

$$acc = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP, TN, FP and FN are respectively the true positive, true negative, false positive and false negative. The accuracy simply represents the number of correctly labelled samples, both positive and negative ones. However, this measure alone

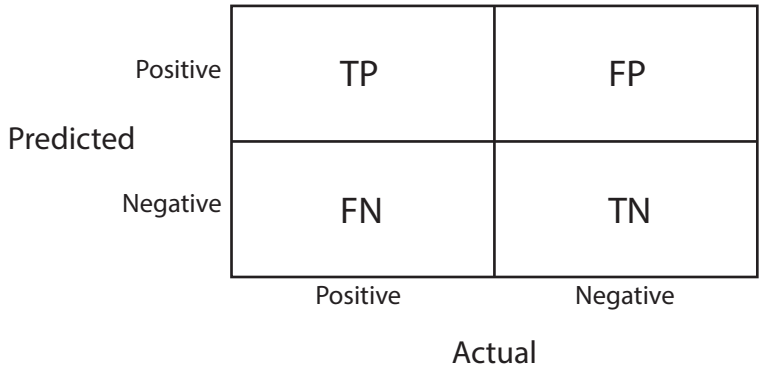

Figure 5.4: The confusion matrix for binary classification.

could not be enough: for example, a test set contains 10000 elements, of which 100 in the positive class and 9900 in the negative one; after the test phase, the obtained accuracy is equal to 99%, which is pretty high. Using only the accuracy, this would be considered an almost perfect model, but the truth may be that the model is assigning the negative class to all the samples; in fact $9900/10000 = 0.99$. The other two measures that are typically used to find these problems are precision and recall. Precision is computed as:

$$p = \frac{TP}{TP + FP}$$

and it represents the correctly classified positive samples among the samples that have been labelled as positive. It can be computed also for negative samples, if necessary. High precision means that when the model says that a sample is positive, it is quite probable that it is truly positive. Instead, recall is computed as:

$$r = \frac{TP}{TP + FN}$$

and it represents the number of positive samples that have been labelled as positive. A high recall means that the model is good at finding the positive samples. Using the example explained above, the precision would have been $p = \frac{0}{0+0}$ and the recall would have been $r = \frac{0}{0+100}$ and an alarm would have triggered.

Furthermore, many more measures exists to evaluate the performances of a classifier: other examples are the balanced accuracy and the F-score. Both measures aim to solve the problem of the simple accuracy explained above. The balanced accuracy is computed as the average of the accuracies obtained by each class individually:

$$acc_{balanced} = \frac{\frac{TP}{TP+FN} + \frac{TN}{TN+FP}}{2}$$

where the two terms in the numerator represent the number of positive samples correctly classified as positive and the number of negative samples correctly classified as negative. Still using the same example explained above, the balanced

accuracy would have been $acc_{balanced} = \frac{\frac{0}{0+100} + \frac{9900}{9900+0}}{2} = 0.5$. Instead, the F-score can be computed directly from the precision and recall measures as:

$$F - score = 2 * \frac{p * r}{p + r}$$

where p and r are respectively the precision and the recall. The main difference with respect to the balanced accuracy is the fact that the F-score gives more importance to the correct classification of the positive samples because its equation does not contain any information about the True Negative samples. Instead, the balanced accuracy treats both classes in the same way: in a context where both classes have equal importance, this behaviour is preferred, while the F-score is more suited for problems where a class has more importance than the other (e.g. in intrusion detection the correct classification of malicious traffic is more important than the correct classification of legitimate traffic).

5.2 Classification techniques

As explained in Section 5.1, classification algorithms are mostly part of the supervised family of machine learning techniques and their objective is to predict a label, rather than a continuous number like in regression problems. Moreover, these algorithms can belong to the "shallow learning" or to the "deep learning" family. The difference between these two families mainly lies in the way the model learns its parameters. Shallow models learn the parameters of their statistical model directly from the features of the dataset. Such features are hand-crafted relying on previous knowledge of the domain.
Differently from shallow models, deep learning algorithms learn the values of their statistical model's parameters both from the input features and from the multiple layers of their architecture. This means that while the input features may be given or not, the model learns by itself what are the important characteristics of the dataset to perform the classification [35]. The typical example of input without hand-crafted features is images because the input is simply the tensor created with the pixels values.
The shallow models presented in this work are Decision Trees, Random Forests and Support Vector Machines. Instead, the deep learning models described are neural networks.

5.2.1 Decision tree

A decision tree [36] model is based on a tree structure, where each internal node is labelled with an input feature. Starting from the root, the source set is split into subsets following specific splitting rules. Then children of an internal node can be other internal nodes or all the possible values of the target class. Also, in this case, the simplest form is a binary classifier, as seen in the example of Figure 5.5.

Some examples of splitting measures are the Gini index and Information gain. Generally speaking, the split algorithm works top-down: at each step it chooses the

feature that best splits the set, relying on the mentioned measures. For example, the Gini index for a given node t is computed as:

$$GINI(t) = 1 - \sum_j [p(j|t)]^2$$

where j is the class index and $p(j|t)$ is the frequency of class j at node t. This index measures the "purity" of the split, reaching its maximum value when the number of samples is equally distributed among all classes ($1 - \frac{1}{n_c}$ with n_c the number of classes) and the minimum value when all the samples belong to the same class after the split ($1 - 1 = 0$). A lower value of the index means that the split can better distinguish between the target classes.

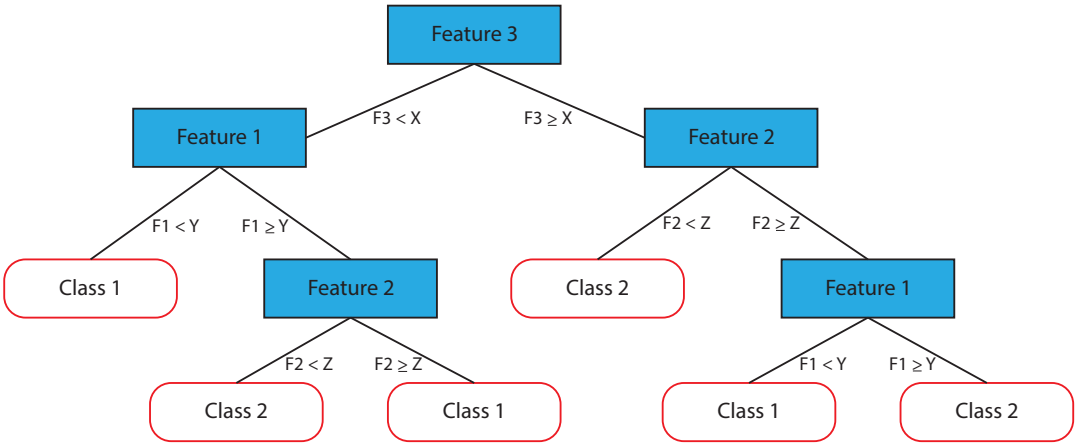

Figure 5.5: An example of a binary decision tree classifier.

Among the main advantages, there is the simplicity of interpretation of the model: differently from other ones, it is clear why each decision has been taken. Moreover, this model can perform well on big datasets and does not require particular preparation of the data. Unfortunately, there are some limitations: decision trees are not very robust to changes in the training dataset, plus the learning of an optimal decision tree is NP-complete; for this reason, the attribute order to perform the split at each step is chosen in a "greedy" way, choosing the attribute that leads to the best impurity index at each step, rather than finding the optimal order of attributes. Finally, the solution given by a decision tree can easily produce overfitting, needing further mechanisms to overcome it, such as pruning techniques. Pruning can be performed both as "pre-pruning" or "post-pruning", the common idea is to remove nodes that lead to a too precise split between classes and this can be done at training time (pre) or after the creation of the tree (post).

5.2.2 Random forest

Random forest [37] is a typical example of model ensemble, which consists in the usage of multiple models to increase performances. Random forests are composed by multiple decision trees and have as output the mode (i.e. the most recurring one) of the classes predicted by individual trees. The standard algorithm comes

from the union of two concepts: feature bagging and bootstrap aggregation.
The first one means that at each split during the training, a subset of random features is selected (typically the square root of their number). The second one consists of a random subsampling with replacement as input for each tree of the forest. So the algorithm can be summed up as follows: T random subsamples with replacement are created from the dataset, for T decision trees to be trained. Each tree is created separately and at each split a random subset of the features is taken into account, rather than the whole pool.

This process leads to some advantages; first of all the decorrelation between the trees. Also, the features that are chosen the most times in the higher splits are naturally identified as the most relevant ones. Finally, the accuracy is generally higher than simple decision trees, at the cost of a more complex model and less easy to interpret results and of course higher training times.

5.2.3 Support Vector Machines

The aim of Support Vector Machines (SVM) is to find the best dividing hyper-plane among the samples [38]. In their standard version SVM try to find the hyper-plane that has the highest distance from the samples so that all the samples belonging to the same class are contained in the same side of the plane: this is called the large margin problem. An example of a boundary created by a linear SVM is shown in Figure 5.6.

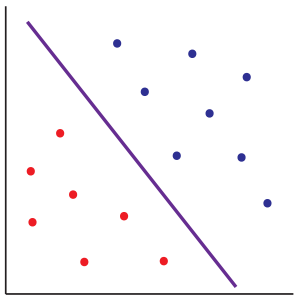

Figure 5.6: An example of a trivial SVM binary classification.

Usually, in real-world applications, samples are hardly exactly separable with a hyper-plane: the presence of outliers is quite common in every dataset. Differently from other algorithms, SVM can integrate a regularization technique directly in their mathematical formulation. It is possible to add a so-called "slack variable" to the optimization problem that allows some errors in the classification to take into account the presence of outliers. This is called the soft margin problem.
Furthermore, one of the main advantages of an SVM model is the fact that it only needs to store support vectors to perform classification. Support vectors are the only points that actually determine the position of the separating hyper-plane since they are the nearest ones to such plane.
To classify a new sample, in case of a linear SVM, the equation is the following:

$$y = \langle w, x \rangle + b$$

where x is the sample and b is the so-called bias term, while the $\langle w, x \rangle$ term represents the scalar product between the two vectors. The sample is assigned to the positive class if $y > 0$ and negative otherwise. The w term is calculated as:

$$w = \sum_i \alpha_i x_i y_i$$

with x and y being the samples with their corresponding class, while α is a term that is equal to 0 for all the points, except for the support vectors.

Finally, it is possible to apply kernel functions to separate classes that are not linearly separable. Using Figure 5.7 as a reference, imagine a binary classification problem with two features and samples distributed as shown: a linear SVM clearly can not find a separating hyper-plane to correctly classify the samples. A possible solution is the introduction of kernels, that allow the mapping of points in a higher-dimensional space: keeping the same example of Figure 5.7, the points can be mapped with the new dimension $z = (x^2 + y^2)$ (which is the equation of a circumference); now the points are linearly separable in the new space. Unfortunately, mapping all the points of a dataset in a higher-dimensional space is computationally too expensive. The so-called "kernel trick" [39] is a method that allows introducing the kernel function inside the scalar product of the linear SVM, that now becomes:

$$y = \langle w, \phi(x) \rangle + b$$

where $\phi(x)$ is the kernel function. Many possible kernel functions can be used. The most common ones are the linear kernel, the polynomial kernel, the radial basis function (RBF) kernel and many others.

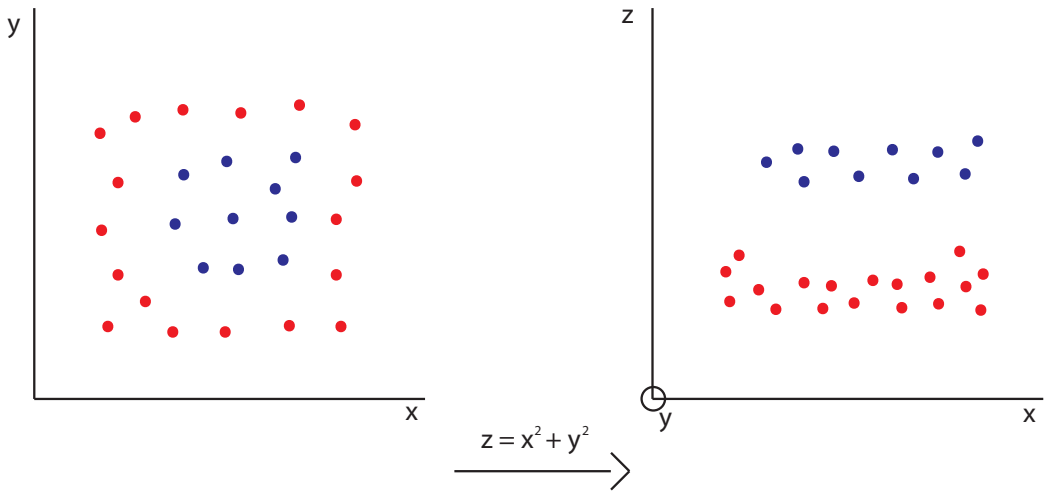

Figure 5.7: An example of a kernel mapping, from 2D to 3D.

5.2.4 Neural Networks

The basic architecture of a Neural Network [40] (NN) is a multilayer stack of simple modules, also called neurons: in this case, they are also called fully connected NN

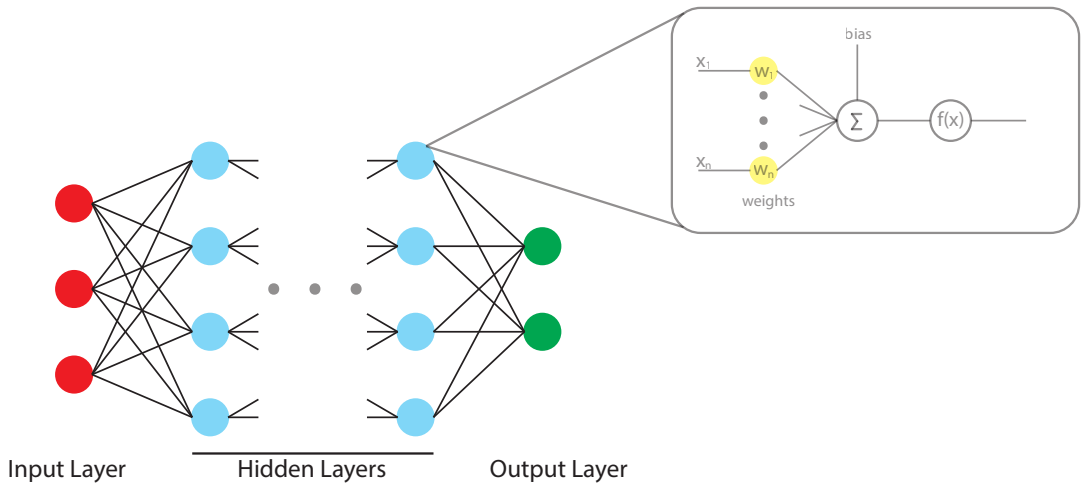

Figure 5.8: An example of a Neural Network, with the detail of a single neuron.

or feed-forward NN. What happens inside a neuron is a weighted sum of the inputs of that neuron, followed by an activation function, as better explained in Figure 5.8.

The activation function carries out an essential role inside the network: in fact it introduces the non-linearities that are fundamental to distinguish the network from a shallow model. Actually, several types of activation functions exist, with the most common being:

Sigmoid $f(x) = \frac{1}{1+e^{-x}}$
historically famous, but with many disadvantages. It saturates (i.e. has un upper or lower limit for the output of the function), the output is not zero-centred and contains the exponential, which is expensive to compute;

tanh $f(x) = tanh(x)$
zero-centred, but still saturates;

ReLU $f(x) = max(0, x)$
Rectified Linear Unit does not saturate in the positive region, it is very efficient to compute. Unfortunately the output is still not zero-centred and a negative input produces an output equal to zero;

PReLU $f(x) = max(\alpha x, x)$
Parametric ReLU. All the advantages of the ReLU, but will not be killed by negative inputs.

The issue with non-zero-centred activation functions is due to the peculiar way a NN is trained: an always-positive (or negative) output of an activation function will slow the process of convergence of the network [41], as better explained below. Usually, ReLU or PReLU are the chosen ones. Finally, Figure 5.9 shows these functions, to better visualize what already explained.

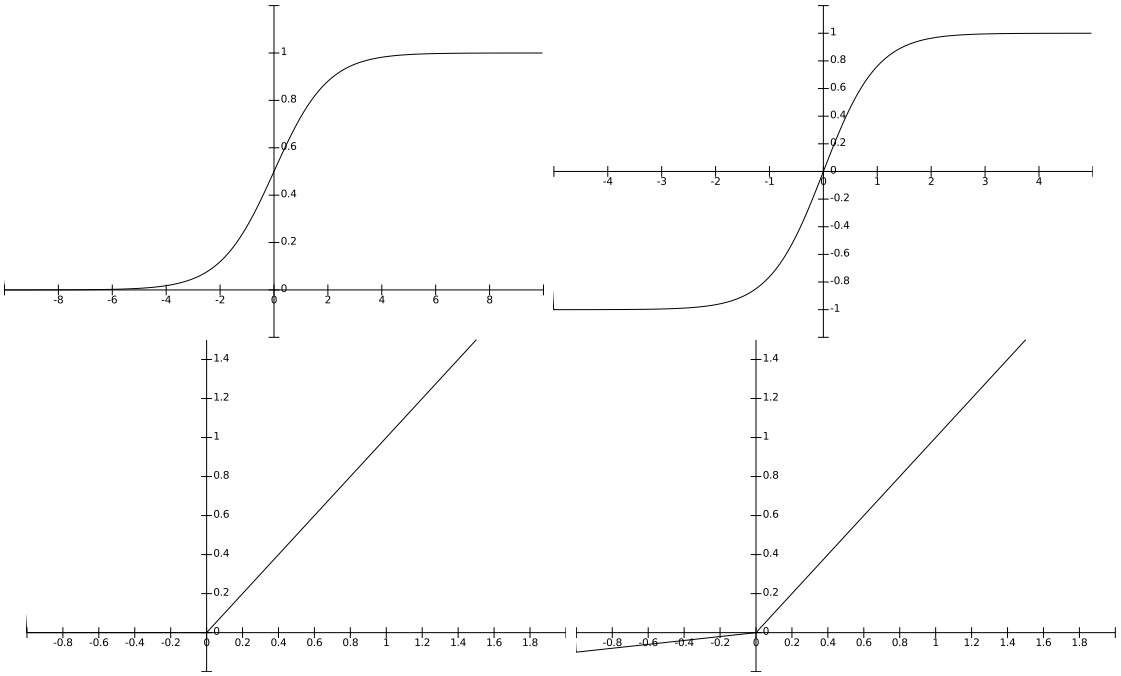

Figure 5.9: Examples of activation functions. From left to right, on the top: sigmoid and tanh; on the bottom: ReLU and PReLU.

At the end of all the fully connected layers, a softmax function compresses the outputs to have a sum equal to one, to be interpreted as a probability. The equation of the softmax function is:

$$\sigma(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

where z_i is the current element for which the softmax is computed.

The training of a neural network happens differently from the shallow algorithms explained above: it is composed of two phases, the forward propagation and the back-propagation. Given a single sample, each of its features is typically used as input for a different neuron of the input layer and the output of each neuron is then forwarded to the next neurons, until the end is reached: this is the simplest part and it is called forward propagation.
At this point, the error between the predicted output and the actual label must be computed with a "loss" function (or cost function). There are different possible choices for the loss function, but the most popular one is indeed the logistic loss [42], which is defined as:

$$L(y, p) = -(y \log(p) + (1 - y) \log(1 - p))$$

where $y \in \{0, 1\}$ is the true label and $p = P(y = 1)$ is the probability estimate.
The loss measure is used to perform the second step of the training, the back-propagation. This step consists in the computation of gradients to update the weights of the nodes. More in detail, by using a component of the NN called "optimizer", it is possible to choose among different algorithms: the most common ones are Stochastic Gradient Descent [43] (SGD) and Adam. The objective of these algorithms is to minimise the loss function, proceeding iteratively. After each batch of forwarded input data, the computed loss is used to update the weights and then,

once all the training set has been forwarded through the network, the epoch ends. The number of epochs, i.e. the number of times the whole training set is forwarded through the network, can be a fixed number or the interruption of the training can be triggered dynamically. For example, a common method is to interrupt the training if the loss is not decreasing by a fixed percentage since a certain interval of epochs.

Finally, another type of neural networks exists: Convolutional Neural Networks [44] (CNN). They are designed to process data that come in the form of multiple arrays. Besides images, many other datasets are composed by such form, e.g. sequences and text, audio spectrograms and videos. CNN try to exploit some intrinsic characteristics in this form of data with some additions to the fully connected NN architecture: convolutional layers and pooling layers are added to the architecture. A convolutional layer can be seen as a filter that moves over the input: each time this filter moves, it performs the scalar product between the weights of the filter and the values of the part of the input it is passing over. Like in standard NNs, the output produced by a layer is used as input for the next one. These filters are helpful to grasp local features typical of images and other similar inputs like audio or video tracks. Instead, pooling layers are needed to reduce dimensionality and make the representations more manageable. An example of pooling filter that chooses the max value of an area of the input is shown in Figure 5.10. Some popular CNN architectures are ResNet [45] or Inception [46].

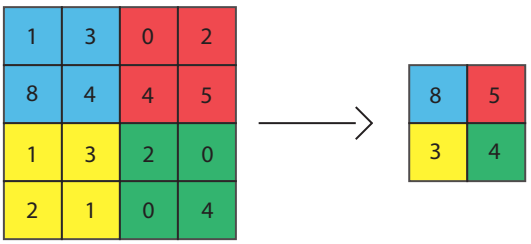

Figure 5.10: An example of a pooling mechanism.

Chapter 6

Solution design

This chapter's objective is to explain the workflow followed during this work's experiments and to describe in detail the different phases that compose it.
The two main steps of the experiments are the creation of the machine learning models that allow to classify network traffic as benign or malign and the integration and deployment of these models inside a working IDS, Suricata. The technical details on IDSes can be found in Chapter 4, while the overview of machine learning techniques is in Chapter 5. Finally, Figure 6.1 shows an high level diagram with the whole workflow of the experiments.

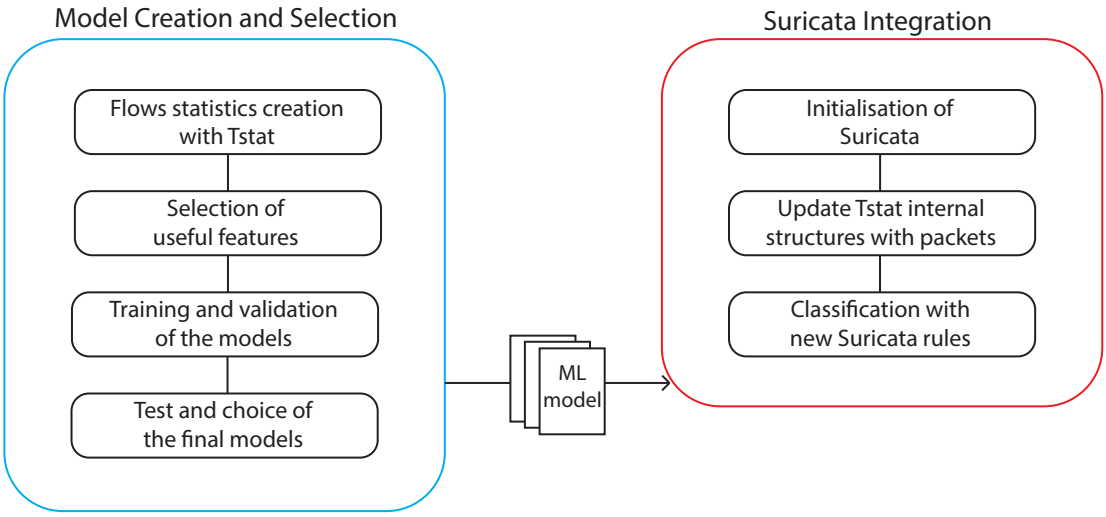

Figure 6.1: The high-level workflow of the proposed solution.

6.1 Creation and selection of classifiers

This phase of the experiments starts from the download of the two chosen datasets, which characteristics have been deeply analysed in Chapter 7, and ends with the creation of several different trained classifiers, that will be later employed in the integration phase. The design followed to obtain this result is the one described in

Section 5.1, composed by a first phase of pre-processing of the dataset, followed by the visualisation of the dataset characteristics and another phase of pre-processing, with different aims from the previous one.
Using the results and the observations of these first steps, the algorithms that better suit the characteristics of the dataset have been chosen. Finally, the training, validation and testing pipelines have been created, in order to produce and store different classifiers, ready for a later use. In this regard, for each of the chosen algorithms have been stored several trained models, depending on the specific training set used to create it. The technical steps to reproduce this workflow are described in Appendix B, while the explanation of the usage of the final tools from a user perspective can be found in Appendix A.

6.1.1 Extraction of statistics

After the download of the network traffic captures in the form of "pcap" files, explained in Section 7.1, the Tstat tool has been used to extract the network statistics in the form of several CSV file. Since the original dataset, counting both the CSE-CIC-IDS2018 and the Torsec one, is composed by around 4400 pcap files, an ad-hoc Python script has been written to perform the Tstat analysis on each one of them. Each type of network attack has been analysed separately, producing one or two CSV files for each attack, depending on the number of days during which the attack has been performed. Each CSV file contains the core TCP statistics that have been described in Section 2.2.1.
Then, each file is used as input of another Python script, which objective is to append the label of each flow: using the information regarding the IP addresses of the attackers machines provided by the CSE-CIC-IDS2018 website (and considering that no IP spoofing technique has been used), each flow is labelled as "Benign" or "Malign". Instead, the flows originated from the Torsec dataset have been all labelled as malign since only attack captures have been used among the ones contained in the Torsec dataset, as explained in Section 7.1.2.

6.1.2 Features selection

The various CSV files that compose the final dataset have been analysed, grouping them by the network attack contained. First of all, the distribution of malign and legitimate traffic among the different attacks has been analysed, as better described in Section 7.2, leading to the removal of one of the attacks, due to lack of positive samples. Then, the set of features of the resulting dataset has been reduced with respect to the original one.

The whole set of features created by Tstat has been presented in Section 2.2.1. Starting from these, only a subset has been chosen to perform the experiments. First of all, from all the different logs available, the richest one has been used: "log_tcp_complete". The "log_udp_complete" log has been excluded due to the lack of statistics created by Tstat: for this type of traffic only 6 useful statistics (3 per direction) are available, way less than the TCP ones.
Then, from the 44 columns of the core TCP statistics, a further selection has been

performed. The IP and port information of both source and destination has been removed from the set because they typically do not carry useful information about the presence of an ongoing attack: this is due to IP address and port spoofing, a technique explained in Section 3.2. Moreover, another couple of columns that have no use in the intrusion detection are the absolute time of the first and last packet observed per flow, for both directions. Instead, the total flow duration has been kept. Finally, some qualitative information about the flow has been discarded, like the presence of CryptoPAn [47] IP addresses or the position of the client/server inside the network (e.g. internal/external). The final selection of features is shown in Table 6.1 and will be used as reference for all the following experiments.

#	**Description**
0	# of packets sent by client
1	RST sent by client
2	# of ACK sent by client
3	# of ACK sent by client without data
4	# of bytes sent by client in the payload
5	# of segments with payload sent by client
6	# of bytes sent by client in the payload, including re-transmissions
7	# of re-transmitted segments by client
8	# of re-transmitted bytes by client
9	# of client segments out of sequence
10	# of SYN sent by client
11	# of FIN sent by client
12	# of packets sent by server
13	RST sent by server
14	# of ACK sent by server
15	# of ACK sent by server without data

#	**Description**
16	# of bytes sent by server in the payload
17	# of segments with payload sent by server
18	# of bytes sent by server in the payload, including re-transmissions
19	# of re-transmitted segments by server
20	# of re-transmitted bytes by server
21	# of server segments out of sequence
22	# of SYN sent by server
23	# of FIN sent by server
24	Flow duration (in ms)
25	Client first payload since start (in ms)
26	Server first payload since start (in ms)
27	Client last payload since start (in ms)
28	Server last payload since start (in ms)
29	Client first ACK without SYN (in ms)
30	Server first ACK without SYN (in ms)

Table 6.1: The chosen Tstat features.

6.1.3 Training and Validation

Due to the huge size of the dataset, the choice of the classification algorithms is naturally narrowed. Algorithms like K-Nearest-Neighbours that do not scale well with big datasets have been directly excluded and the chosen ones have been the ones described in Section 5.2: Random Forest, Support Vector Machines and Neural Networks. SVM have been included, but their "kernel trick" explained in Section 5.2.3 typically takes too much time to converge with big datasets: for this reason a linear SVM has been used.

Then, among the many frameworks available for machine learning, the chosen one has been Scikit-learn[1]. It is a popular Python framework that offers many ML algorithms, pre-processing techniques and evaluation methods. Moreover, one of its points of strength is the ease of use: it allows the creation of a Machine Learning classifier with a single line of code and the training of such model with another line of code.

The training phase has been designed with the aim of making possible to run it many times, over many days, with different inputs and algorithms and with the ability to store the output of each training to analyse it at a later time. With these requirements fixed, the logical blocks that compose the training phase are shown in Figure 6.2.

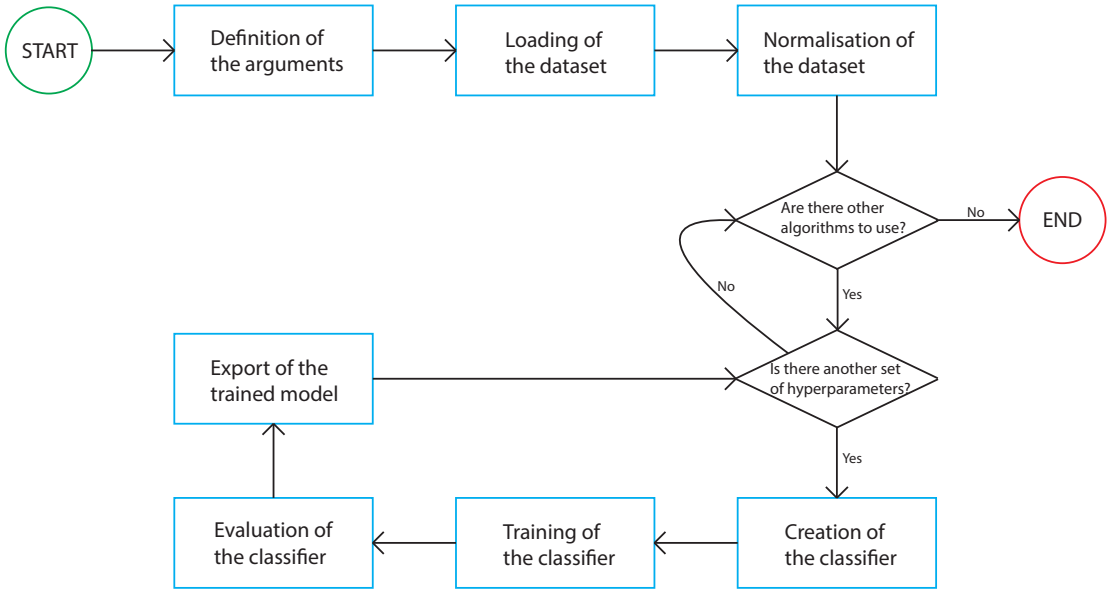

Figure 6.2: The logical sequence of actions that are part of the training phase.

First of all, the configuration file relative to the training phase is read and used to setup some internal variables. As better explained in Appendix B.3.2, the configuration file contains information about the path to the dataset or to the subsets of attacks of interest for the specific training, the path to the output folder

[1] https://scikit-learn.org/stable/

for the storage of the created models and the list of features that will be used for the training phase.

Then, the appropriate dataset is loaded from its folder. At first, the set of chosen flows is split between the positive and negative class, i.e. malign and benign flows; then, the two sub-sets are further split into training, validation and test sets and then merged. The reason for this choice has been explained in Section 5.1.4. Once the datasets have been created, the standardisation is performed by fitting a Scikit-learn "StandardScaler"[2] on the training set and then transforming the training and validation sets with the fitted scaler. The standardisation is performed in this way, because the validation set, used as evaluation set for the training, must not be used to fit the scaler to avoid the introduction of bias.

After the creation and standardisation of the training and validation sets, for each of the selected algorithms, a cycle over the combinations of hyper-parameters is performed. For each iteration, the appropriate classifier is instantiated and then trained with the training set. The obtained classifier is evaluated with the validation set and stored in the chosen output folder.
This process of storing the model happens as follows: first, a Python object is created, containing the set of hyper-parameters used for the current iteration; instead, another Python object contains the evaluation measures obtained on both training and validation sets, which are described below. Then, the two Python objects are encapsulated inside another Python object, that contains these two objects together with a list of the names of the network attacks used for the specific experiment (or eventually a single name, in case only one dataset is used to train the classifier) and with the trained model itself. This final Python object, which structure is summed up in Listing 6.3, is then serialised using the joblib[3] library and stored in an external file. The set of stored statistics, inside the "stats" object, includes the accuracy, balanced accuracy and F-score obtained on both training and validation sets, together with the time needed for the training. The technical description of these statistics can be found in Section 5.1.5.

Once several models have been created, the validation step has been performed. The process to choose among the different trained classifiers has been automated by cycling over the stored Python objects, now de-serialised, and selecting the one with the highest F-score: this process of selecting the best set of hyper-parameters by cycling over all the possible combinations is known as grid-search, as detailed in Section 5.1.4. As explained in Section 5.1.5, the F-score for a binary classification problem can be interpreted as the ability of correctly classifying positive samples: its value increases with a higher number of True Positive samples and decreases with a higher number of False Positive or False Negative samples. The correct classification of legitimate traffic is not considered by this measure (the True Negative value), differently from other measures like the balanced accuracy. Since the objective of a model to be integrated into an IDS is to correctly classify malicious traffic with a low rate of False Positive or False Negative samples, this measure has been

[2] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[3] https://joblib.readthedocs.io/en/latest/

```
{
        'params' : {
                'hp1' : hyper-param 1,
                'hp2' : hyper-param 2,
                ...
                'hpN' : hyper-param N,
        },
        'stats' : {
                'stat1' : statistic 1,
                'stat2' : statistic 2,
                'stat3' : statistic 3,
                ...
                'statN' : statistic N,
        },
        'attacks' : [
        "Attack01",
        "Attack02"
        ],
        'model' : trained model
}
```

Figure 6.3: The structure of the exported model, represented in JSON format.

preferred.
As a result of the validation phase, each one of the three algorithms (NN, RF and SVM) has a single trained classifier for each one of the attacks in the dataset: the next phase has the objective of evaluating these models and choosing the best algorithm for each attack.

6.1.4 Testing

Finally, the test phase decides which models are most suited to be integrated in the IDS. The optimal solution would be to select as much as possible of the best performing models, but it must be kept in mind that each model would add an overhead to the performance of the IDS and one of the objectives of this work is to obtain a working IDS, by interfering as less as possible with its standard routine and performance after the integration. For this reason, the resulting models have been compared both under the aspect of classification results and the time required to perform the predictions.

Hence, the best models selected after the previous phases have been trained again, this time over a bigger training set, composed by the former training and validation sets. Each classifier has been trained using the set of hyper-parameters stored in the corresponding Python object, over the same set of attacks used for its first training. To perform this step, a process similar to the one previously described in Figure 6.2 has been followed, with two major differences: the set of

hyper-parameters is fixed and the standardisation scaler is fit over the new training set, composed by the old training and validation sets. The obtained models have been exported in the same way as the previous ones, to perform more analysis.

Then, the results of the different models have been compared, as explained in Section 5.1.4, using the same set of measures used during the validation phase, together with the average time needed to perform a prediction. Additionally, to better visualise the results, the confusion matrix has been created for each classifier. All the results obtained from both validation and testing phases have been collected and presented in Chapter 8.

6.2 IDS integration

Once the machine learning models are ready, a way to actually integrate their usage into the Suricata IDS has been designed. Using the Suricata workflow described in Section 4.4.1 as a reference, some of its modules have been modified. The resulting workflow is shown in Figure 6.4, where the added functionalities have been highlighted with respect to the original workflow. The initialisation phase and the decoding and detection modules have been modified, as explained in the following sections. Moreover, a Python script to perform the classification has been created.

Through the whole integration process, the Tstat API functions provided by its library (libtstat) have been used to communicate with Tstat and obtain the TCP flows statistics. The five API functions provided by Tstat are:

1. int tstat_init(char *config);
2. void tstat_new_logdir(char *file, struct timeval *pckt_time);
3. int tstat_next_pckt (struct timeval *pckt_time, void *ip_hdr, void *last_ip_byte, int tlen, ip_direction);
4. tstat_report *tstat_close (tstat_report *report);
5. void tstat_print_report (tstat_report *report, FILE *file).

The first two functions, `tstat_init` and `tstat_new_logdir`, have to be called during the initialisation of Suricata, because they are needed to setup Tstat internal structures. Instead, `tstat_next_pckt` has to be called in a place where the IP packet is available inside Suricata, since it needs a pointer to the IP header. Finally, the `tstat_close` function is used to flush to file the collected statistics, while `tstat_print_report` simply prints on screen a report of the flows analysed by Tstat.

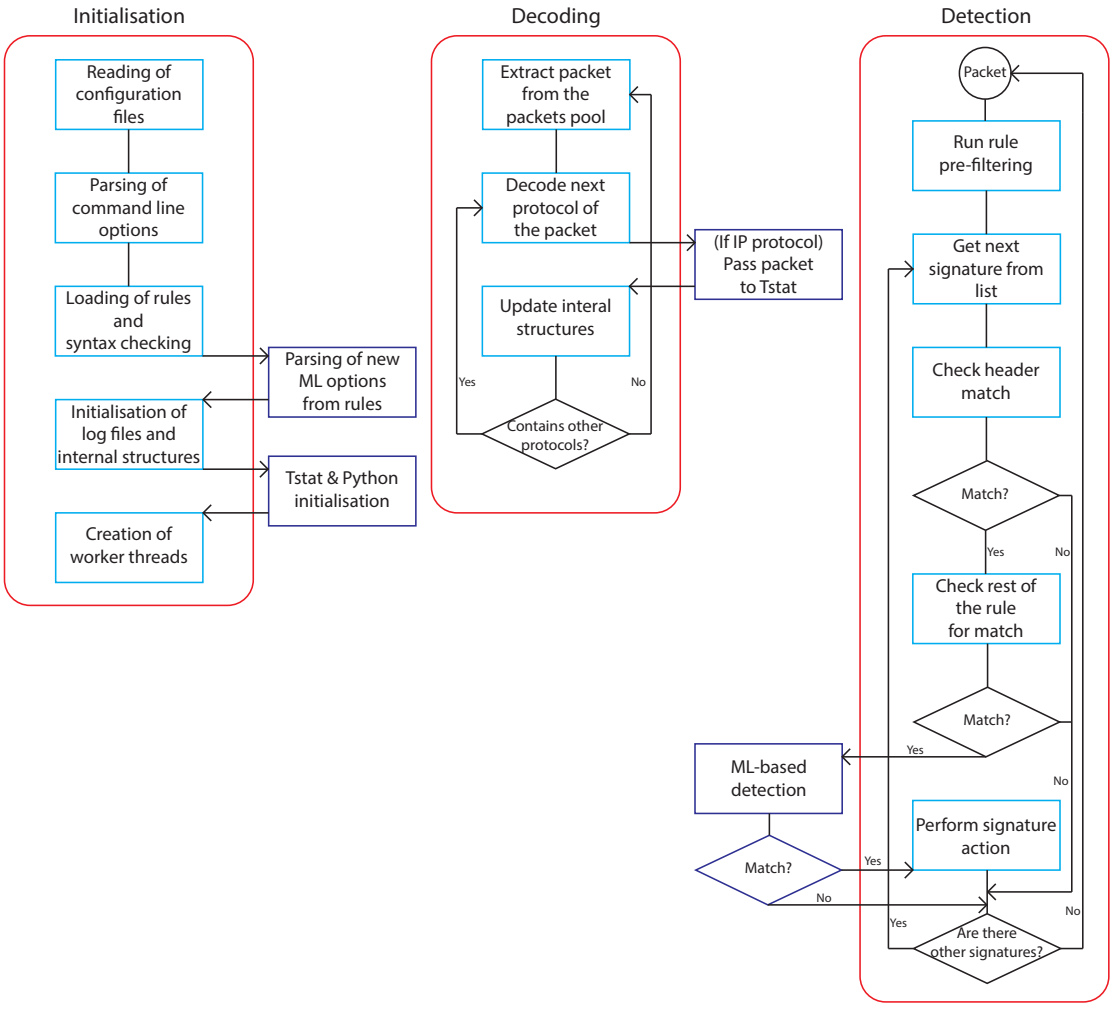

Figure 6.4: The modified workflow of the Suricata modules.

6.2.1 Suricata initialisation

Inside the Suricata initialisation phase have been inserted two more sequences of initialisation: the Tstat initialisation and the Python interpreter initialisation. The Tstat initialisation is needed by the Tstat tool, integrated inside Suricata by using two of the five API functions provided by libtstat, while the Python interpreter initialisation is needed to be able to use the C/Python API[4], as deeply explained in Appendix B.

The best position to initialise Tstat is inside the main function of Suricata, before the initialisation of the threads that will perform all the Suricata activities. After all the basic checks and configurations of internal modules have been completed, the Tstat initialisation function has been inserted. In this way, it is possible to take advantage of the Suricata internal logging system to produce informative logs about the status of the integration mechanism.
This process performs many preparative actions before actually calling the Tstat

[4] https://docs.python.org/3.8/extending/embedding.html

API `tstat_init`. Some of these actions are directly needed to obtain some configuration parameters for Tstat, while other are useful to setup some internal variables that will be used later by the decoding module. First of all, the configuration file used for the integration is retrieved (called "integration.conf"): in case it is not found, a critical error is emitted with the Suricata logging system and the execution is interrupted since this file is necessary to continue with the initialisation. Then, the `tstat_init` and `tstat_new_logdir` API functions are called and the Tstat initialisation sequence ends.

Once Tstat has been successfully initialised, the Python interpreter performs some similar actions. It is initialised through some of its API functions and then a configuration file containing information about the ML models is read, with the result of storing the ML models inside an ad-hoc data structure, ready to be used at a later stage. The reason for this choice is the fact that as much of preparative work as possible has been moved inside the initialisation phase, to allow a faster execution of the detection phase. For this same reason, the pointer to the Python function contained inside the external Python script is stored: in this way the function is ready to be called as soon as the needed arguments are ready, as explained later in the classification phase.

6.2.2 Packet decoding

The remaining part of the integration concerns the creation of the Tstat flows statistics and the subsequent detection of malicious flows. The optimal place to perform this step of the integration has been located inside the detection module of Suricata, where the packets are directly available and can be passed to Tstat, as required by the `tstat_next_pckt` function. More in detail, the `DecodeIPV4` Suricata function is the exact place where the IP packet bytes are available. However, the whole section of integration inside this function has been considered a critical section, since multiple threads operate concurrently in Suricata: for this reason a mutex lock has been required before the beginning of the actual integration section and has been released after all the operations have been performed. However, this integration step performs only a single action: the IP packet is passed to Tstat, that automatically updates its internal structures with the packet data.

6.2.3 Packet classification

The final phase of the integration happens inside the Suricata detection module. The ML-based detection has been placed after the normal Suricata process of signature matching: in this way, the ML classification is performed only if all the other options of a rule provide a match, allowing to further reduce the added overhead. The ML-based detection happens in different stages: first, the flags of the current signature analysed by Suricata are read and the new machine learning flags are retrieved; then, the statistics of the flow of the current packet are obtained from Tstat; finally, the ML flags and the statistics are passed to the external Python function, that performs the prediction and returns the outcome to Suricata.

More in detail, the flags are obtained from a Suricata internal structure (a C struct) that contains information about a specific signature. This structure has been populated by Suricata during its initialisation phase and contains, among the other data, the options of the specific signature (e.g. action, protocol, id). If the ML options are found, the ML-based detection process starts, otherwise the normal Suricata workflows continues. These special flags contain information about which ML models have to be used during the detection phase of a specific signature.
Instead, the statistics related to the current packet's TCP flow are obtained from Tstat using a sixth API function: this function has been created ad-hoc to retrieve statistics of a single flow because the other libtstat functions do not allow to do so. A specific flow is identified inside the Tstat internal data structures by using the IP address/port pair of both source and destination of the analysed packet, plus a timestamp to not find eventual old flows.
Finally, the external Python function performs very few actions: given the already loaded models, the statistics of a flow and the list of flags, it performs the prediction with each one of the models selected by the flags and returns the outcome to Suricata.
Since the ML-based detection happens at the end of the Suricata detection phase, if the outcome is positive Suricata performs the action specified in the signature, while it continues with the analysis of the next signature if the outcome is negative.

Chapter 7

Dataset analysis

This chapter contains an in-depth description of the datasets used for the experiments described in Chapter 8. Then, it proceeds with an overview of the features selected from the ones extracted by Tstat, among the ones described in Section 2.2.1. Finally, Section 7.2 explores the statistical characteristics of the final dataset.

7.1 Dataset composition

The dataset used for this work's experiments comes from two different sources. The first source is a publicly available dataset, that contains several attack types, while the second one has been collected by the Torsec group[1] in Politecnico di Torino and is focused on three families of network attacks. The availability of public datasets containing network attacks is quite rare, since many are collected internally and can not be shared due to privacy issues. Moreover, network behaviours and patterns change with time, as well as intrusions. For these reasons, the most up-to-date datasets have been chosen, containing a wide set of attack types.

7.1.1 CSE-CIC-IDS2018

This dataset[2] is part of the collaboration between the Communications Security Establishment (CSE) and the Canadian Institute for Cybersecurity (CIC). Among the different datasets available on their website, this is the most complete one and also recent enough to contain novel attack methodologies. The main objective of this dataset is to develop a systematic approach to generate a comprehensive benchmark dataset for intrusion detection. The final dataset includes seven different attacks, which technical description and the related tools can be found in Chapter 3. The attacks are the following:

- Brute force, performed with Patator on both FTP and SSH;

[1] https://security.polito.it/

[2] https://www.unb.ca/cic/datasets/ids-2018.html

- Heartbleed, performed with Heartleech tool;
- Botnet, performed with Ares tool;
- DoS, performed with Hulk, GoldenEye, Slowloris, Slowhttptest;
- DDoS, performed with LOIC and HOIC on UDP, TCP and HTTP;
- Web attack through SQL injection and XSS;
- Infiltration, exploiting an application vulnerability an malicious emails.

The attacking infrastructure includes 50 machines, while the victim organization has 5 departments and includes 420 machines and 30 servers, with both Windows and Linux machines. The generated traffic is split in two categories, benign and malign, and a profile of network traffic usage is created for each one of them. In particular, the benign profile is designed to extract the abstract behaviour of a group of human users: it tries to encapsulate network events produced by users with machine learning and statistical analysis techniques. The encapsulated features are distributions of packet sizes of a protocol, number of packets per flow, certain patterns in the payload, size of payload, and request time distribution of protocols. Once the benign profiles have been created, the CIC-BenignGenerator tool has been used to generate the benign background traffic for the attack scenarios [48].

The CSE-CIC-IDS2018 dataset can be downloaded with the Amazon AWS Command Line Interface[3] (AWS CLI), which is available for Microsoft, Linux and Mac systems. Then, after the statistics extraction with Tstat, as explained in Section 6.1.1, the resulting CSV files have been used to perform the required experiments.

7.1.2 Torsec dataset

The dataset provided by the Torsec group is not publicly available. This dataset is much smaller with respect to the CSE-CIC-IDS2018 one and is composed by both legitimate and malicious traffic captures. The legitimate traffic has been created using Chrome, Firefox and Edge web browsers, while the malign one is split in web scrapers, DoS attacks and vulnerability scanners. The reason for the presence of this dataset inside this work is to have the possibility of comparing network traffic coming from a different distribution, i.e. created and captured with different techniques with respect to the CSE-CIC-IDS2018 one.
However, the only portion of traffic used to perform this work's experiments is the one containing DoS attacks. This choice has been done to increase the size of the DoS dataset of CSE-CIC-IDS2018. Instead, the legitimate traffic contained inside the Torsec dataset has been discarded since the CSE-CIC-IDS2018 already contains a lot of legitimate traffic, as shown in Section 7.2. The statistical differences between these two datasets can be found in Section 7.2, together with other measures and

[3] https://aws.amazon.com/it/cli/

graphs. The DoS attacks of this dataset have been performed with some tools already described in Chapter 3 and are: Goldeneye, Hulk, Rudy, Slowhttptest and Slowloris.

7.2 Traffic analysis

The final dataset composed by merging CSE-CIC-IDS2018 and Torsec ones has been analysed, to take better decisions for the design of the training phases. First of all, the histogram of the two classes (malign and benign) distribution across the attacks has been produced and it is shown in Figure 7.1.

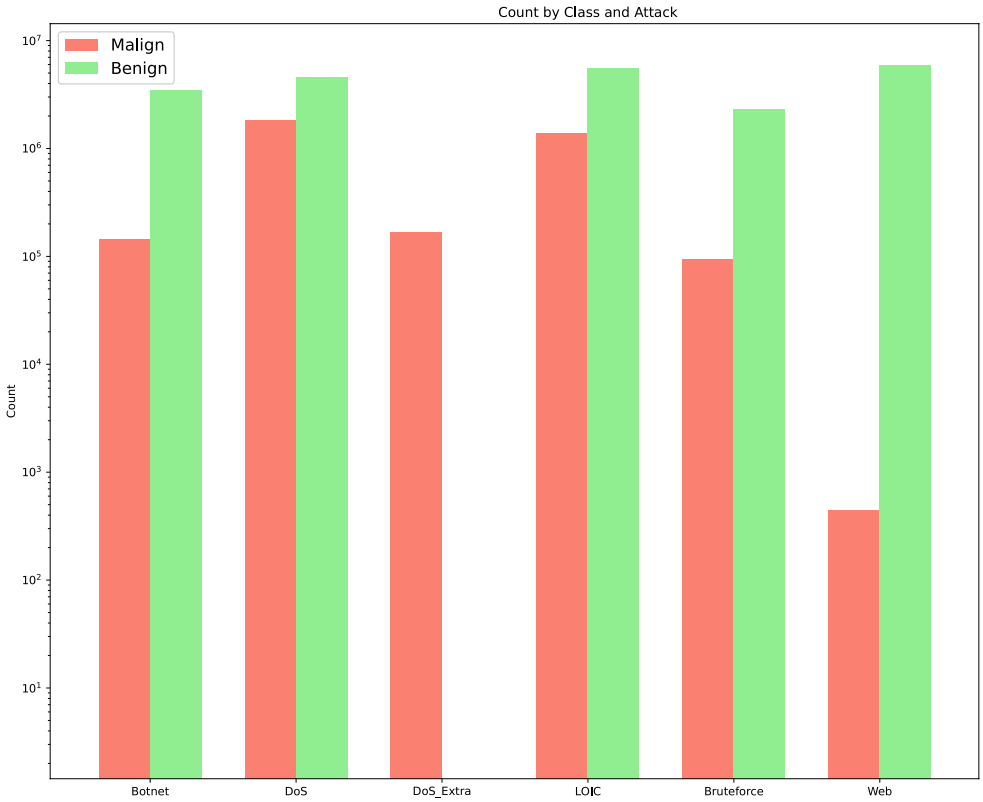

Figure 7.1: The class distribution of TCP flows across the different datasets.

The DoS attack captures from the Torsec dataset have been called "DoS_Extra" to distinguish them from the other ones. Also, the count scale is logarithmic, to allow a better visualisation of the global distribution. Each pair of columns represents the amount of legitimate and malicious TCP flows contained in the respective dataset: in fact, each set of traffic captures of the CSE-CIC-IDS2018 dataset contains both legitimate and malicious traffic.
The first thing to notice is the fact that malign flows are always less than the benign ones, in each of the different attack types. This has to be expected, since anomalous traffic is typically less frequent than the normal one. The ratio between benign and malign traffic can be visually computed by looking at the logarithmic scale and goes from 2:1 to around 20:1 for most of the attacks. The only attack

that clearly have a different ratio is the web attack. To have a better idea of the different ratios, the exact numeric distribution can be found in Table 7.1.

Attack	**Benign**	**Malign**	**Total**
Botnet	3.413.095	142.925	3.556.020
DoS	4.570.259	1.834.752	6.405.011
DoS_Extra	0	166.584	166.584
LOIC	5.504.503	1.371.621	6.876.124
Bruteforce	2.312.771	94.207	2.406.978
Web	5.893.766	438	5.894.204
TOTAL	**21.694.394**	**3.610.527**	**25.304.921**

Table 7.1: The numeric class distribution of TCP flows across the different datasets.

The highly unbalanced ratio of the web dataset could be easily solved by removing samples of the most present class, the negative one in this case. However, it has been chosen to not remove the samples and perform the training with the unbalanced dataset, to see how well can the models adapt to this situation. The final dataset is composed by more than 25 millions of TCP flows, of which the 15% belong to the positive class.

The next type of analysis performed consists of a comparison between the average values of some features, differentiated between positive and negative flows. The set of chosen features is composed by the average size in bytes of the client and server payloads and by the average duration in milliseconds of a flow. The results of this analysis are presented in Table 7.2.
The first thing to notice is the fact that the negative class (i.e. the legitimate traffic) has almost the same distributions of values among each attack type: this means that more or less the type of traffic modelled for the legitimate users has been consistent among all the different experiments during the creation of the dataset. Instead, the average values of the positive class are mostly different with respect to the ones of the negative class.

Attack	**Client payload**		**Server payload**		**Duration (ms)**	
	Pos.	**Neg.**	**Pos.**	**Neg.**	**Pos.**	**Neg.**
Botnet	324	986	129	10205	14	36683
DoS	346	950	931	13719	1120	38822
DoS_Extra	752	n.a.	25434	n.a.	6382	n.a.
DDoS	240	929	941	14902	4991	36764
Bruteforce	1934	939	2664	12290	367	40131
Web	30615	956	69802	15764	33476	47274

Table 7.2: The average value of some features of the TCP flows, split by class and attack type.

Chapter 8

Experimental results

This chapter describes the results obtained during the different phases of the experiments performed in this work. Since one of the objectives of this work is the creation of models capable of detecting malicious attacks without adding too much overhead to the IDS where they have been integrated, the first section of this chapter focuses on the classification ability, while the second one concerns the time dimension. All the time performances have been measured on an Ubuntu PC with an Intel i5-7200U 2.5GHz CPU and 12GB of available RAM.

8.1 Classification results

As explained in Section 5.1.5, there are many possible measures to evaluate the goodness of a classifier. The ones chosen for this work's experiments have been the accuracy, the balanced accuracy and the F-score. The following sections contain the result obtained with each one of the chosen algorithms and then a comparison between the best models is performed.

8.1.1 Random Forest

The random forest classifier[1] has been trained with different combinations of hyperparameters, changing the number of estimators and the maximum depth allowed for each decision tree of the forest. The numbers of estimators that have been tried are 20, 50, 100 and 150, while the possible maximum depth has been set to 16, 24 or "unlimited". The only parameter that has been changed among the default ones is the "class_weight" parameter, that has been set with the value "balanced_subsample", to balance the weight of the two classes depending on the number of samples of each one.

Moreover, Table 8.1 contains the accuracy, balanced accuracy and F-score obtained by the best random forests for each attack, together with the set of hyperparameters chosen with the validation phase, as explained in Section 6.1.3. The

[1] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

results can be considered really good, even compared with the ones that can be found in literature and that have been described in Chapter 9. The "Web" attack is the one with the lowest performance, but as explained in Section 7.2, it contained very few positive samples.

Attack	**Accuracy %**	**Balanced accuracy %**	**F-Score %**	**Estimators**	**Max depth**
Botnet	100	100	100	20	16
Bruteforce	99.99	99.99	99.99	20	16
DoS	99.99	99.99	99.99	100	16
DDoS	99.99	99.99	99.99	20	16
Web	99.99	96.59	96.47	20	16

Table 8.1: The evaluation metrics for the best RF models.

Additional considerations can be made by looking at the hyper-parameters: to obtain the best performance among the different models, the least amount of estimators available has been enough, for exception of a single case, the DoS attack. The max depth has also been limited to the least amount possible by the validation phase: this means that the RF does not need all the available features to perform well.
Additionally, since the random forest allows to extract the features that had more weight during the training phase, these have been analysed and the top-5 for each attack are shown in Table 8.2. The number of the features is the one presented in Section 6.1.2. It is possible to notice that some features numbers are recurrent for many attacks: for example, features number 5 (number of segments with payload sent by client), 16 (number of bytes sent by server in the payload) or 17 (number of segments with payload sent by server) are more frequent than others, even if in different order of importance among the attacks. Moreover, it is possible to notice that the other features in the top-5 list are mostly different between the various attacks: this means that the RF is probably capable of finding specific characteristics for each attack. Furthermore, it is possible to notice that almost all the attacks include some time-related features in their top-5 list: features 24 (flow duration), 26 (time of server first payload sent since start), 29 (time of client first ACK without SYN) and 30 (time of server first ACK without SIN).

Attack	**#1**	**#2**	**#3**	**#4**	**#5**
Botnet	17	24	29	16	4
Bruteforce	5	30	0	18	17
DoS	16	18	17	5	12
DDoS	29	16	0	5	26
Web	29	30	5	6	12

Table 8.2: The top-five most important features for each attack type.

8.1.2 Support Vector Machine

The linear SVM classifier[2] has been trained with a single hyper-parameter, that is C, the regularization parameter. The set of used values is composed by 0.5, 1, 1.5 and 3. As explained in Section 5.2.3, this parameter is responsible of the amount of error tolerated by the SVM in the soft-margin problem. The only parameter that has been changed among the default ones is the "class_weight" parameter, that has been set with the value "balanced", to balance the weight of the two classes depending on the number of samples of each one.

Instead, Table 8.3 contains the accuracy, balanced accuracy and F-score obtained by the best support vector machines for each attack, together with the value of the relative hyper-parameter, C. The scores are not really high, specially if compared with the random forest ones. The balanced accuracies are acceptable, but the F-scores are definitely not. The only possible explanation for this result is a high number of false positive classifications since the balanced accuracy is high and knowing that the dataset is highly unbalanced towards the negative class: even a small percentage of negative samples classified as positive has a high impact on the F-score.

Attack	**Accuracy %**	**Balanced accuracy %**	**F-Score %**	**Regularisation parameter (C)**
Botnet	82.45	90.86	31.42	3
Bruteforce	76.03	87.52	24.61	0.5
DoS	83.79	88.35	77.78	3
DDoS	81.48	88.43	68.30	3
Web	49.91	74.95	0.02	3

Table 8.3: The evaluation metrics for the best SVM models.

In this case, the most chosen value for the C hyper-parameter has been 3, while only one model chose the value of 0.5. Since a higher value of C typically means that the SVM allows for less misclassification and uses a smaller margin for its separating hyper-plane, this result had to be expected. However, a smaller margin leads to more error if eventual new samples come from a different distribution.

8.1.3 Neural Network

The neural network classifier[3] has been trained with different combinations of hyper-parameters, changing the value of the batch size, the number and size of the hidden layers and the learning rate. The different sizes of the batches have been 128, 256 and 512, while the learning rate has been set to 0.01, 0.005 and 0.001; instead,

[2] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

the architecture of the NN has been set with one hidden layer, with 16 or 24 nodes and with two hidden layers, with 20 nodes in the first and 10 in the second. The only parameter that has been changed among the default ones is the "solver" parameter, that has been set with the "adam" solver, considered the best choice for big datasets.

Instead, Table 8.4 contains the accuracy, balanced accuracy and F-score obtained by the best neural networks for each attack. Also in this case the results are very good. The scores are overall very high, for exception of the "Web" attack that performs in a worse way, like with all the other algorithms. On the other hand, the hyper-parameters distributions are quite various. There is not a clear preference of the models for a set of hyper-parameters. Each attack dataset used a different combination of hyper-parameters: the only one that has never been chosen is the single hidden layer architecture with 24 nodes.

Attack	**Accuracy %**	**Balanced accuracy %**	**F-Score %**	**Batch size**	**Hidden layers**	**Learning rate**
Botnet	99.99	99.99	99.98	256	(16)	0.005
Bruteforce	99.99	99.99	99.99	128	(16)	0.01
DoS	99.99	99.99	99.99	256	(20, 10)	0.001
DDoS	99.99	99.98	99.98	512	(20, 10)	0.005
Web	99.99	96.21	96.06	512	(20, 10)	0.001

Table 8.4: The evaluation metrics for the best NN models.

8.1.4 Comparison between algorithms

This section contains a comparison between the results obtained by each algorithm for each attack. The measure chosen to perform this comparison is the F-score since it gives more weight to the correct classification of malicious traffic, as explained in Section 5.1.5. Figure 8.1 shows in a single graphic the results already presented in the previous sections, focusing on the F-score. As it is possible to see, both the RF and NN algorithms outperform the SVM one. Moreover, the RF and NN algorithms obtained very similar results across all the attack types.

8.2 Classification time

This section focuses on the results obtained by the best classifiers, concerning the time dimension. The average time needed to perform the prediction of a single sample by the best classifiers selected after the validation is presented in Table 8.5. In this case, the algorithm that outperforms the others in clearly the SVM, with an average time needed to classify a sample of 0.05 μs. The RF has consistent times across all the attacks, for exception of the DoS. The reason for this difference can be found by looking at Table 8.1 that has been previously described: the DoS attack

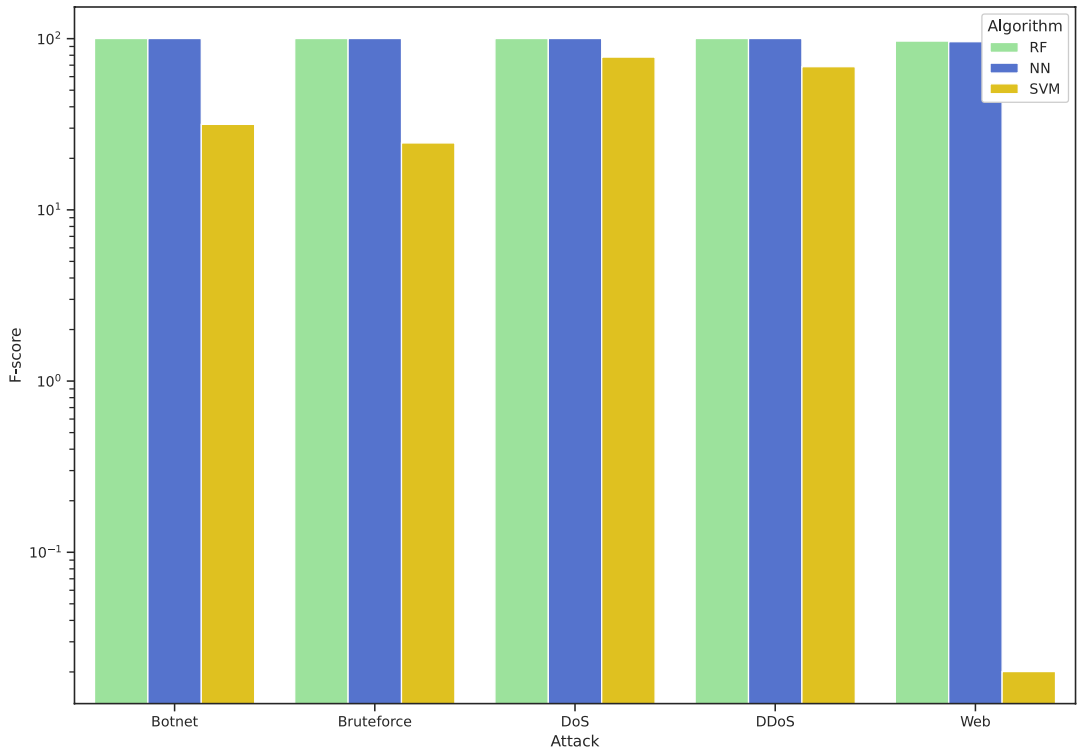

Figure 8.1: The F-score obtained by each algorithm on each attack type.

is the only one with 100 estimators, while the other attacks uses 20 estimators, five times less. In fact, also the classification times are five times smaller with respect to the DoS one. On the other hand, the NN does not show a clear correlation between the classification times, but they are mostly higher with respect to the RF ones.

Attack	**Random Forest**	**SVM**	**Neural Network**
Botnet	0.4 μs	0.05 μs	8.9 μs
Bruteforce	0.4 μs	0.04 μs	17.8 μs
DoS	2.4 μs	0.05 μs	3.8 μs
DDoS	0.5 μs	0.04 μs	14.7 μs
Web	0.5 μs	0.05 μs	1.2 μs

Table 8.5: The times used by each algorithm for each attack type.

8.3 Suricata integration

This section presents the measurements of the overhead added to the Suricata tool with the usage of one or more of the provided classifiers. Moreover, it also shows the difference between the detection ability of Suricata with and without the new classifiers.

Even if the Suricata integration involved many aspects of the original Suricata tool, only the detection phase has been considered for the analysis of the overhead. The reason for this choice is the fact that the initialisation time is not important for the performance of an IDS: Suricata itself performs many CPU-expensive operations during the setup of the internal structures and takes some seconds before being completely operative. So, the operations performed by the integrated ML classifiers have been moved as much as possible in the initialisation phase, that is performed only once. In this way, the detection engine has almost everything it needs ready to be used. The times considered for the comparison of the performances of Suricata with or without the integrated ML models have been computed using the rule shown in Listing 8.2, adding or removing ML keywords for different tests and averaging the results over 20 tries for each test. This base rule is the same one used in Appendix A, where its precise meaning can be found.

```
alert tcp 192.168.178.20 any -> any any (msg:"My custom rule
   with machine learning!"; flow:established, <ML options>;
   content:"facebook"; nocase; classtype:policy-violation;
   sid:666; rev:1;)
```

Figure 8.2: The base rule used to perform tests.

The base rule has been expanded using different combinations of classifiers. Since the overall best-performing models have been random forests for all the attacks, both in terms of classification score and time performance, these classifiers have been used for all the attack types for this specific test. Table 8.6 presents the average time needed by the Suricata engine to complete the execution of the detection phase of a single packet, including the normal Suricata workflow. In this way it is possible to analyse how much does the ML engine impact on the Suricata detection performance.

Number of ML models used	**ML detection time**	**Total detection time**
0	0 μs	5 μs
1	18232 μs	18257 μs
2	27735 μs	27766 μs
3	36070 μs	36093 μs
4	56186 μs	56208 μs
5	59932 μs	59960 μs

Table 8.6: The times used by each algorithm for each attack type.

The overhead added by the presence of the ML detection engine is noticeable. A single ML model inside the rule takes 18ms on average to classify the flow. However, increasing the number of models inside the same rule does not increase linearly the amount of time needed. Furthermore, the difference between the total

detection time spent by Suricata and the time used only by the ML engine is almost constant and equal to 20-30 μs. Finally, a consideration about the obtained time measures must be made: almost all the time needed by the ML engine is used by a single function: the `predict()` function contained in the Python script provided. However, this same time had been previously computed for the classification time of the classifiers, as already shown in Table 8.5, and in that case the times are considerably lower, around 0.5 μs. A possible reason for this high discrepancy between the times taken by the same function can be found directly in the Scikit-learn documentation[4], which shows that the prediction time for a single sample is higher in proportion than the average prediction time needed for a batch of samples.

[4] https://scikit-learn.org/stable/auto_examples/applications/plot_prediction_latency.html

Chapter 9

Related works

This chapter presents an overview of the different approaches found in literature about both the main topics encountered in this work: the detection of anomalies in network traffic with Machine Learning algorithms and the integration of such techniques within an IDS or IPS. Beside the description of the different works available, a direct comparison between this work and the literature ones is performed, highlighting differences and common choices.
Section 9.1 contains works that are more similar to the general approach of this work, i.e. they use some specific dataset to generate models that are then integrated in some IDS tool. Instead, Section 9.2 works are slightly different and cover a broader range of approaches that focus mainly on the Machine Learning network traffic analysis aspect, without considering the presence of an IDS. Finally, Table 9.1 presents the characteristics of the datasets that will be described in the following sections, focusing on the size and the type of network attacks contained.

9.1 Machine Learning and IDS

The works present in literature about the integration of a Machine Learning pipeline inside an IDS are not many and most of them focus on different aspects of the problem. For this reason, a direct comparison may sometimes be difficult.
For example, a 2020 work [59] implemented an IDS based on Docker containers[1], where each one of the three proposed containers works independently. The collection of network traffic is performed in batches, storing the last three hours of network activity and then performing the predictions. This IDS has been trained using HTTP traffic, gathered from different datasets: the CSIC-2010[2] and the CIC-IDS2017[3], which is a direct predecessor of the dataset described in Section 7.1. Another major difference is the choice of the training algorithms: three different types of Neural Networks have been chosen, fully-connected NNs, recurrent NNs and convolutional NNs, with an average F-score of 80%. Finally, the implemented

[1] https://www.docker.com/

[2] https://www.isi.csic.es/dataset/

[3] https://www.unb.ca/cic/datasets/ids-2017.html

Dataset	Size	Network attacks	Related works
KDDCup 99	5 millions	DoS, probe, privilege escalation	[49, 50, 51, 52, 53, 54]
NSL-KDD	1 million	DoS, probe, privilege escalation	[50, 51, 55, 56, 57, 58]
CSIC-2010	60.000	Web attack	[59]
ADFA-LD	5000	Bruteforce, privilege escalation	[55]
UNSWNB15	2.5 millions	DoS, port scan, backdoor, probe, worms	[60]
CIDD-001	31 millions	DoS, probe, bruteforce	[57]
CIC-IDS2017	3 millions	DoS, DDoS, bruteforce, Web attack, botnet	[57, 59, 61]
CSE-CIC-IDS2018	28 millions	DoS, DDoS, bruteforce, Web attack, botnet	This work

Table 9.1: The datasets used by the related works.

IDS, that has been called AI-IDS, has been designed to work in parallel with a standard IDS like Snort, rather than being strictly integrated inside it. The authors suggest to use it as an "assistant system", capable of detecting anomalies that may bypass Snort rules and they propose to use the generated logs to manually improve existing Snort rules.

Another work from 2020 [55] proposes a novel IDS framework, called AlarmNet-IDS, that works both as a HIDS and as a NIDS, which theoretical explanation has been presented in Section 4.3. This IDS has been deployed on a Linux machine that collects and parses network traffic and system call traces to perform predictions. In this case, two distinct dataset have been used for the training of the classifiers: one for the NIDS and ore for the HIDS. The NIDS dataset is the NSL-KDD[4], a popular dataset created starting from an old standard dataset from 1999, the KDDCup 99[5]. The NSL-KDD has been created to solve some of the problems of the KDDCup 99, like the presence of redundant records [62]: in fact, from the almost 5 millions of training samples and 300.000 test samples of the KDDCup 99 dataset, only an average of 25% of the records is actually unique. This reduction leads to a total size of the training set of around 1 million of samples: 80% of them represents legitimate traffic, while the remaining 20% is split among the various attacks contained in the dataset. On the other hand, the dataset used for the HIDS part is the ADFA-LD[6],

[4] https://www.unb.ca/cic/datasets/nsl.html

[5] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[6] https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/

which has been designed specifically for Linux threats and contains system call traces of various Linux servers. There is another version of this dataset, designed for Windows, the ADFA-WD.
Furthermore, the high-level architecture of the AlarmNet-IDS is shown in Figure 9.1 and follows the general scheme described in Section 4.1.2. The network packets are captured using tcpdump and then parsed with Bro (today known as Zeek[7]), while system call traces are captured with "auditd", a Linux audit tool. The feature extraction is performed with an unsupervised Neural Network called autoencoder, which aim is to reconstruct the input data passing through a latent space. This less-dimensional space is then used as input of a fully-connected NN, composed by a single layer followed by the softmax layer to obtain predictions. Instead, for the HIDS data, a CNN composed by a single convolutional layer has been used to classify the input. The average F-score for NIDS data is 78% and 75% for the HIDS one. Also in this case, the resulting IDS is designed as a stand-alone tool, than can operate by itself and it is not connected to any existing IDS software.

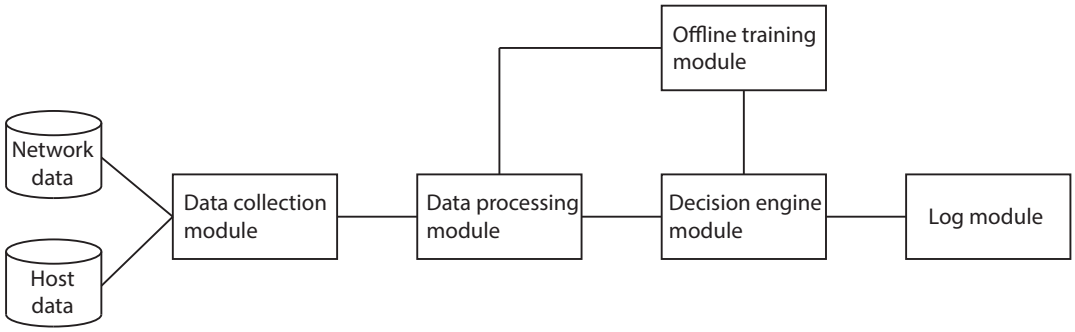

Figure 9.1: The high level architecture of the AlarmNetIDS.

The next work presented has the objective of creating an almost real-time IDS using Machine Learning techniques [49]. The authors designed an IDS workflow starting from the capture of the packets and ending with the production of the logs. Differently from previous works, this real-time IDS focuses on three different network protocols: TCP, UDP and ICMP. In this case, the dataset used to train the classifiers has been created by the authors themselves, performing 4 different types of DoS attacks and 13 probe attacks performed with an nmap tool on Windows. The normal traffic has been captured in the local department of a University. The KDDCup 99 dataset has been also used for the test phase.
Then, the traffic is pre-processed using the Information Gain technique, which has been explained in Section 5.2. Moreover, the algorithms used to perform the classification are: Decision Trees; a rule-based algorithm called Ripper Rule; a fully-connected NN and Naïve Bayes classifier. The Ripper Rule algorithm consists of two stages: the first stage is to initialize the rule conditions, while the next stage uses a rule optimization technique. This step re-prunes each rule of the rule set to minimize the errors. Instead, the Naïve Bayes classifier is a simple technique

[7] https://zeek.org/

for classification using a probabilistic model from Bayes's theorem with the assumptions of independent attributes. The prediction phase happens by collecting the captured data in batches, and passing each batch to the chosen classification techniques. The detection result is decided with a majority voting by the various algorithms and it is stored in a log file. The F-score is not available for a direct comparison, but the average True Positive rate is around 99%. Furthermore, some resources usage measures are also available in this case: the developed system uses around the 25% of the CPU (a 2.83GHz Intel Pentium Quad core) and 100MB of memory, with a network rate of 100Mbps.

The following is a work with a slightly different approach: it focuses on a complex IDS to monitor 5G networks [50]. The proposed architecture is split in three layers: a forwarding layer, a management and control layer and a data and intelligence layer. The forwarding layer is responsible for traffic monitoring and capturing. It can collect and upload network flows to the control layer, and block malicious flows according to the instructions of the controller. Management and control layer identifies suspicious flows and detects anomalies preliminarily using uploaded flow information. It also generates protection strategies according to decisions made by the intelligent layer and instructs the forwarding layer. The data and intelligence layer makes further analysis through feature selection and flow classification using adaptive machine learning algorithms.
The training datasets used are a 10% sub-set of the KDDCup 99 and its improved version, the NSL-KDD. The selected algorithms are: Random Forest; K-means (an unsupervised clustering technique) and Adaptive Boosting, an ensemble classifier composed of different classifiers, in a way similar to Random Forest. The average F-score is 90%. Differently from other works, in this one the authors only provide a proposal of IDS design, without actually providing any implementation information.

However, the approach that is most similar to the proposed one can be found in a 2018 work [51]. In this case the authors performed a preliminary study on two popular IDSes: Snort and Suricata (both presented in Section 4.4). Then, they attempted to improve the accuracy of Snort with Machine Learning techniques, by developing an ad-hoc plug-in for Snort. The chosen datasets are the DARPA Intrusion Detection Evaluation[8] dataset and the NSL-KDD one, while the chosen algorithms are Support Vector Machines, Decision Tree, Fuzzy Logic and Naïve Bayes classifier. The SVM has been selected as the best performing algorithm and used in the subsequent integration.
The resulting architecture of Snort after the deployment of the plug-in is shown in Figure 9.2, where the original architecture is highlighted and separated from the plug-in. As explained by the authors, the plug-in operates in parallel with the Snort detection engine. The pre-processor sends decoded packets to the plug-in, which uses the Machine Learning algorithm to classify the legitimate and malicious traffic.
Also in this case the F-score is not available, so a direct comparison is not possible. However, the average False Positive rate obtained by SVM on the used dataset is

[8] https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset

around 2% and increases to 8.6% after the integration with Snort. In this case the authors provide some performances measures: for example the CPU usage with a 100Mbps network traffic goes from 30% without the plug-in to 35% with the plug-in and the respective memory usage goes from 1.5GB to 1.6GB. Instead, for a 10Gbps traffic, the CPU usage increases from 65% to 73% and the memory from 3GB to 3.7GB.

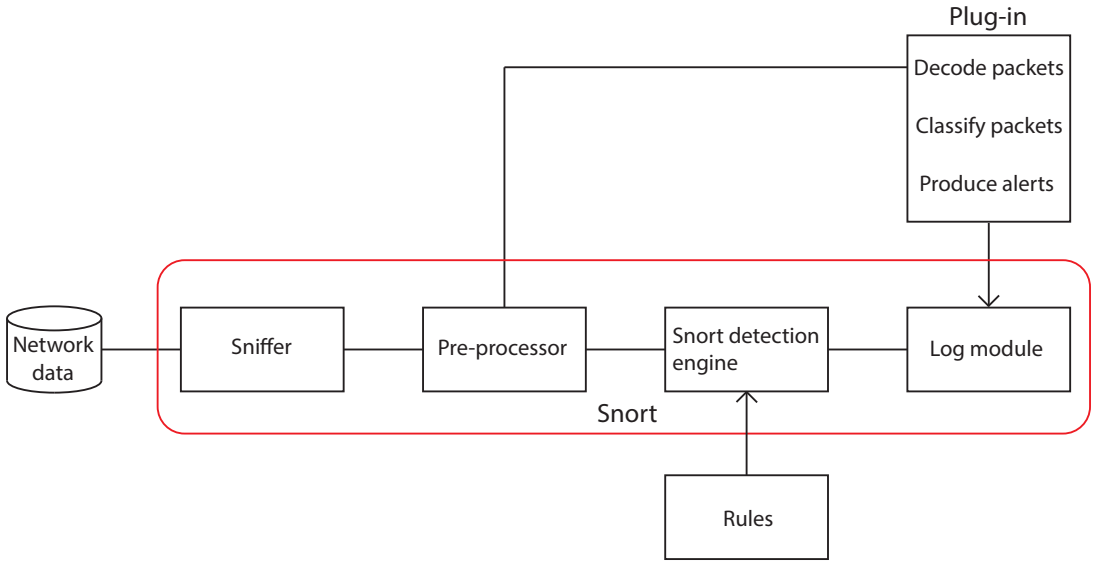

Figure 9.2: The resulting Snort architecture after the deployment of the detection plug-in.

Finally, a brief overview of the differences and similarities between the proposed work and the ones described in this section can be summed up with these points: only one of the works found in literature proposes and develops an integration with an already existing tool [51], while the others propose a stand-alone IDS solution [59, 55, 49, 50]; the datasets used are in most cases NSL-KDD or KDDCup 99 or other old datasets, the only exceptions being CIC-IDS2017 [59] or the ADFA-LD dataset specific for HIDS [55]. The analysed attacks are a direct consequence of the chosen dataset, with exception of a work where the dataset has been created ad-hoc with manually performed attacks [49]. The set of chosen algorithms contains some different types of NNs, Decision Trees, Random Forest, Naïve Bayes classifier, SVM, K-means clustering and Fuzzy Logic and Ripper Rule algorithms. The detailed reasons for this work's design choices can be found in Chapter 6, while the ones concerning the dataset have been presented in Chapter 7.

9.2 Machine Learning and traffic analysis

Differently from the works presented in Section 9.1, the following ones approach the problem of anomaly detection in network traffic mainly under the aspect of the classification of the traffic, without considering the implementation of an actual IDS solution or the integration inside an existing tool. For this reason, the comparison with the proposed work will concern mostly the Machine Learning choices.

Even if many works describe their results as a "Machine Learning based IDS", many of them only focus on the classification part of the problem. For example, a 2019 work [60] uses different feature selection techniques to optimise the training and then performs the classification of the collected traffic. In this case, the UNSW-NB15[9] dataset has been used, which contains around 100GB of packets captured with tcpdump. The authors propose a feature selection step composed by an outlier detection algorithm and a Genetic algorithm step to choose the best sub-set of features to perform the training. This step is then performed using a Random Forest. The average F-score obtained is 62%.

A completely different approach is adopted by the authors of a work focused on detecting anomalies in mobile devices traffic [63]. In this case the authors decided to create themselves a dataset with mobile network traffic, developing a tool to collect iPhone data on a voluntary base (i.e. each user had to willingly install a client on the iPhone). The data collected during these experiments contain telephone call information, SMS and web browsing history. More specifically, the dataset consists in 35 iPhone users, around 8300 phone calls, 11300 SMSs and 800 hyperlinks. The chosen algorithms are an RBF Neural Network (a NN with a specific activation function), the K-Nearest Neighbours algorithm, Random Forest and a Bayesian Network. The best performing algorithms are the Random Forest and K-NN with an average True Positive rate of 99.9%. However, this work was focused on the creation of legitimate user behavioural profiles, rather than in the detection of actual anomalies.

Instead, other works focus on the comparison of different classification algorithms using consolidated datasets. In this case, the authors used the NSL-KDD dataset to compare the results obtained by a recurrent NN with the benchmark results of more common algorithms, such as SVM, K-NN and Random Forest [56]. Different RNNs have been tried for this work, performing the validation step on the number of hidden nodes and the learning rate. The authors show that their best RNN reaches an accuracy of 83%, with respect of the average of 80% obtained in literature by other algorithms. However, the authors did not mention the True Positive rate, which is in this case more significant than the accuracy and in this case equals only 73%.

Another work that created its own dataset to perform its experiments concerns the analysis of network anomalies in a Supervisory Control And Data Acquisition (SCADA) system inside an industrial control system [64]. Due to the privacy issues of industrial network traffic, the authors had to develop an real-world testbed that resembles an actual industrial plant: more specifically, a system that supervises the water level an turbidity of a water storage tank. The attacks performed to create the dataset include backdoors, command injection and SQL injection. Instead, the used algorithms include SVM, K-NN, Naïve Bayes classifier, Random Forest, Decision Tree, logistic regression and Neural Networks. The library used for all of them is scikit-learn, for exception of the NN, which has been created with Keras library. In this case, the measure used to compare the results is the Matthews

[9] https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/

Correlation Coefficient (MCC), a popular measure that shows the correlation between the observed and predicted values. The three best performing algorithms are Random Forest, Decision Tree and K-NN with a MCC of 96.81%, 94.26% and 93.44% respectively.

A 2019 work [57] focuses on the optimization of a single Machine Learning algorithm over multiple datasets. The chosen algorithm is a Neural Network, with a varying number of hidden layers. The peculiar characteristic of this work is the usage of a Genetic algorithm to choose the optimal number of hidden layers of the NN. This step works as a sort of validation step, that searches in an iterative way the best parameters for the NN. Once this parameters have been found, the training works as usual. Three different datasets have been used for this work: the CIC-IDS2017, the NSL-KDD and the CIDD-001[10] dataset. The F-score obtained separately on all three datasets is equal to 99%.

A possible different approach consists in the usage of a Big Data environment to develop the classification algorithms. For example, the Apache Spark engine[11] has been used to create a workflow entirely based on Big Data tools [52]. In this case the KDDCup 99 dataset has been loaded in a Spark Resilient Distributed Dataset (RDD) and preprocessed. Then, the Spark Chi-SVM algorithm has been used to train and evaluate the model. The difference between this method and a standard SVM is the presence of a regularization term inside the SVM equation. The results provided show an Area Under Precision-Recall Curve (AUPR) of 96.24% for the Chi-SVM, with respect of the 94.36% of the standard SVM and 92.77% of a logistic regression algorithm. Moreover, the authors give the total prediction time for the three compared algorithms, that are equal to 1.21s, 1.37s and 1.58s respectively.

Some works are focalised on a single dataset and experiment different algorithms to compare their performances. The CIC-IDS2017 dataset has been used to test some Machine Learning algorithms using libraries available in the R language [61]. After a first pre-processing of the dataset, the authors performed a long feature selection step with an unspecified algorithm, to reduce the dimensionality of the dataset. From the original 80 features provided in the dataset CSV files, the top 15 ones have been picked to proceed with the training phase. The two algorithms used have been a NN with 500 iterations and Random Forest with 10 epochs. The obtained True Positive rate average is equal to 96% for the NN and 98% for the Random Forest.

Another work used a single dataset to compare a different set of Machine Learning algorithms [53]. In this case the chosen dataset is the KDDCup 99 and the set of algorithms is composed by: a Decision Tree, a NN with a single hidden layer and a Naïve Bayes classifier. The KDDCup 99 records had to be pre-processed to remove the duplicate records; then the different algorithms have been trained and tested over the dataset. The average True Positive rate for the three classifiers is equal to 92%.
The same dataset has been used similarly by another work [54]. The authors followed the same approach, but with a different set of algorithms. The chosen ones

[10] http://groups.di.unipi.it/~hkholidy/projects/cidd/

[11] https://spark.apache.org/

have been SVM and Random Forest, for which has been provided the test accuracy: the SVM obtained a 92% of accuracy and the Random Forest a 91.4%. However, the confusion matrix shows a 92.5% recall for the DoS attack and only an 84.5% for less frequent attacks like probing, for the Random Forest. Instead, the SVM recall is 96% for DoS, bu only 57% for probing attacks.

The last work presented used an unsupervised clustering technique to separate the legitimate traffic from the malicious one [58]. The NSL-KDD dataset has been used and the selected algorithm is the K-means clustering technique. The authors tried different numbers of clusters to compare different results: 11, 22, 44, 66 and 88 clusters have been created to separate the 22 classes contained in the dataset. The results are not very good, with a maximum detection rate of 28.8% for the positive class and 22 clusters and a 52.8% for the negative class. Increasing the number of clusters further reduces the obtained results.

Finally, the differences and similarities of the described works with the experiments proposed in this work concern the following choices: the datasets used include mostly outdated ones (KDDCup 99 or NSL-KDD), but the CIC-IDS2017 has been used by some works with good results [57, 61]; the most adopted algorithms are Random Forest, Neural Networks and SVM, with some usage of the Naïve Bayes classifier [64, 53] and a single case of an unsupervised clustering technique [58]. Also, some different frameworks and languages have been used beside Python, like Scala for the Spark environment [52] and R [61].

Chapter 10

Conclusions

The objective of this work has been the integration of machine learning classifiers of network traffic attacks inside an IDS, Suricata. The statistics of the analysed network traffic have been extracted with Tstat, an automated tool for passive network monitoring. As explained in Chapter 2, Tstat has been chosen because it can produce a good amount of TCP flows statistics and it provides some API functions to interact with it from other external software.
Then, the statistics of the resulting dataset, a composition of the CSE-CIC-IDS2018 and the Torsec datasets, have been analysed, as explained in Chapter 7. As a result, some of the Tstat core TCP features have been removed from the dataset because they have not been considered useful for the experiments.

Once the preparation of the dataset has been completed, the machine learning workflow to create and select the best models has been designed. The three different classification algorithms, i.e. random forests, support vector machines and neural networks, have been used to train different classifiers for each attack, using various combinations of hyper-parameters, as explained in Chapter 6.
Therefore, a new set of classifiers has been trained using the best set of hyper-parameters, producing a single model for each algorithm and for each attack type, for a total of 15 models. The number of models has been further reduced by choosing, for a single attack type, only the best performing algorithm: as a result each one of the five attack types has its own ML classifier. As shown in Chapter 8, the random forest algorithm has been chosen for all the network attacks analysed, due to an F-score higher than the other two algorithms and a reasonably low average classification time: around 0.5 μs per flow.

Finally, the obtained models have been integrated into the Suricata IDS. To perform this action, the Suricata workflow has been modified in some specific points, as detailed in Chapter 6. A new set of options has been created inside the Suricata rules syntax, that allows to create signatures able to detect specific attacks using the relative ML models. In this way it is possible to combine ML-based detection together with the normal Suricata rules options. In order to obtain this result, the modules that have been modified are the decoding and detection ones, beside the Suricata initialisation phase. Moreover, the Tstat API has been expanded with a new function, to obtain the statistics of a single specific flow.

Instead, Chapter 9 contains a comparison of other existing similar solutions, by

describing them and highlighting differences and similarities.
Although the experimental results shown in Chapter 8 are encouraging, with an average F-score higher than 99% and a classification time of 0.5 μs per flow, the proposed solution has some limitations that could be improved in a future work: first, the set of used networks attacks, even if wider than the ones typically used in literature, should be further expanded with other types of threats, to cover the highest number of cyber-attacks possible; then, the extracted statistics should include protocols different from TCP, maybe using an extraction tool different from Tstat or by extending its functionalities. Moreover, some technical improvements could be developed for the integration of the models inside Suricata, by reducing the overhead caused by the ML classification. An example of a possible solution for this problem is to try different ML frameworks, more optimised to perform predictions of single samples, rather than in batches.

Appendix A

User manual

This appendix contains the manual for any user that would like to use the modified version of the Suricata IDPS, which includes the ability of detecting malicious TCP flows using pre-made machine learning classifiers. The details to create additional custom ML models can be found in Appendix B. Instead, this manual describes the technical steps to follow, starting from the installation of the required tools and the setup of the environment.

A.1 Requirements

The software needed by the modified version of Suricata is composed by the Tstat tool, Suricata itself, the source code of the integrated functionalities and the machine learning models. All the described steps have been tested on Ubuntu 20.04.2 LTS and for this reason the example commands given below come from that OS; however, the set of operating systems compatible with the final tool include all the Linux and Mac systems. Windows is not supported because the Tstat tool is not available for that OS.

The latest version of Tstat[1] is available through its SVN repository[2] and can be downloaded with the command
`svn checkout http://tstat.polito.it/svn/software/tstat/trunk tstat`
while a specific version can be downloaded with the command
`wget http://tstat.polito.it/download/tstat-3.x.y.tar.gz`
where `3.x.y` represents the specific version. The latest version available during the execution of this work's experiments is the `3.1.1`, hence the compatibility with later versions is not guaranteed. Moreover, Tstat requires some specific libraries to work, like `libpcap`: in case they are missing from the user's system, the Tstat installation procedure will automatically show the list of needed libraries. The installation guide of the Tstat[3] tool can be found on its website and it is not explained

[1] http://tstat.polito.it/

[2] http://tstat.polito.it/viewvc/software/tstat/trunk/

[3] http://tstat.polito.it/HOWTO.shtml

in this manual. However, it has to be noted that during the installation of the Tstat tool, the option to enable the library version provided by Tstat (`libtstat`) has to be used, with the command
`./configure --enable-libtstat`

Suricata[4] can be downloaded and installed in different ways. For example, on Ubuntu it can be downloaded with the provided Personal Package Archive[5] (PPA) by executing the command
`sudo add-apt-repository ppa:oisf/suricata-stable`
However, the suggested way to download Suricata is by downloading the source code, available on the Suricata website[6]. The latest Suricata version available during the development of the integrated tool is the `6.0.1` and it is the suggested version for any user that is not interested in manually managing the source code, as explained in Appendix B. The complete list of available versions can be found on a dedicated page on the Suricata website[7].
Moreover, Suricata requires the installation of many dependencies on the user's system. The recommended list and the related command is shown in Listing A.1, while the guide to successfully install and configure Suricata can be found on its website[8].

```
apt-get install libpcre3 libpcre3-dbg libpcre3-dev \
       build-essential libpcap-dev libnet1-dev libyaml-0-2 \
       libyaml-dev pkg-config zlib1g zlib1g-dev libcap-ng-dev \
       libcap-ng0 make libmagic-dev libgeoip-dev liblua5.1-dev \
       libhiredis-dev libevent-dev python-yaml rustc cargo
```

Figure A.1: The command to obtain the recommended Suricata dependencies.

Instead, the source code needed to install the modified version of Suricata, together with a set of pre-made machine learning classifiers, can be found in the repository associated with this work. The only additional dependencies needed to use this code are the Python libraries used by the classification script and can be obtained using the commands `pip install joblib` and `pip install scikit-learn` provided the fact that the Python 3 version is available on the user's system.

[4] https://suricata.readthedocs.io/en/latest/quickstart.html

[5] https://help.launchpad.net/Packaging/PPA

[6] https://suricata-ids.org/download/

[7] https://www.openinfosecfoundation.org/downloads/

[8] https://suricata.readthedocs.io/en/latest/install.html

A.2 Installation

Once all the previous steps have been performed, the user should be in a situation where both Tstat and Suricata have been installed separately and are able to run independently. The process to obtain the integrated tool used in this work is composed by a few more steps.
First, the content of the "tstat modified" folder available inside this work's repository must be copied into the Tstat source folder. More in detail, the files `inireader.c options.c tcp.c tstat.c tstat.h` must be copied into the "tstat" folder contained in the Tstat source code directory. Instead, the `libtstat.h` file contained in the provided "include" folder must be copied into the "include" folder contained in the Tstat source code directory. Now, the Tstat tool must be installed again in the system, simply using the `make` and `make install` commands used to install it the first time. Then, a similar process must be performed for the Suricata source code folder. In this case, all the files contained inside the provided "suricata modified" folder must be copied into the "src" folder contained in the Suricata source code directory. Now, the Suricata installation pipeline must be informed of the existence of the newly installed `libtstat` library, by adding the command
`AC_CHECK_LIB([tstat], [tstat_next_pckt],, AC_MSG_ERROR([missing 'tstat' library]))`
inside the "configure.ac" file contained in the Suricata source code directory. Moreover, Suricata must be informed also about the location of the Python libraries: this is done by adding a similar command to the same "configure.ac" file, which is
`AC_CHECK_LIB([pyhton3.8], [PyArg_ParseTuple],, AC_MSG_ERROR([missing 'python3.8' library]))`
where `python3.8` is the version of Python used for the integration and for this reason is the suggested one. At this point, the Suricata tool can be installed again using the `make` and `make install` commands used to install it the first time. Finally, the tool is ready to be used: the last missing steps are the configuration and the usage of the tool.

A.3 Usage

This section contains the steps needed to correctly configure the installed tool, followed by a complete example of usage and creation of new signatures.

A.3.1 Configuration

If the tool has been installed correctly, before trying to use it it is necessary to setup some configurations. The file "integration.conf" contained in the "config" folder provided in the repository is the main configuration file: Suricata searches for this file in the current working directory at running time. The structure of this file is explained in its first line and an example is shown in Listing A.2. The remaining three lines of the file contain three absolute paths and are used by Suricata to determine respectively the path to the Tstat configuration file ("tstat.conf" in this case), the path of the folder containing the Python script for classification

(with the fixed name of "ml_detect.py") and the path to the script configuration file ("ml_detect.conf" in this case). In this way, it is possible to place these files in the preferred directory: the only constraints are the presence of the "integration.conf" in the current working directory and the existence of a "ml_detect.py" in the directory specified in the configuration.

```
# Format: line 0 = this line, line 1 = path to tstat.conf, line
   2 = path to directory containing ml_detect.py, line 3 = path
   to ml_detect.conf
/home/matteo/suricata-6.0.1/TSTAT/tstat.conf
/home/matteo/suricata-6.0.1/TSTAT/
/home/matteo/suricata-6.0.1/TSTAT/ml_detect.conf
```

Figure A.2: An example of the content of the integration.conf file.

The exact contents of the "tstat.conf" file are explained in the Tstat website[9]: it contains some configuration information needed by Tstat, together with the paths of other Tstat files. It is fundamental to correctly setup the Tstat configuration to be able to use the new Suricata tool. Instead, the "ml_detect.conf" file contains the absolute paths to each of the classifiers available and the related scalers, in the form of a dictionary: the key for each line is the value that must be used inside the Suricata signatures to perform the classification of flows. Finally, the user has to configure Suricata before running the tool. The complete guide to customise Suricata can be found on its website[10]: the "suricata.yaml" configuration file must be edited with the user's information (e.g. the name of the network interface to monitor) and the path to any additional file of signatures must be added.

Once all this steps have been completed, the user can run Suricata from the command line with the command
`sudo suricata -c <suricata.yaml path> -i <interface name> -l <suricata log dir>`
that starts Suricata. If the operation is successful, the user should see a message similar to the one provided in Figure A.3.

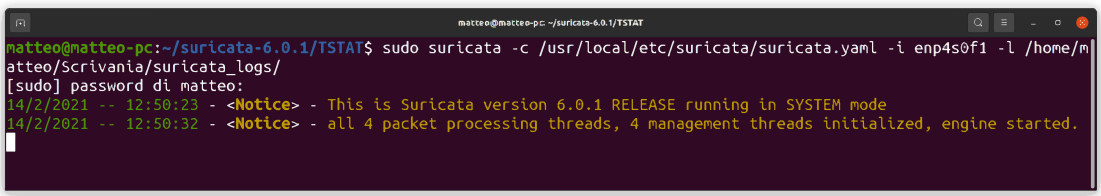

Figure A.3: The output of a successful start of Suricata.

[9] http://tstat.polito.it/HOWTO.shtml

[10] https://suricata.readthedocs.io/en/latest/quickstart.html#basic-setup

A.3.2 Creation of new signatures

With the new tool installed and ready to be used, the user can classify TCP flows with the chosen machine learning classifiers. To do so, the user can modify any already existing Suricata signature or can create a new one. The following example will concern the creation of a new signature: the complete list of available keywords and rules format can be found on the Suricata website[11].

For this example, the file "custom.rules" provided in the repository has been used, but the user can create any number of rule files, remembering to add their path inside the "suricata.yaml" configuration file. First of all, it is necessary to write a classic Suricata rule, like the one provided in Listing A.4.

```
alert tcp 192.168.178.20 any -> any any (msg:"My custom rule!";
   flow:established; content:"facebook"; nocase;
   classtype:policy-violation; sid:666; rev:1;)
```

Figure A.4: An example of a trivial Suricata rule.

This example rule is very simple: the action performed in case of match is an `alert` and the protocol to look for is `tcp`. The source address has been set to a local host (`192.168.178.20`), while the source port and the destination address and port are set to match to any address and port number. The `flow: established` option is used to match only flows where the TCP handshake has been completed, the `content: facebook` option is used to match packets that contain the word "facebook" inside the payload and the option `nocase` tells the engine that the content value is not case-sensitive. The `sid` value can be freely chosen by the user, paying attention to not use a value already used for other signatures since Suricata would overwrite the previous rule in that case. An example of usage of this very trivial rule is to check if the host `192.168.178.20` visits or searches for the Facebook website, leading to a policy violation (`classtype:policy-violation` in the example).

Then, to add the machine learning classification of the flow to look for specific attacks, the user has to add one or more special keywords inside the option of the rule. The keywords supported by the pre-made set of classifiers available in the repository are:

ml_detect_bot to match with flows classified as botnet attacks;

ml_detect_brute to match with flows classified as bruteforce attacks;

ml_detect_dos to match with flows classified as DoS attacks;

ml_detect_loic to match with flows classified as DDoS attacks;

[11] https://suricata.readthedocs.io/en/latest/rules/index.html

ml_detect_web to match with flows classified as web attacks;

ml_detect_all to match with flows classified as any of the previous attacks.

The description of the attacks used to train these classifiers can be found in Section 7.1, while Chapter 3 contains the technical explanation of these attacks. The user can choose a combination of any of the previous keywords to detect multiple attacks with a single rule: if at least one of the chosen classifiers finds a match, the whole machine learning detection engine considers the flow as matching. Hence, the overall rule is considered as matching by Suricata if both the classic options and the machine learning engine consider the flow as matching. The only exception with the explained syntax is the `ml_detect_all` keyword: if this keyword is used, none of the other machine learning keywords can be inserted in the rule because it already includes all the classifiers automatically. In any case, the Suricata initialiser performs a syntactic and semantic check of all the provided rules and informs the user of any error. Finally, the previous example has been integrated with some of these keywords in Listing A.5, to show a practical example of the usage of the new keywords. In this case, the `ml_detect_bot` and `ml_detect_web` keywords have been used: they must be inserted as values of the option `flow`.

```
alert tcp 192.168.178.20 any -> any any (msg:"My custom rule
   with machine learning!"; flow:established, ml_detect_bot,
   ml_detect_web; content:"facebook"; nocase;
   classtype:policy-violation; sid:666; rev:1;)
```

Figure A.5: An example of Suricata rule with the new keywords.

Appendix B

Developer manual

This appendix is focused on the technical details useful for developers that would like to modify or to expand the provided work. While the installation and usage of the tool can be found in Appendix A, the following sections contain the necessary information to perform actions like the addition of more classifiers, modify the training pipeline, use different datasets, use a different set of features from the same dataset, add other keywords to the Suricata rules or improve the existing work in any other way.

B.1 Requirements

The system requirements to modify the provided tool are the same needed to use it as-is and can be found in Appendix A. The new Suricata tool, Tstat and the files provided in this work's repository can be all necessary, depending on the type of action to perform. If the developer is interested in modifying the training pipeline or add new models, as explained in Section B.3.2, it may be necessary to install additional Python libraries to use the provided scripts. This can be done with the command `pip install matplotlib`.

B.2 Installation

The manual to install the tools provided in this work has been already explained in Appendix A. The additional files useful for a developer are only Python scripts and therefore they do not require any installation. If the developer is interested in using a dataset different from the one used in this work and presented in Chapter 7, the Tstat tool may have to be used as a stand-alone software, instead of as a library. In this case, it is suggested to perform the Tstat configuration as needed, as explained in the Tstat website[1].

[1] http://tstat.polito.it/HOWTO.shtml

B.3 Code usage

This section contains the technical description of the files provided in this work's repository. This description has been split in different sections, depending on the purpose of each file. First, the Python scripts used to parse the "pcap" files of the dataset, extract flow features with Tstat and add the label to each flow are presented. Then, The Python scripts to create the machine learning classifiers starting from the dataset are described. Instead, the following section concerns the C files and the Python script used to expand the functionalities of Suricata with the ML classifiers. Finally, the last section analyses the Tstat C files that have been modified to expand the `libtstat` API to produce the data needed by Suricata at run-time.

B.3.1 Dataset scripts

The two Python scripts used to parse the pcap packets captures, extract the related Tstat statistics and label the flows are contained in the "dataset scripts" directory of the provided repository. Each one of the two scripts comes with its own configuration file, needed to customise the execution of the script itself.

The first script is "read_pcap.py" and it is used to extract the Tstat flow statistics from many pcap files at the same time. In fact, to analyse many pcap files with Tstat it is necessary to pass the name of each file as a command line argument or to call Tstat multiple times, one for each file. This script simply performs this operation and collects the outputs in a single file. The configuration file contains three different absolute paths: the path to the "runtime.conf" configuration file used by Tstat, the path to the input folder containing the pcap files and the path to the output folder, where the statistics will be produced.

Instead, the second script is used to add the label "Malign" or "Benign" to each one of the flows produced by the previous script. The distinctions between legitimate and malicious traffic has been performed using the IP addresses of the attackers provided by the creators of the dataset, as explained in Chapter 6. The produced output is split in a set of files with a maximum length of 500.000 lines. The reason for this choice is to ease the management of the output, in case it is too big to be processed at once by the machine learning pipeline. Also, in this way it is possible to use only a sub-set of the data, in case it is needed.
Furthermore, the configuration file of this script contains more lines with respect to the previous one. The first path contains the name of the input file, while the second one is the output folder path. Instead, the third value is the prefix of the output files produced by the script: for example, if the chosen prefix is `my_dataset`, the files produced will be `my_dataset_01 my_dataset_02` and so on, until all the input file has been labelled. Finally, the last line contains the list of IP address to consider as malign. It is a list of space-separated IPv4 addresses, like `18.219.211.138 18.217.165.70 18.217.151.44`.

B.3.2 Creation and selection of classifiers

The four Python scripts used to create ML classifiers are contained in the "ml scripts" folder of the provided repository, with the relative configuration files. The only script without a configuration file is "ml_lib.py", which is used as a library module for the other scripts. It exposes four different functions and contains some others used for internal computations. The four exposed functions are used to create the sub-sets used for the training, validation and test phases starting from the labelled CSV files, to scale the features of the dataset and to compute the different types of statistics needed by the machine learning pipeline. The details of these steps have been explained in Chapter 6, while the technical description of the used algorithms can be found in Chapter 5.
However, the functions to create the datasets expects a specific directory structure to work correctly: a root directory must contain many sub-directories, each one with a specific dataset. An example of this structure is provided in Figure B.1, where `<root dir>` contains two sub-folders `<sub-dir 01>` and `<sub-dir 02>`, each one with two different CSV files inside. Ideally, each sub-directory contains the dataset of a different network attack and passing to the script functions the list of sub-directories it is possible to create different ad-hoc datasets as needed. Furthermore, these functions receive the proportions to split the dataset. The function used to create the sub-sets for training and validation receives the proportions of the training, validation and test sub-sets (e.g. `[0.5, 0.2, 0.3]`), while the function that creates the training and test sub-sets receives only two values (e.g. `[0.7, 0.3]`) that represents the proportions between the training and test sub-sets. It is guaranteed that if the same values are passed in both functions for the test set proportion, the test sets used will be the same.

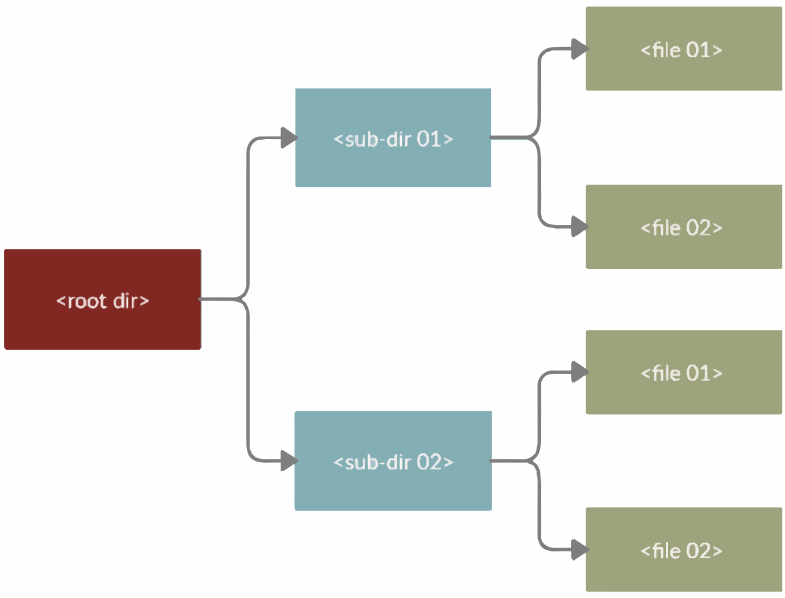

Figure B.1: An example of the dataset directory structure.

Instead, there are two scripts used for the training and are "training.py" and "final_training.py": the first one is used to train many models with different combinations of hyper-parameters, while the second one is used after the validation

phase, to train only the best classifiers chosen by the validation phase. The two configuration files have the same structure, so are described together. The first path contained in each configuration file is the root directory for the dataset, i.e. `<root dir>` using the example explained before. The next line contains the names of the sub-directories, separated by a space: still using the same example, the line would be `<sub-dir 01> <sub-dir 02>`. The subsequent line contains the path to the folder containing the created ML models. For the "training.py" script this is only an output folder, but the other script uses it as an input folder to obtain the hyper-parameters of the best models and as an output folder for the final models that will be used for the integrated Suricata tool. Finally, the last line contains the list of space-separated columns of the dataset to use for the training phases. The reference for the numbers in this line is the table of Tstat features explained in Section 2.2.1, while the chosen ones for this work's experiments have been described in Section 6.1.2. If, for any reason, the developer decides to use a different set of features to create the ML classifiers, it is necessary to change this list inside the configuration file and to modify the Tstat API function that provides the list of features to Suricata. This is a simple process and it is explained in Section B.3.4.

Finally, the last script provided contains the functionalities needed to extract some statistics from the trained models. The script "statistics.py" mainly performs two types of actions: it selects the best set of hyper-parameters for each algorithm starting from the models created with the script "training.py" and then computes some statistics with the final models created with the script "final_training.py". The list of produced statistics, like the F-score or the balanced accuracy, have been presented in Chapter 6, while a theoretical description of these measures can be found in Section 5.1.5. The configuration file of this script has the same structure as the other two, for exception of the last line: it has an additional line that contains the number of neural networks, random forests and SVMs that have to be validated.

Hence, if a user is interested in creating new ML models, the steps to follow can be summed up as:

1. parse the network traffic captures with the script "read_pcap.py";
2. label the obtained CSV files with the script "label_dataset.py";
3. train one or more classifiers with the preferred set of hyper-parameters using the script "training.py";
4. validate the trained models with the script "statistics.py", that produces one model for each algorithm;
5. use the best set of hyper-parameters to train the final models on the whole training set, with the script "final_training.py";
6. optionally, use the script "statistics.py" again to compare multiple final models to perform a sub-selection of the best algorithms.

B.3.3 Suricata integration code

The code provided to integrate the ML functionalities inside Suricata is composed by two parts: a set of C files that has to overwrite the corresponding Suricata source files, as explained in Appendix A, and the new classification scripts with the related configuration files. The Suricata expansion takes place in different points of its pipeline, as explained in Section 6.2: during the initialisation of the tool, during the decoding phase and in the detection phase.

Furthermore, the initialisation has been split in two more steps: a first phase used to setup internal variables and structures used by the ML models and a phase of integration of the new keywords inside the Suricata rule syntax. The file "suricata.c" contains the initialisation routine of Suricata. Here have been performed two main actions: the Tstat `libtstat` has been initialised through its API functions `tstat_init()` and `tstat_new_logdir()` and the content of the configuration file "integration.conf" is parsed; then, the Python interpreter has been initialised and the "ml_detect.py" module with the related `classify()` function has been loaded through the C/Python API. The complete manual to use these API can be found inside the Python documentation[2]. Finally, inside the "suricata.c" file one last action is performed: after the closure of all the Suricata secondary threads, at the end of the process, the `libtstat` API function `tstat_close()` to close and clean the Tstat internal structures is called: in this way the statistics of all the flows processed by Tstat are produced and printed in the Tstat log directory.
Instead, the other file used during the initialisation phase is "detect-flow.c" (and the related headers "detect-flow.h" and "detect.h"). In this case, this file contains the details of the additional keywords presented in Appendix A that have been integrated in the Suricata rule's syntax. At initialisation time, Suricata parses all the available rules and builds a corresponding internal representation for each one of them, by using specific structures and flags. If one of the new keywords is found inside a rule, Suricata adds the corresponding new flags inside the internal signature structure (a C struct, specifically). In this same place, the check for a correct use of the new keywords is performed: as explained in Appendix A, the same keyword can not be used twice in the same rule and the keyword `ml_detect_all` excludes the presence of the other ones.

Then, the next place of the Suricata pipeline that has been modified is the decoding function for IPv4 packets. The file "decode-ipv4.c" contains the call to the `libtstat` API function `tstat_next_pckt()` that is used to pass each packet analysed by Suricata to Tstat: in this way, Tstat can produce its statistics when needed. To avoid conflicts, race-conditions or segmentation errors due to the multi-thread architecture of Suricata, the call to the Tstat API function is performed inside a critical section entered through the possession of a mutex lock, that is released after the completion of the call.

Finally, the detection action performed by the ML classifiers is managed inside the "detect-engine.c" file. As explained in Section 6.2, the new detection process starts only if one of the new flags is found inside the signature that is currently being

[2] https://docs.python.org/3.8/extending/embedding.html

examined by Suricata and in any case only after all the other options of the signature have provided a match, in this way the overhead is reduced to the least amount possible. The statistics of the flow of the current packet are retrieved through a new `libtstat` API function created ad-hoc for this purpose and better explained in Section B.3.4. Then, the `classify()` function contained in the "ml_detect.py" script is called and it determines if the flow is considered positive for at least one of the ML models required by the current signature. The response is returned to the original Suricata pipeline that processes it as usual and performs eventual actions accordingly to the specific rule.

Instead, the "ml_detect.py" script operates in a different way: it contains two functions that are used in different moments of the integration pipeline. The first function is `init()` and receives as input the path of its configuration file. The function is called by Suricata during the initialisation phase, when the Python interpreter is created, as explained above. This function loads from the locations contained in its configuration file the list of ML classifiers and the related scalers to be used at prediction time and returns them to Suricata as `PyObject` references. Instead, the other function is `classify()` and receives as input the statistics of a single flow, created with the new Tstat API function, and the list of ML models and scalers to use for the current classification. It iterates over this list and returns the result of the classification to Suricata.

B.3.4 Selection of Tstat statistics

The "tstat.c" and "tcp.c" Tstat source files have been modified to add a new API function to `libtstat`: the `tstat_get_stats()` function. Given two pairs of IPv4 addresses and ports and a timestamp, this function computes the statistics of the related TCP flow (if existing) using the Tstat internal structures. Inside "tcp.c" the reference to the flow of interest is obtained from a list of flows; then, the extracted statistics are used to create an ad-hoc string similar to the ones produced by Tstat inside its log files. This string of statistics is given back to Suricata that uses it for the classification step. As explained above, if the developer is interested in modifying the list of Tstat features used to create the dataset and perform the predictions, this string of statistics has to be adapted with the ones of interest, beside modifying the training configuration files.

Bibliography

[1] A. Finamore, M. Mellia, M. Meo, M. M. Munafo, P. Di Torino, and D. Rossi, "Experiences of internet traffic monitoring with tstat", IEEE Network, vol. 25, May 2011, pp. 8–14, DOI 10.1109/mnet.2011.5772055

[2] M. Mellia, R. L. Cigno, and F. Neri, "Measuring IP and TCP behavior on edge nodes with Tstat", Computer Networks, vol. 47, January 2005, pp. 1–21, DOI 10.1016/s1389-1286(04)00201-4

[3] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off", Advances in Cryptology - CRYPTO 2003, Santa Barbara (California), August 17-21, 2003, pp. 617–630, DOI 10.1007/978-3-540-45146-4_36

[4] D. M'Raihi, S. Machani, M. Pei, , and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm." RFC-6238, May 2011, DOI 10.17487/RFC6238

[5] S. Ramanauskaite and A. Cenys, "Taxonomy of DoS attacks and their countermeasures", Central European Journal of Computer Science, vol. 1, September 2011, pp. 355–366, DOI 10.2478/s13537-011-0024-y

[6] M. Bogdanoski, T. Shuminoski, and A. Risteski, "Analysis of the SYN Flood DoS Attack", International Journal of Computer Network and Information Security, vol. 5, June 2013, pp. 15–11, DOI 10.5815/ijcnis.2013.08.01

[7] D. Mills, J. Martin, B. J., , and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification." RFC-5905, June 2010, DOI 10.17487/RFC5905

[8] M. Sargent, J. Kristoff, V. Paxson, and M. Allman, "On the Potential Abuse of IGMP", ACM SIGCOMM Computer Communication Review, vol. 47, January 2017, pp. 27–35, DOI 10.1145/3041027.3041031

[9] N. Tripathi and B. Mehtre, "DoS and DDoS attacks: Impact, analysis and countermeasures", National Conference on Advances in Computing, Networking and Security, Nanded (India), December, 2013. https://www.researchgate.net/publication/259941506_DoS_and_DDoS_Attacks_Impact_Analysis_and_Countermeasures

[10] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti, "Millions of targets under attack", Proceedings of the 2017 Internet Measurement Conference, London (United Kingdom), November, 2017, pp. 100–113, DOI 10.1145/3131365.3131383

[11] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art", International Journal of System Assurance Engineering and Management, vol. 8, September 2015, pp. 512–530, DOI 10.1007/s13198-015-0376-0

[12] J. P. Singh, "Analysis of SQL injection detection techniques", Theoretical and

Applied Informatics, vol. 28, February 2017, pp. 37–55, DOI 10.20904/281-2037

[13] M. Feily and A. Shahrestani and S. Ramadass, "A Survey of Botnet and Botnet Detection", 2009 Third International Conference on Emerging Security Information, Systems and Technologies, Athens (Greece), 18-23 June, 2009, pp. 268–273, DOI 10.1109/SECURWARE.2009.48

[14] N. Ianelli and A. Hackworth, "Botnets as a vehicle for online crime", CERT Coordination Center, vol. 2, no. 1, 2007, pp. 19–39, DOI 10.5769/j200701002

[15] Z. Zhu, G. Lu, Y. Chen, Z. J. Fu, P. Roberts, and K. Han, "Botnet Research Survey", 2008 32nd Annual IEEE International Computer Software and Applications Conference, Turku (Finland), 28 July-1 August, 2008, pp. 967–972, DOI 10.1109/COMPSAC.2008.205

[16] C. Kalt, "Internet Relay Chat: Client Protocol." RFC-2812, April 2000, DOI 10.17487/RFC2812

[17] H. J. Liao, C. H. R. Lin, Y. C. Lin, and K. Y. Tung, "Intrusion detection system: A comprehensive review", Journal of Network and Computer Applications, vol. 36, January 2013, pp. 16–24, DOI 10.1016/j.jnca.2012.09.004

[18] B. Woodberg, M. K. Madwachar, M. Swarm, N. R. Wyler, M. Albers, and R. Bonnell, "Configuring Juniper Networks NetScreen & SSG Firewalls", Syngress, 2007, ISBN: 978-1-59749-118-1

[19] J. Postel, "Transmission Control Protocol." RFC-793, September 1981, DOI 10.17487/RFC0793

[20] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges", Computers & Security, vol. 28, February 2009, pp. 18–28, DOI 10.1016/j.cose.2008.08.003

[21] J. M. Estevez-Tapiador, P. Garcia-Teodoro, and J. E. Diaz-Verdejo, "Anomaly detection methods in wired networks: a survey and taxonomy", Computer Communications, vol. 27, October 2004, pp. 1569–1584, DOI 10.1016/j.comcom.2004.07.002

[22] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems", Computer networks, vol. 31, April 1999, pp. 805–822, DOI 10.1016/s1389-1286(98)00017-6

[23] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends", Computer networks, vol. 51, August 2007, pp. 3448–3470, DOI 10.1016/j.comnet.2007.02.001

[24] P. Stavroulakis and M. Stamp, "Handbook of information and communication security", Springer Science & Business Media, 2010, ISBN: 978-3-642-04116-7

[25] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade", Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, Hilton Head (South Carolina), 25-27 January, 2000, pp. 119–129, DOI 10.1109/discex.2000.821514

[26] P. Calhoun, M. Montemurro, and D. Stanley, "Control and Provisioning of Wireless Access Points (CAPWAP) Protocol Binding for IEEE 802.11." RFC-5416, March 2009, DOI 10.17487/RFC5416

[27] T. Worster, Y. Rekhter, and E. Rosen, "Encapsulating MPLS in IP or Generic Routing Encapsulation (GRE)." RFC-4023, March 2005, DOI

10.17487/RFC4023

[28] B. Caswell, J. Beale, and A. R. Baker, "Snort Intrusion Detection and Prevention Toolkit", Syngress, 2007, ISBN: 978-1-59749-099-3

[29] T. M. Mitchell, "Machine Learning", McGraw-Hill, 1997, ISBN: 978-0-071-15467-3

[30] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization", IEEE Transactions on Evolutionary Computation, vol. 1, April 1997, pp. 67–82, DOI 10.1109/4235.585893

[31] S. P. Adam, S.-A. N. Alexandropoulos, P. M. Pardalos, and M. N. Vrahatis, "No free lunch theorem: a review", Approximation and Optimization : Algorithms, Complexity and Applications, pp. 57–82, Springer, May, 2019, DOI 10.1007/978-3-030-12767-1_5

[32] H. Taherdoost, "Sampling Methods in Research Methodology; How to Choose a Sampling Technique for Research", SSRN Electronic Journal, April 2016, DOI 10.2139/ssrn.3205035

[33] D. Anguita, L. Ghelardoni, A. Ghio, L. Oneto, and S. Ridella, "The'K'in K-fold Cross Validation", 2012 ESANN 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges (Belgium), 25-27 April, 2012, pp. 441–446. https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2012-62.pdf

[34] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks", Information processing & management, vol. 45, July 2009, pp. 427–437, DOI 10.1016/j.ipm.2009.03.002

[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", Nature, vol. 521, May 2015, pp. 436–444, DOI 10.1038/nature14539

[36] L. Rokach and O. Maimon, "Decision Trees", Data Mining and Knowledge Discovery Handbook, pp. 165–192, Springer-Verlag, 2005, DOI 10.1007/0-387-25465-x_9

[37] C. J. Mantas, J. G. Castellano, S. Moral-García, and J. Abellán, "A comparison of random forest based algorithms: random credal random forest versus oblique random forest", Soft Computing, vol. 23, November 2018, pp. 10739–10754, DOI 10.1007/s00500-018-3628-5

[38] W. S. Noble, "What is a support vector machine?", Nature biotechnology, vol. 24, December 2006, pp. 1565–1567, DOI 10.1038/nbt1206-1565

[39] C. Wallraven, B. Caputo, and A. Graf, "Recognition with local features: the kernel recipe", Proceedings Ninth IEEE International Conference on Computer Vision, Nice (France), 13-16 October, 2003, pp. 257–264, DOI 10.1109/iccv.2003.1238351

[40] J. Zou, Y. Han, and S.-S. So, "Overview of Artificial Neural Networks", Methods in Molecular Biology, pp. 14–22, Humana Press, 2008, DOI 10.1007/978-1-60327-101-1_2

[41] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks", International Journal of Engineering Applied Sciences and Technology, vol. 4, May 2020, pp. 310–316, DOI 10.33564/ijeast.2020.v04i12.054

[42] A. Painsky and G. Wornell, "On the universality of the logistic loss function", 2018 IEEE International Symposium on Information Theory (ISIT), Vail (Colorado), 17-22 June, 2018, pp. 936–940, DOI 10.1109/isit.2018.8437786

[43] L. Bottou, "Stochastic gradient learning in neural networks", Proceedings of Neuro-Nımes, vol. 91, November 1991, p. 12. https://leon.bottou.org/publications/pdf/nimes-1991.pdf

[44] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network", 2017 International Conference on Engineering and Technology (ICET), Antalya (Turkey), 21-23 August, 2017, pp. 1–6, DOI 10.1109/icengtechnol.2017.8308186

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas (Nevada), 27-30 June, 2016, pp. 770–778, DOI 10.1109/cvpr.2016.90

[46] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston (Massachusetts), 7-12 June, 2015, pp. 1–9, DOI 10.1109/cvpr.2015.7298594

[47] J. Fan, J. Xu, M. H. Ammar, and S. B. Moon, "Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme", Computer Networks, vol. 46, October 2004, pp. 253–272, DOI 10.1016/j.comnet.2004.03.033

[48] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, "Towards a reliable intrusion detection benchmark dataset", Software Networking, vol. 2017, July 2017, pp. 177–200, DOI 10.13052/jsn2445-9739.2017.009

[49] P. Sangkatsanee, N. Wattanapongsakorn, and C. Charnsripinyo, "Practical real-time intrusion detection using machine learning approaches", Computer Communications, vol. 34, December 2011, pp. 2227–2235, DOI 10.1016/j.comcom.2011.07.001

[50] J. Li, Z. Zhao, and R. Li, "Machine learning-based IDS for software-defined 5G network", IET Networks, vol. 7, March 2018, pp. 53–60, DOI 10.1049/iet-net.2017.0212

[51] S. A. R. Shah and B. Issac, "Performance comparison of intrusion detection systems and application of machine learning to Snort system", Future Generation Computer Systems, vol. 80, March 2018, pp. 157–170, DOI 10.1016/j.future.2017.10.016

[52] S. M. Othman, F. M. Ba-Alwi, N. T. Alsohybe, and A. Y. Al-Hashida, "Intrusion detection model using machine learning algorithm on Big Data environment", Journal of Big Data, vol. 5, September 2018, pp. 1–12, DOI 10.1186/s40537-018-0145-4

[53] M. Alkasassbeh and M. Almseidin, "Machine learning methods for network intrusion detection", September 2018. https://arxiv.org/abs/1809.02610

[54] M. A. M. Hasan, M. Nasser, B. Pal, and S. Ahmad, "Support Vector Machine and Random Forest Modeling for Intrusion Detection System (IDS)", Journal of Intelligent Learning Systems and Applications, vol. 6, February 2014, pp. 45–52, DOI 10.4236/jilsa.2014.61005

[55] J. Liu, K. Xiao, L. Luo, Y. Li, and L. Chen, "An intrusion detection system integrating network-level intrusion detection and host-level intrusion detection", 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), Macau (China), 11-14 December, 2020, pp. 122–129, DOI

10.1109/qrs51102.2020.00028

[56] C. Yin, Y. Zhu, J. Fei, and X. He, "A deep learning approach for intrusion detection using recurrent neural networks", IEEE Access, vol. 5, October 2017, pp. 21954–21961, DOI 10.1109/access.2017.2762418

[57] Z. Chiba, N. Abghour, K. Moussaid, M. Rida, *et al.*, "Intelligent approach to build a Deep Neural Network based IDS for cloud environment using combination of machine learning algorithms", Computers & Security, vol. 86, September 2019, pp. 291–317, DOI 10.1016/j.cose.2019.06.013

[58] S. Duque and M. N. bin Omar, "Using data mining algorithms for developing a model for intrusion detection system (IDS)", Procedia Computer Science, vol. 61, November 2015, pp. 46–51, DOI 10.1016/j.procs.2015.09.145

[59] A. Kim, M. Park, and D. H. Lee, "AI-IDS: Application of deep learning to real-time Web intrusion detection", IEEE Access, vol. 8, April 2020, pp. 70245–70261, DOI 10.1109/access.2020.2986882

[60] J. Ren, J. Guo, W. Qian, H. Yuan, X. Hao, and H. Jingjing, "Building an effective intrusion detection system by using hybrid data optimization based on machine learning algorithms", Security and Communication Networks, vol. 2019, June 2019, pp. 1–11, DOI 10.1155/2019/7130868

[61] Z. Pelletier and M. Abualkibash, "Evaluating the CIC IDS-2017 Dataset Using Machine Learning Methods and Creating Multiple Predictive Models in the Statistical Computing Language R", Science, vol. 5, 2020, pp. 187–191. http://irjaes.com/wp-content/uploads/2020/10/IRJAES-V5N2P184Y20.pdf

[62] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set", 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications, Ottawa (Canada), 8-10 July, 2009, pp. 1–6, DOI 10.1109/cisda.2009.5356528

[63] D. Damopoulos, S. A. Menesidou, G. Kambourakis, M. Papadaki, N. Clarke, and S. Gritzalis, "Evaluation of anomaly-based IDS for mobile devices using machine learning classifiers", Security and Communication Networks, vol. 5, June 2011, pp. 3–14, DOI 10.1002/sec.341

[64] M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan, and R. Jain, "Machine learning-based network vulnerability analysis of industrial Internet of Things", IEEE Internet of Things Journal, vol. 6, August 2019, pp. 6822–6834, DOI 10.1109/jiot.2019.2912022