Automatic and on-the-fly Firewalls Configuration

Vincent Rossetto

University of Liège - School of Engineering and Computer Science

Master's thesis carried out to obtain the degree of
Master of Science in Computer Science
by Vincent Rossetto
Academic Year 2020-2021

Supervisor	Pr. Benoit Donnet
Jury	Pr. Laurent Mathy
	Pr. Guy Leduc

Abstract

Automation has the potential to improve reliability and efficiency wherever it takes root. However, firewalls are still configured manually, which is difficult and error prone. As a firewall is essentially a program that determines whether incoming traffic is legitimate or malicious, firewall configuration, given a representative dataset of recorded traffic, can be treated as a classification problem, i.e. a discrete output supervised machine learning problem. The possibility of using a machine learning framework for automatic firewall configuration was studied. A training dataset, composed both of legitimate traffic and attacks, was generated and statistically described both in terms of packets and flows. With the guidance of the statistical observations, per-packet and per-flow features were defined, and the performance of classification algorithms using those features was evaluated. The benefits of a potential feature pre-processing were evaluated. It was found to be useful and even necessary in some cases. A limitation was found in the use of classification algorithms in the form of their lack of interpretability. Practical use of firewalls requires that they can be reconfigured in the case where they make the wrong decision. Such classification errors cannot be avoided completely even with a dataset containing enough examples. This necessity makes the direct use of opaque classifiers for automatic firewall configuration impossible. In conjunction with performance constraints, the most realistic solution consists in configurating classical rule-based firewalls. Rule extraction techniques from opaque classifiers are discussed. The decision tree algorithm was used both to extract rules directly from data and from models built using other classification algorithms. Both approaches yielded a similar classification performance. Finally, rules were extracted from decision trees and translated into a valid format to configure a firewall based on mmb (Modular MiddleBox) using a subset of the per-packet features.

Acknowledgments

I would first like to thank my supervisor Benoit Donnet for giving me the opportunity to work on this interesting subject. He was always available to offer help and advice when I needed it and, for that, I am grateful. He was one of the most important teachers I had during my time at the University and was instrumental in my understanding of computer security and computer science in general.

My thoughts also go to Laurent Mathy, with whom I had the pleasure of studying many interesting subjects, from operating systems to sophisticated data structures and programming patterns, and who entrusted me with teaching the fundamentals of programming to students in the first year of their Bachelor in computer science.

I would also like to thank in particular Guy Leduc, who gave me a taste for networking problems and protocols, and Pascal Gribomont, who taught me about the importance of mathematics and rigor in computer science.

I am, more generally, grateful to anyone in the teaching staff that I had the pleasure to be guided by during my time at the University and the Montefiore Institute. It is my belief that I have received an instruction of great quality and that I could not have hoped for a better formation in the field of computer science.

Finally, I want to thank my friends and family who were always there for me in times of need.

Contents

1 Introduction

Cybersecurity is ever increasingly becoming more relevant. In 2021, global Internet traffic is almost 30% higher that is was in 2019. [McK] As we rely more and more on network applications in our daily lives, it becomes increasingly important to ensure online security.

In 2020, the FBI reported an increase of 400% in cyberattacks. [Inv] The first line of defence against cyberattacks is the firewall, which still today, relies on a difficult, time-consuming, and error prone manual configuration. A study by IBM shows that in 95% of all security breaches, human error is a major contributing cause. [Ser] Automating firewall configuration could help weed out the damages of human error in an increasingly critical piece of software.

The reason why manual firewall configuration is so tedious and error prone is that the system administrator in charge of configurating the firewall must define rules specifying explicitly all of the traffic that is allowed and block everything else.

The purpose of this thesis is to find another way to configure firewalls. Ideally, they should be able to self configure and reconfigure themselves on-the-fly without any human intervention.

To hope achieving this goal, a precise method is going to be necessary, one cannot just guess the rules. There have been many advances in the field of artificial intelligence recently. In particular, supervised machine learning techniques have been used for a variety of problems and have been quite successful in dealing with problems that were thought of as difficult.

A particular sub-type of those techniques is known as classification algorithms. The objective of supervised machine learning algorithms is, given the input values of a certain function and the corresponding output values, to approximate the function as best as possible.

In classification algorithms, the function to approximate has a discrete output. This means that for any given data point, it can be assigned a particular output class. Given many points, they can all be split in the classes that are assigned to them by the function.

The similarity between classification problems in general and firewall configuration is obvious. A firewall must determine if incoming packets are to be accepted or rejected based on the belief that they are either legitimate or malicious. A firewall is thus approximating a function that associates points in the "traffic-space" to the two classes "legitimate" and "malicious".

A firewall itself can thus be seen as a classification problem. This however does not mean that there is no challenge in this problem and a simple out-of-the-box well-known algorithm can be applied to configure it.

The first issue that directly arises is building an appropriate dataset to train the machine learning algorithms. In this thesis, a dedicated testbed was used for traffic generation. The main idea is to generate labelled legitimate and attack traffic to train the algorithms. The testbed that was used consists in two Kali Linux virtual machines running on the same host and connected via a virtual link.

This traffic needs to be generated with particular care to be diverse enough so that attacks and legitimate traffic are not trivial to distinguish. Legitimate and attack traffic should be similar apart from what constitutes their malicious nature or lack thereof. In a real use case, one would record from legitimate applications that must be allowed and they could use traffic records from previous attacks, e.g from an Intrusion Detection System (IDS). In the context of this thesis, the data was generated on the dedicated testbed. To ensure sufficient diversity of traffic, several types of network applications were used with a legitimate client and attacked.

Once a dataset is available, one must decide what constitutes an interesting set of input variables (called features) that are to be fed to the algorithms. To try and achieve this, the traffic was described statistically in order to find out what are the main differences between legitimate and malicious traffic and how those can be used to define expressive features.

Two approaches were considered, treating the traffic per packet or regrouping packets into flows. Statistics were performed and described for each approach in order to define adequate features. The results of the statistics show that, although attacks and the corresponding application often act similarly, attacks tend to adhere to protocols less strictly. Sending rates vary greatly depending on the context, which makes them interesting. Often, only the application layer distinguishes the attack from the legitimate application.

Many classification algorithms were considered and evaluated in terms of their respective performance for both approaches. Most algorithms considered obtained similar results in terms of performance. Only naive Bayesian classification reaches poorer results than the others, most likely due to its unsatisfied assumptions. However, there are a few obstacles to using these algorithms for practical configuration.

Important problems that need to be handled are interpretability and the classification errors. A firewall is unusable in practice if when it goes wrong, e.g blocking legitimate traffic, the system administrator cannot modify it to change its behaviour. Models of the relationship between traffic and the associated output class must thus be understandable and editable.

As it is usually not the the case with classification algorithms, methods to use such techniques to configure a classical rule-based firewall were studied. This essentially means that, in order to be able to use a classification algorithm for firewall configuration, one needs to be able to change it into a set of rules. All algorithms that were considered, with the exception of decision trees, cannot be directly used to derive rules. Decision trees can be used to derive rules directly from data or from models built using other classification algorithms. Both techniques were used on the dataset generated on the testbed and met similar efficiency. The limiting factor in terms of classification error appears to be the set of chosen features.

Finally, once all of these problems have been addressed, the extracted rules must be transformed into a valid format to configure an actual firewall. The solution that is presented configures a firewall based on **mmb** (Modular MiddleBox), which is a generic tool for setting up middle-boxes.

Many of the tasks that were performed in the context of this thesis required writing some scripts.
All related code can be found here: https://github.com/vinci-r/TFE.

Chapter 2 explains and justifies the choice to use **mmb** as a firewall. Chapter 3 describes in details the testbed's configuration. Chapter 4 details the traffic generated on the testbed and describes it statistically. Chapter 5 discusses the choice of features and justifies it based on the statistical analysis. Chapter 6 introduces the considered classification algorithms and their main characteristics. Chapter 7 explains the methodology used for machine learning experiments and presents the results. Chapter 8 explains how extracted rules can be adapted for practical firewall configuration. Finally, chapter 9 concludes this work.

Part I

Technical choices

2 **mmb** as a firewall

For the task at end, not all firewalls are interchangeable. There are many criteria that need to be met, and those all belong to the **mmb** middlebox deployment tool.

What is required is an efficient and powerful firewall, highly configurable with an easy and descriptive grammar. It must be able to work in a stateless fashion for packet discrimination, but also on a higher-level statefully on reconstructed flows. The amount of rules should not have a too heavy impact on traffic performance and rules must be able to be added, removed, or modified on-the-fly while the firewall is still performing its role. It is quite rare to see a firewall (or a generic middlebox tool) checking all of these boxes, but **mmb** was conceived with these objectives in mind.

mmb stands for Modular MiddleBox, meaning it is not merely a firewall but allows us to write different set of rules for traffic forwarding in a very generic fashion. It is actually a tool for configurating any kind of middlebox using user-defined rules. [Ede+19][Ede19]

mmb is a plugin for VPP [fdi], the Vector Packet Processing technology, which is a kernel-bypassing framework. It has a very decent performance, even for large sets of rules, as it aims to reach line-rate forwarding performance. Relying on the underlying capabilities of VPP, as I/O batching, low-level parallelism, efficient caching, it is competitive in terms of performance. It also reaches baremetal-like performance on virtual machines thanks to PCIe passthrough technology. [Ede+19] [Ede19]

Easily configurable with a simple CLI, it is very expressive and allows to express many different types of rules, either statelessly on packets, or statefully on flows. It is protocol-agnostic, dealing similarly with TCP and UDP flows, on which it can do either stateless or stateful matching. It can operate on both transport and network layers' fields. Upon matching, it can drop, forward, or modify packets. [Ede+19][Ede19]

Such a performance and flexibility allows an artificial intelligence to reconfigure on-the-fly the firewall accordingly as it has detected an attack. This, being the objective of this master thesis, will be much more explored later. What can be said at this point is that this could not be done without such an highly flexible firewall which is the cornerstone of all further considerations.

Example of usages of mmb as a firewall

Let us first consider a few simple use cases.

```
vpp# mmb add ip-proto != udp ip-proto != tcp drop
```

Blocks every IP protocol but TCP and UDP

```
vpp# mmb add tcp-opt 22 drop
```

Drops all TCP packets that contain option 22

We can see it is fairly easy to block some unwanted traffic via a simple command on-the-fly. If we want however to define a model of what is acceptable traffic and block everything else, this can also be done easily and intuitively.

```
vpp# mmb flush #erases all pre-existing rules
vpp# mmb add-stateful ip-proto tcp ip-saddr 10.0.0.10/24 accept
vpp# mmb add ip-proto tcp drop
```

Reflexive ACL that blocks everything but TCP connections initated from 10.0.0.10/24

In this example, we define an ACL (access control list) that allows TCP connections originated from a particular subnetwork and blocks everything else. We can see from these examples that, whether we want to exclude a particular type of traffic (for example after detecting an attack) or redefine entirely our model of allowed traffic, this can be done with little difficulty (no advanced programming required) from a simple command line interface.

Part II

Data Generation

3 Testbed description

Designing an AI-powered firewall poses many challenges among which the collection of data to design and train said artificial intelligence. In order to do so, a special test environment to create and record legitimate and malicious traffic was necessary. The configuration of this testbed is fairly simple as a simple network of only two (virtual) machines was sufficient. This configuration is designed solely for traffic generation and capture, hence, it is a fairly simple configuration

In the beginning, the emphasis is put on generating legitimate and malicious traffic to capture in order to have sufficient data for experimentation, various tests and for training learning algorithms. No firewall needs to be active during this phase. The topology here is made of 2 Kali Linux hosts virtual machines connected with each other through a virtual link.

The machines are running on the same host machine through the VirtualBox hypervisor. If Internet access is required for some reason, as for adding some tools, which should be considered very carefully as there will be malicious traffic on the network, a NAT interface can be optionally added to the desired virtual machine. This allows access to the wider Internet with the host acting as a software NAT. However, this should be avoided as soon as malicious traffic has been generated. The use of virtualization can allow to restore it to a safer previous state once infected, if Internet communication were needed, before allowing it Internet access.

As VirtualBox completely isolates machines apart from virtual links, the address of the interface connected to the NAT could be the same for the two machines. It is dynamically assigned through the DHCP protocol by the hypervisor. It does not matter much as the only purpose of these virtual subnetworks is to offer Internet connectivity through the hypervisor software acting as a NAT.

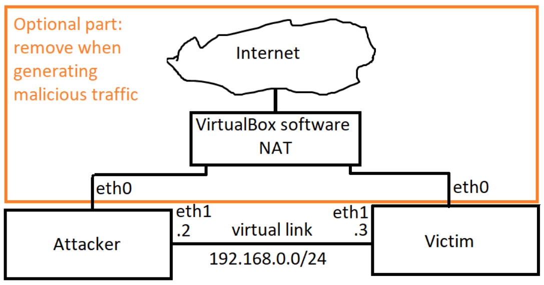

Figure 3.1: Data generation testbed network layout

Each virtual machine was allocated 2.5 GB of RAM and a single CPU core. They both ran under Kali Linux version 2020.3. The choice of Kali Linux as both attacker and target is motivated. Indeed, it is fairly obvious that a distribution specially tailored for penetration testing is particularly adequate for the attacker. The great amount of tools it provides, like the Metasploit framework, makes it perfect for generating malicious traffic.

Using it also on the target, apart from the ease brought by the uniformity of the testbed environment, is also motivated by the fact that a distribution made specially for the teaching of penetration testing might present (voluntarily) a little more vulnerabilities to be taken advantage off. Until fairly recently, every program on Kali Linux ran as the root user. [Kal] This means that although the goal is to record the traffic and not necessarily to infect the target, we are more likely to see the target comply and thus to record more interesting samples of malicious traffic.

In later stages of the thesis, the data generated on this testbed will be used to configure the **mmb**-based firewall. This will be described in the corresponding chapters dedicated to firewall configuration.

4 Generated traffic

The traffic generated on the testbed has the purpose of being used to train (supervised) machine learning algorithms to establish a model of what is legitimate traffic and what is illegitimate traffic (that is traffic used as part of an attack).

This gives us two classes of traffic to detect. These should be the classes the classifier algorithm is trained to detect. For this to work, the traffic will have to be "featurized", which will be discussed later, but more importantly, it must be divided in the 2 classes already.

The solution to this problem is actually pretty self-evident. If we generate and record the legitimate and attack traffic at different and mutually exclusive period of times, they are very simple to divide into the 2 classes. More practically, they are simply recorded into different files labelled with the type of traffic recorded.

Let us now discuss the different types of generated traffic, and the motivations for generating and capturing it. The first thing that needs consideration is that machine learning techniques are very often opaque. They excel at pattern recognition, but often, describing said pattern is not an easy feat. What must be avoided is creating undesired pattern that the algorithm could pick upon.

If we generate many types of attacks that only use UDP, and we generate legitimate traffic that is only TCP-based, the machine learning algorithm would most likely detect that parasitic pattern as a major criterion for discriminating traffic,that is if the network protocol is indeed one of the features, and thus bring nothing of value, discriminating in an inadequate fashion. Worse even, it might do so silently. If the model considered is completely opaque, let's say for example, a neural network, every performance measure of the algorithm might be excellent, but it would only be due to the undesired patterns in the considered learning and testing sets and there would be no way of detecting this silent failure without introducing new data that does not present this pattern.

To avoid introducing this kind of unwanted patterns, the legitimate and undesired traffics must look alike as much as possible, that is in every way but for their "maliciousness" or lack thereof.

The generated traffic is divided in 4 types of applications and exploitations thereof: HTTP (web traffic), FTP (file transfert), SMTP (mail), and UDP applications (audio streaming).

For most types of traffic, no matter whether when generating malicious or legitimate traffic, there was always a simple server (not necessarily vulnerable) running on the target machine and the attacker machine acted as a client. The presence of a server is often necessary as in its absence, the target does not respond adequately and the attack is interrupted before any malicious traffic can be captured. The server needs not be vulnerable as long as the malicious traffic can be captured. The malicious traffic is mostly generated using the Metasploit framework, which usually allows to attack regardless of being aware of a vulnerability and just reports at the end of the attack whether the exploit was successfully or not.

4.1 Statistical description: introduction

In this section, the generated traffic is described statistically with respect to the total number of both packets and flows per class of traffic and proportion thereof. A detailed list of attacks is also provided.

These statistics illustrate similarities and differences between attack and legitimate traffic. The amount of collected data is a prime example of this. The attacks on a given application generate an often significantly different amount of traffic than the actual application. This is an indication we should very much care about the sending rate when trying to classify our traffic.

Another interesting detail would be that attack TCP flows often terminate wrongly. Although this is interesting, it is unlikely to be usable by a firewall which cannot block all packets up to the point of termination, which would prevent normal communication.

These statistics provide our first unanswered dilemma: "Should we classify traffic at the flow or the packet level ?". Some applications, like UDP audio streaming, will appear as one long flow whereas some attacks as port scanning, will create a flow per couple of packets. Which representation for traffic will allow to better recognize those? Would this be adequate for a different kind of traffic? As it is unclear which performs better in a general case, classification will later be done both at the packet and flow level in an investigative comparative study.

Almost all malicious traffic was generated through the Metasploit framework on the attacker machine. Directly shipped with Kali Linux, this framework allows to perform easily a number of known attacks on known vulnerabilities of some software. One just needs to select an attack and pass it basic parameters as the address of the target machine and the targeted port number. A complete list of attacks used is available in section 4.4. The attack traffic that does not originate from the Metasploit framework is simple port scanning generated using the nmap tool, which is also provided with Kali Linux.

Legitimate traffic was generated by genuine use of services running on the target machine. For HTTP traffic, the victim machine ran a simple web server in Python that exposed a few simple HTTP-based forms and pages. The attacker machine acted as a client, requesting pages and filling forms through the Firefox web browser.

For legitimate SMTP traffic, in a similar fashion, the victim machine ran a simple SMTP server in Python. The server did not really transfer mails anywhere, it just received them and did nothing with them. The attacker machine sent a few emails to the server through a small Python script. The emails' body, sender, and receiver were chosen randomly among a pool of predetermined candidate values. This would be very questionable if we were dealing with spam, but it shouldn't matter much here as the considered attacks are very much different from this legitimate traffic and this information will not really be used much by the machine learning techniques considered.

Legitimate FTP traffic was built with the same approach. The victim machine ran a simple FTP server written in Python while the attacker machine, with a similar script, acted as client. The client used the FTP connection to request and download a list of pictures stored on the attacker's machine.

Legitimate UDP traffic was generated in the form of audio streaming. The victim machine shared an audio file to the attacker machine by sending it over UDP on the shared link. The victim and attacker both used the VLC media player respectively as server and client for this audio application.

The simple servers used for HTTP, SMTP, and FTP legitimate traffic generation also ran when generating the malicious traffic targeting vulnerabilities of applications using these protocols. The reason is that it was necessary for the malicious traffic to be generated and captured. Attacks in the Metasploit framework do not send anything if they do not receive responses compliant with the protocol. To capture anything representative of malicious traffic, one must thus comply with the protocol.

All traffic generated, both legitimate and malicious, was captured using tcpdump on the victim machine. Each different application and attack was stored in a different file. This allows an easier analysis of the composition of the total captured traffic for statistical description.

4.2 Packet-related statistics

The statistics in this section come from analyzing the pcpap capture files generated with tcpdump with Wireshark. Some of these statistics can be obtained by merging all of files related to either legitimate or malicious traffic into a single pcap file (for example by using the mergecap tool). However, the traffic generation was interrupted multiple times due to schedule constraints. For recording minutes worth of traffic, there is interruption of several days. This gives an extremely long and mostly inactive time span when analyzing the entire capture file. This also yields extremely low average sending rates which are not representative of the actual network load. This is why table 4.1 gives figures calculated on a time span that is calculated as the sum of the time spans of each capture file, without accounting for the interruptions. This is referred to as capture time. Real time is also given as an indication but it really cannot be interpreted much apart from reflecting the conditions of the experiments.

	LEGITIMATE	MALICIOUS
Number of packets	44373	31038
Number of TCP packets	37024	26494
Number of UDP packets	7349	4544
Average packet size	6437.62 B	2984.27 B
Number of bytes	285 656 784	92 606 267
Time span (real time)	26.07 days	6.97 days
Time span (capture time)	2748.98 sec	1019.216 sec
Average packets/sec (real time)	0.0197	0.0514
Average B/sec (real time)	126.8	153.65
Average packets/sec (capture time)	16.14	30.45
Average B/sec (capture time)	103 913.73	90 860.29

Table 4.1: Packet-related statistics

From table 4.1, we observe 2 important facets of traffic: sending rate and packet size. Indeed, clear differences are observed between legitimate and malicious traffic on both aspects. Malicious traffic seems to consist of more smaller-sized packets sent at a higher rate than legitimate traffic which sends larger packets at a slower rate. We should note if legitimate traffic has a lower packet sending rate, it also has a larger bitrate, meaning that the sending rate is only lower due to the bigger transmission time required for bigger packets. Should we conclude that the sending rate is not important information? That would be too hasty. It was easy to get this general conclusion from the two statistics but it would have been impossible, had we ignored the second. The trend might not be the same for a smaller subset of legitimate and malicious packets and both are needed to be interpretable in this way.

4.3 Flow-related statistics

Another way to look at the traffic from the testbed is by looking at flows. Flows were extracted from the same capture files used in the previous section by using the tshark tool. This tool can be used to directly extract relevant statistics about the flows in a capture file. This is part of what has been done to obtain the statistics in this section. It can also be used to extract in separate files the packets belonging to each flow. Using a custom Python script, this was used to extract additional information that was not given by tshark regarding flow termination (table 4.3).

These statistics (table 4.2) show that an observation that was made at the packet level tends to be true also at the flow level: attack traffic is often made of smaller packets sent at a higher rate. An interesting feature of malicious traffic is that many flows are quite short-lived. This is in part due to the port scanning traffic but not only. For example, many exploit attacks are very small-sized: they comply with the application for a little while until they get an authorization to send a packet containing the malicious payload and then either the exploit succeeds or not, but communication can end as the attack is already completed. Even attacks that generate large amount of traffic tend to generate many flows rather than one large flow. We see that legitimate traffic prefers longer-lived flows. In the particular case of TCP, TCP connections are often reused. The number of packets (see table 4.1) is higher for legitimate traffic but there are nevertheless less flows. The TCP TTL is always identical because it is an artefact of the testbed configuration. TCP RTT does not vary greatly either. It is interesting to see a significantly shorter RTT in one direction but it is unclear how to interpret this. This however somewhat consolidates the idea that the sending rate and delays in general are important characteristics of traffic.

From the information about the proportion of each traffic type (tables 4.2 and 4.3), few conclusions can be made but what is easily observed is that the number of flows is greatly influenced by the application type and whether or not the traffic is malicious. This is an indication that the application type will be important for discrimination. In the subsequent sections, when considering means of discrimination, finding ways to identify the type of application must be a part of the discrimination strategy.

Termination statistics (table 4.3) on TCP flows show that malicious flows are much less likely to end in correctly (using the FIN TCP flag). While this information mostly cannot be used for discrimination, as we cannot simply wait for flows to end before transmitting them without breaking the application, it illustrates an interesting property of attacks: their compliance to protocols is minimal: they only comply enough for the attack to work and are not particularly concerned with complying with standards. This indicates that loose compliance with protocols is something to look for when trying to identify attack traffic. Connection termination cannot be used but network protocols are standardized in many ways. Deviations from the norm could show patterns that would allow identification of attacks.

Retransmission and goodput statistics (table 4.3) show that there is no big difference in goodput. Less retransmissions seem to appear in attack traffic. This can be explained by the smaller in average size of attack flows and maybe to an extent by lack of protocol compliance. This is however somewhat dubious as many attacks would care about reliable transmission of their malicious payload.

From table 4.4, we see that the amount of traffic in a single flow can be very different between legitimate applications and attacks for a given type of application. It is interesting but of little practical application for discrimination. It is however an additional indication that the sending rate is important. This consolidates the idea that results of per-packet and per-flow discrimination might be significantly different and that both should be considered.

	LEGITIMATE	MALICIOUS
Number of flows	1546	3466
Average nb of packets / flow	28.702	8.95
Max nb of packets / flow	7349	32
Min nb of packets / flow	10	1
Mode of nb of packets / flow	10	1
Max nb of KB / flow	9746.03	4969.59
Min nb of KB / flow	1.10	0.058
Average nb of KB / flow	180.44	26.09
Average flow duration	8.21 ms	5.3 ms
Max flow duration	765.17 ms	3509 ms
Min flow duration	0.9 ms	0 ms
Number of TCP flows	1545	2215
Number of UDP flows	1 (large audio streaming flow)	1251
Proportion of size 1 UDP flows among UDP flows (port scanning flows)	0 %	100 %
TCP RTT (Attacker → Victim)	13 ms	52 ms
TCP RTT (Victim → Attacker)	275 ms	264 ms
TCP TTL	64	64
Number of HTTP flows	1273	1103
Proportion of HTTP flows among TCP flows	82.39 %	49.80 %
Number of FTP flows	72	8
Proportion of FTP flows among TCP flows	4.67 %	0.36 %

Table 4.2: Flow-related statistics

	LEGITIMATE	MALICIOUS
Number of SMTP flows	200	105
Proportion of SMTP flows among TCP flows	12.94 %	4.74 %
Number of port scanning flows among TCP flows	0	999
Proportion of port scanning flows among TCP flows	0 %	45.10 %
Number of port scanning flows among UDP flows	0	1251
Proportion of port scanning flows among UDP flows	0 %	100 %
Number of TCP flows ended with ≥ 1 FIN flag	1545	1128
Proportion of TCP flows ended with ≥ 1 FIN flag	100 %	50.94 %
Number of TCP flows ended "normally" with ≥ 2 FIN flags	1544	124
Proportion of TCP flows ended "normally" with ≥ 2 FIN flags	99.94 %	5.6 %
Number of TCP flows ended with ≥ 1 FIN, 0 RST flags	1545	29
Proportion of TCP flows ended ≥ 1 FIN, 0 RST flags	100 %	1.3 %
Number of TCP flows ended with timeout	0	0
Proportion of TCP flows ended with timeout	0 %	0 %
Average number of unique bytes per TCP flow	176 833.29	36 223.70
Average number of retransmitted bytes per TCP flow	433.01	0
Average TCP goodput	99.21 %	98.52 %

Table 4.3: Additional flow-related statistics

	LEGITIMATE	MALICIOUS
HTTP	7380.174	66061.99
FTP	3 662 699.33	278.56
SMTP	490.57	292.42

Table 4.4: Average number of bytes per TCP flow type

4.4 Details of attacks

A detailed list of attacks, proportion of traffic they represent and short descriptions are provided below. All attacks are performed except for port scanning are performed using the default configuration in the Metasploit framework. Only essential parameters as target IP address and port were passed as parameters. nmap was used for the port scanning. For TCP, SYN scan was used (option -sS). A SYN scan consists in sending the first packet to initiate a TCP connection and wait for a response.[Don19a] The TCP connection is never really initiated (TCP's handshake requires an exchange of 3 packets). For UDP, empty (in terms of payload) UDP packets are sent. If a UDP response is received, a UDP service is running. If an ICMP notification is received instead, the port is closed or filtered.[Don19a] This is done with nmap's option -sU.

Tables 4.5 and 4.6 illustrate the extent of the variety of the attack traffic. We also see that although some attacks generate much more traffic than others, it is interesting to see that the proportions vary greatly depending on whether we analyze traffic per packet or per flow.

Name	CVE-ID	EDB-ID	% of flows	% of packets
Easy FTP buffer overflow	N/A	16737	0.028 %	0.058 %
Filezilla FTP Server Malformed Port DOS	2006-6564 2006-6565	2914	0.028 %	0.09 %
FTP JCL execution	N/A	N/A	0.057 %	0.12 %
War-FTPd 1.65 Password Overflow	1999-0256	16706	0.028 %	0.045 %
War-FTPd 1.65 'USER' Remote Buffer Overflow	2007-1567	3570	0.028 %	0.038 %

Table 4.5: Attacks list (part 1)

Name	CVE-ID	EDB-ID	% of flows	% of packets
Pure-FTPd-External Authentication Bash Environment Variable Code Injection	2014-3659 2014-3671 2014-7196 2014-7227 2014-7910 2014-6271 2014-6277	34862	0.057 %	0.103 %
Apache Tomcat Manager-Application Deployer Authenticated Code Execution	2009-3548 2009-3843 2009-4188 2009-4189 2010-0557 2010-4094	16317	0.028 %	0.032 %
Apache Tomcat - CGI Servlet enableCmdLine Arguments Remote Code Execution	2019-0232	47073	0.46 %	0.7603 %
Oracle Weblogic Apache Connector - POST Buffer Overflow	2008-3257	18897	0.028 %	0.032 %
Win VNC Web Server 3.3.3.r7 - GET Overflow	2001-0168	16491	0.028 %	0.039 %
Hashtable HTTP Collisions DOS module	2011-4858 2011-4885 2011-5034 2011-5035	N/A	28.85 %	82.98 %
Open SMTPd - MAIL FROM Remote Code Execution	2020-7247	48038	0.057 %	0.051 %
Port scanning (nmap)	N/A	N/A	67.37 %	11.01 %
SMTP fuzzing	N/A	N/A	2.897 %	4.48 %
Haraka SMTP command injection	N/A	41162	0.028 %	0.09 %
Mercury Mail SMTP Auth Cram-MD5	2007-4440	16281	0.028 %	0.064 %

Table 4.6: Attacks list (part 2)

A short description of each attack is provided below.

Easy FTP buffer overflow

This attack exploits a stack-based buffer overflow in EasyFTP Server 1.7.0.11 and earlier. The server fails to check input size while parsing 'CWD' commands, which leads to a buffer overflow. A small payload of roughly 500 bytes is injected into a 264 bytes buffer 'fixing' the return address post exploitation. [Sec] [Rap] The malicious payload can be chosen. The output of the attack is to make the server run arbitrary code. Usually, this is used to establish a reverse shell connection.

Filezilla FTP Server Malformed DOS

This attack triggers a denial-of-service condition in the FTP Server Administration Interface in version 0.9.4.d and earlier. By sending a procession of excessively long USER commands, it overwrites the stack and causes an exception. [Sec] [Rap]

FTP JCL execution

This attack exploits a vulnerability in the FTP server on z/OS mainframes that allows issuing of JCL jobs to run arbitrary code. z/OS is an operating system by IBM that was created in 2000 for their newer mainframes. JCL stands for Job Control Language. It is a collection of scripting languages used on IBM mainframes to create batch jobs and for starting subsystems. An authenticated user is able to provide code that will be executed by the server through a JCL jobs if they can upload files. [Sec] [Rap]

War-FTPd 1.65 Password Overflow

War-FTPd is a free FTP server for Windows. A buffer overflow found in the PASS command in War-FTPD 1.65 is exploited. A successful exploit makes the server execute a malicious payload consisting of arbitrary code chosen by the attacker (usually to initiate a reverse shell connection) but even failed attempts can bring the service down completely (acting de-facto as DoS). [Sec] [Rap]

War-FTPd 1.65 'USER' Remote Buffer Overflow

A buffer overflow found in the USER command in War-FTPD 1.65 is exploited.

Successful exploit makes the server execute a malicious payload, i.e. arbitrary chosen by the attacker (usually to initiate a reverse shell connection). [Sec] [Rap]

Pure-FTPd-External Authentication Bash Environment Variable Code Injection

This exploits the ShellShock vulnerability, a flaw in how the Bash shell exploit arguments. The server is often set up to use an external Bash script for authentication which allows to make use of the vulnerability to make it run a malicious payload. [Sec] [Rap]

Apache Tomcat Manager-Application Deployer Authenticated Code Execution

This attack exploits Apache Tomcat servers with an exposed manager application. It uploads a payload as a WAR archive containing a jsp application using a PUT request. The manager could also be directly attacked using an upload page in the manager application but this exploit works without doing this. [Sec] [Rap]

Apache Tomcat - CGI Servlet enableCmdLineArguments Remote Code Execution

This exploits a vulnerability in Apache Tomcat's CGIServlet component. When a certain setting (enableCmdLineArguments) is set to true, a remote user can make use of the vulnerability to execute system commands, ultimately giving them the possibility of remote code execution. [Sec] [Rap]

Oracle Weblogic Apache Connector- POST BufferOverflow

Weblogic is web server by BEA Systems. There is plugin for this web server used as a bridge for an Apache server that serves static and dynamic pages to delegate dynamic pages generation to the Weblogic server. This attack uses a stack-based buffer overflow in the BEA Weblogic Apache plugin that fails to properly handle some specially crafted HTTP POST requests. This causes a buffer overflow due to improper use of C's $sprintf$ function. [Sec] [Rap]

Win VNC WebServer 3.3.3.r7 - GET Overflow

The AT&T WinVNC web server v3.3.3r7 (and below) presents a buffer overflow vulnerability in certain conditions. If debugging mode with login is enabled, an excessively long GET request can overwrite the stack. [Sec] [Rap]

Hashtable HTTP Collisions DOS module

This module takes advantage of a common vulnerability in many applications and programming languages. It generates elements that are going to be placed in a hash table. These objects, once passed into the hash function, all have the same output (the hash). This can be used to make a web server get stuck on parsing the parameters of a single HTTP POST request during up to hours of CPU time. [Sec] [Rap]

Open SMTPd - MAIL FROM Remote Code Execution

This attack injects a command in the MAIL FROM field during SMTP communication with an OpenSMTPD server to execute a command as the root user. [Sec] [Rap]

Port scanning (nmap)

This application can be used to identify open ports on a given host. Using a variety of techniques like SYN scanning, ACK scanning, or UDP scanning, packets are sent on the entire range of available ports and depending on the response (or lack thereof), a conclusion is made about which ports are open or likely to be. [Don19a] This information can be further used to start more malicious attacks. For example, if port 80 is open, one can assume the presence of a web server.

SMTP fuzzing

Simple SMTP fuzzer. Generates a variety of inputs, in the form of slightly modified SMTP requests. Depending on the response to such a malformed request, bugs and vulnerabilities can be revealed. [Rap]

Haraka SMTP command injection

A vulnerability in the extension of the Haraka SMTP server for processing attachments in versions 2.8.9 and below allows to inject commands that will be executed by the server.[Sec] [Rap]

Mercury Mail SMTP Auth-Cram MD5

Sending a specially crafted argument to the AUTH CRAM-MD5 command of the Mercury Mail Transport System 4.51,which is used to provide the hash of a combination of the user’s password and data from the server for authentication purposes, the attacker can cause a stack buffer overflow by which he can execute arbitrary code. [Sec] [Rap]

Part III

Automatic configuration

5 Candidate and selected features

With some guidance from the statistical description of both malicious and good traffic, it is time to consider how the data is to be "featurized". Let us explain what it means. Supervised machine learning algorithms function in the following fashion. They are given as inputs a set of inputs and outputs of a certain function and in a way specific to the algorithm, they approximate to the best of their possibilities that function by a model they build. This is called training of the algorithm.

For some machine learning algorithms, like decision trees, that model can actually be transformed into an interpretable and readable function that is equivalent to the model.[GW19] Some other techniques, called "black-box" algorithms have no direct translation between the model and some readable function. In that case, your model is the only exact representation of the approximated function.[GW19]

When we talk about features, we are talking about the specific form of input variables. They are n-dimensional variables, that is vectors of values. They can be numerical or qualitative attributes of the data they represent. In the case of classification, the associated output of each feature vector is a simple label, its class.[GW19]

In more concrete terms regarding the specific problem of automatic firewall classification, we want to identify if packets or flows are either legitimate or malicious. "Legitimate" and "malicious" would thus be the outputs or classes and what must be determined now is which set of input variables (the features) are the most adequate to represent packets and flows in the context of attack detection.

Before making choices regarding candidate features, there are few things that need to be explained. We should keep in mind that the objective is to find models of traffic legitimacy that can be used to configure classical rule-based firewalls. Other discrimination strategies could be used, like generating features evenly spread across the bytes of packets. This strategy has been considered in other research work and has been somewhat successful for the considered data. [Ver14]

However, models resulting from such techniques can only be very opaque and complicated. They also lack in technical motivation and are very different from the type of rules that are used to configure firewalls currently in use. The objective of the research performed here is rather to choose features in a way that will allow to build simpler models from features that are far from random but are rather originated from a knowledge of cyberattacks' mechanisms.

The main advantage comes in the form of interpretability. An important consideration is the practicality of use of an automatically configured firewall. Let us imagine the firewall blocking packets due to values present in different combinations of arbitrary bytes suffers from a false positive issue, that is, it blocks packets that should not be. That would create a nigh-impossible situation for a system administrator tasked to identify which rules have to be removed so that legitimate traffic is allowed again.

This is why technical motivation, simplicity, and interpretability are paramount in the choice of features performed here. In the following section, candidate features, of common use in firewall rules and/or of great technical motivation and interpretability will be discussed. The statistical analysis performed earlier will also be of great help in assessing the potential decision power harbored by said features.

5.1 Discussion of candidate features

Per-packet discrimination: source address

This looks like a prime candidate as feature. Indeed, if an attacker is identified, why should they be allowed to keep sending traffic to the protected network? Nevertheless, this approach is too simplistic for several reasons. First, this is quite inefficient. An attacker could spoof their IP address, or use one address per few packets of attack traffic. Spoofing is known to be extremely prevalent. [Don19a]

Secondly, and more importantly, there is a significant risk of denying access to legitimate users. When possible, security measures should avoid denying legitimate access, otherwise many attacks are automatically transformed into denial of service (DOS). In the context of a firewall, we should remember that legitimate users are susceptible to be infected by malware. When this occurs, legitimate users keep on with their legitimate use but also become a proxy for the attacker to perform attacks. In summary, legitimate users can be attackers, and thus, their access cannot be simply revoked. This is the reason to **reject** this feature in the context of **per-packet discrimination**.

Per-packet discrimination: destination address

When working at the packet level, an interesting question we may ask is the direction of traffic. Are packets leaving the protected network or are they entering it? It may be interesting to make a distinction between those 2 cases. The protected subnetwork is mostly assumed to be safe, as the role of a firewall is to protect this smaller network from attacks originating from other networks or the wider Internet. A response to an attack is to be considered an attack itself (as the attacker may have exploited a vulnerable host), but it does not matter much as if attacks cannot go through, they won't get any response. This does not mean we should not prevent anything from leaving the domain but it is certainly sufficient enough motivation to **keep** destinations addresses **as features** in the context of **per-packet discrimination**.

Per-flow discrimination: Addresses of communicating hosts

In the context of per-flow traffic discrimination, traffic is bidirectional. A request and its response are to be grouped in the same flow. Source and destination addresses are thus often permuted, and at the flow level, we can thus no longer talk about source or destination but just in terms of pairs of addresses of communicating hosts. Every reason for not considering source addresses cited above are still valid. As destination addresses were kept only for the matter of identifying packet's direction, which has become meaningless in this new context, both addresses of communicating hosts are **rejected as features for per-flow discrimination**.

Destination and origin ports

These represent very pertinent information. Indeed, in many applications, the server uses a well-known port. This is low-level information that can be used to determine information about the application layer, which is thus very valuable. The client often uses random port, but it is possible that some attacks always use the same client port, which could help identify them. In terms of destination and origin ports, those will be inverted depending on whether the message is from client to server, or server to client. Nevertheless, in any case they will correspond to either a client or server port whose usefulness for discrimination was already argued. These arguments are valid in both per-packet and per-flow discrimination. Destination and origin ports are thus to be **kept as features both for per-packet and per-flow discrimination** (as a pair of communicating ports).

Transport layer protocol The form of traffic varies greatly depending on the transport-layer protocol. TCP packets/flows offer reliable data transport and congestion controls, while their UDP equivalent are free to do so at the application layer if so needed. Describing a flow without specifying the transport protocol would be a poor idea as these mechanisms could have an important impact on the sending rate which could be an important characteristic of many attacks and legitimate applications. Conformity with the TCP protocol could also differ between attacks and legitimate traffic. In the statistics on generated traffic, we observed that most attack TCP flows were ended wrongly (with RST). This in particular won't be usable in a firewall but maybe these attack flows take other shortcuts with the TCP specification. This would go unnoticed if we treat TCP flows indifferently from UDP flows. The transport layer protocol is to be **kept as feature both for per-packet and per-flow discrimination**.

Per-packet discrimination: TCP/UDP checksum

The checksum is optional in UDP but not in TCP. UDP checksums are potentially more often set to 0 in attack traffic which more often does not care about reliable delivery and error correction. When used correctly, both in TCP, or in UDP (whether by a legitimate application, or an attacker complying with its application-layer protocol), the checksum contains information derived from the payload, that is application-layer data. Such a data summary is quite pertinent as attacks often target the application and not the underlying network stack. **The TCP/UDP checksum should be kept as feature for per-packet discrimination**.

Per-flow discrimination: TCP/UDP checksum

In a flow context, talking about the checksum of a singular packet does not make much sense. Nevertheless, some pertinent information could be derived from the checksums of packets in the flow. The proportion of checksums=0 would be interesting. It would allow to identify easily UDP flows that do not use the checksum (their proportion would be 100%).
Another interesting derived measurement could be the average checksum. Averaging checksums makes little sense in the context of error correction but this can create a summary of typical payload data. Similarly to the proportion of null checksums, this would allow to easily identify UDP flows which do not use the checksum as the average would be 0. Information offered by these two measurements would be heavily correlated as they are derived in a similar way from the same data. As the second encompasses most advantages of the first, it would be wiser to only use it and not the first. Dimensionality reduction is often necessary in machine learning and other types of pattern recognition when considering too high a number of features. Performance deteriorates past a certain point due to lack of training samples for each possible combination of features. This is known as the curse of dimensionality.[Tru79] If two features are a projection of the same dimension of a physical reality they will be highly correlated. Keeping the most expressive and rejecting the second is the reasonable thing to do. **Average TCP/UDP checksum** is thus **kept as feature** while **the proportion of 0-checksums is rejected**.

TCP flags

ACK, RST, SYN, FIN are used for connection setup and termination. The TCP connection handshake is mandatory at the beginning of each connection and termination data is only available when the flow ends which is potentially very long after connection setup. A firewall cannot intercept all packets before connection termination as it would break the application. These are thus irrelevant and **rejected as features** as they will be either similar for legitimate and attack traffic, or simply unusable.
NS, CWR, ECE, URG are likely to be set by middleboxes on the path rather than the attacker or application and there is no reason to think they would be used significantly differently in an attack than a legitimate application anyway. They are thus **rejected as features**
PSH has no UDP equivalent. It might make some sense in a denial-of-service

attack to avoid all buffering by always setting PSH=1. It can be interesting to considered as a feature precisely because of this eventuality. It can simply be set as 0 or N/A for UDP packets. The **TCP PSH flag should be used as a feature in the context of per-packet discrimination**. This can be adapted for per flow discrimination, for which it is pertinent for the exact same reason by simply taking **the proportion of packets in the flow with PSH=1**. This should be used as a feature in the context of **per-flow discrimination**.

TCP options

TCP options are impractical as features due their optional nature and variable length. They should thus be **rejected as features**.

Per-packet discrimination: time since previous packet

The sending rate is an important feature of any network application (e.g. critical in audio/video streaming) and of many attacks (e.g. flooding = high sending rate). The time since the previous packet should somewhat reflect the sending rate. It is however not an excellent estimator as it will be influenced by packets unrelated to the flow the packet belongs to. This is impossible to take into account without working at the flow level. For this reason, the estimator is probably the best we can do in that regard. It should be **used as feature for per-packet discrimination**.

Per-flow discrimination: average time between 2 packets

This is a much better estimator of the sending rate and should thus be used **for per-flow discrimination as a feature**.

(Average) payload length

Attacks often target a vulnerable application, not the network stack. Data pertaining to the application is thus extremely valuable. One of the most basic ways to describe data is its size. It is thus only natural to use payload length as a feature. Combined with checksum and port numbers, we easily get an overview of what is going on at the application layer. **Payload length** should thus be used as **feature for per-packet discrimination** and **average payload length for per-flow discrimination**.

Summary of chosen features

Below is a table summarizing the features chosen for each mode of traffic discrimination.

Features for per-packet discrimination	Features for per-flow discrimination
Destination address	Pair of communicating ports
Destination port	Transport layer protocol
Source port	Average TCP/UDP checksum
Transport layer protocol	Average time between 2 packets
TCP/UDP checksum	Average payload length
Time since previous packet	Proportion of packets with TCP PSH flag
Payload length	
TCP PSH flag	

Table 5.1: Selected features

6 Considered supervised machine learning algorithms

We want to build a model that is able to classify traffic into two classes: legitimate and malicious. It is a classical classification problem to which we can apply supervised machine learning that usually perform well for classification. There is a major trade-off that arises from the nature of the following techniques: **interpretability** versus **efficiency**. Interpretability means the insight that the derived model gives on the relationship between the features and the output of the classifier.

Packet and flow discrimination is indeed essentially a classification problem but there is an additional consideration to keep in mind: the objective is to configure a classical rule-based firewall. If the classification error is minimal but the model cannot be used to configure a firewall, it is essentially useless. A firewall cannot be purely black-box. Aside from very serious performance considerations, in case of misclassification, legitimate traffic is blocked. The system administrator must imperatively be able to bypass the model to restore legitimate network applications' functionality. It would be nigh impossible on a custom firewall based on an black-box classifier.

This is why interpretability is paramount. Unfortunately, the most efficient machine learning techniques are, when not entirely opaque, difficultly transformed into a set of rules. This problem is however mitigated by the fact that there exist techniques and algorithms to change an opaque classifier into a set of rules. [HBV07] This is particularly interesting as this allows a study of different models' performance without considering the issue of interpretability. However, it should be kept in mind that these techniques can lose some of the model's power as they do not yield a direct translation. [HBV07] Let us now go over a few candidate algorithms.

6.1 Decision trees

It is not expected that decisions trees perform particularly well compared to more advanced techniques. However, they should be included regardless because of their ease of interpretability. Indeed, they allow for a direct translation of the model built into a set of rules. The idea is to build a tree to classify the data. Each node contains a set of data points,
that is a pair {Features' values, Output class}.

The root contains the entire learning set used to build the tree, and the leaves contain points that will be classified in the same way. The tree is built by recursively splitting the node into smaller sets that will become their children according to the value of one of the features. The branches represent that value, all nodes below that branch are nodes for which the feature used as splitting criterion exhibits the same value. A path from root to leaf determine a combination of feature values and the associated class.[HBV07] [GW19] Figure 6.1 shows an example of a simple decision tree that could be learned from packet data. In this example, the algorithm has detected a simple pattern in which the only allowed traffic is HTTP.

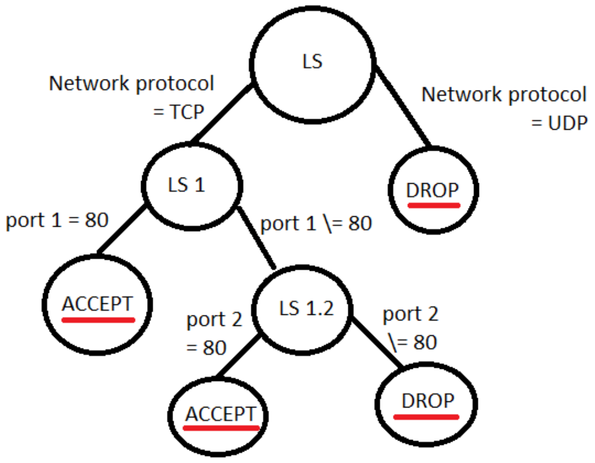

Figure 6.1: Example of decision tree

One could build such a tree so that every element of the learning set is perfectly classified. For a set of such points, the tree can be built recursively with the following procedure[HBV07]:

- If the set contains no element, then it is assigned a class according to some heuristic.
- If the set contains elements of a single class, then it is assigned that class.
- If the set contains elements from multiple classes, then it is split in multiple subsets to which the procedure is applied recursively. The split is based on a single feature's value.

A somewhat important detail is left to explain, which attribute should be used to perform the split. An optimal tree would be the smallest tree (in terms of depth) possible that perfectly classifies the set. Not all choices can be considered as this would be too computationally expensive when considering the whole tree. Finding such an optimal tree is NP-hard. [HBV07] [GW19]

A way to solve this issue is to use a greedy heuristic such as the entropy of a set. Information theory defines the entropy of a set S as

$$H(S) = -\sum_i p_i log_2(p_i) \tag{6.1}$$

where i are the values observed in the set and p_i their frequency of occurrence.

In the case of classification, the values i of interest would be the different classes (/outputs). The best attributes to use as a splitting criterion would be the one yielding the best reduce in entropy, that is the splitting criterion (/feature) $f*$ that maximizes the information gain [HBV07] [GW19]:

$$Gain(S,f) = H(S) - \sum_{k=1}^{N} \frac{|S_k|}{|S|} H(S) \tag{6.2}$$

$$f^* = \text{arg max}_f \text{ Gain}(S,f) \tag{6.3}$$

N being the number of subsets for a given splitting criterion f, S_k being the different subsets for that criterion f.

The algorithm explained above allows for the perfect classification of a given set. However, perfect classification is most often not desirable. A common issue with supervised machine learning algorithms is overfitting.[GW19] Overfitting means that the model built to approximate the relationship between inputs and outputs is too close to the sample data, yielding an overcomplicated model that fits the data more than the actual link between inputs and outputs. Such a model is more costly due to its complexity and performs worse than a model without overfitting.

A solution to avoid overfitting of the decision tree is to stop the splitting before gaining a perfect classification of the learning set. This approach is no longer used as it is not the best one. It presents the disadvantage of stopping with a look-ahead of only one step due to the greedy approach.[GW19] The modern approach consists in in a separate post-pruning phase on the complete tree. [HBV07] [GW19]

A very interesting property of decision trees for the particular problem of firewall configuration is that it is a generic method that can be used to extract rules from any opaque model. One can simply use their opaque classifier on the learning set and then use the pairs {Inputs, Outputs from the opaque model} as training data for a decision tree. Doing so, one generally only keeps the correctly classified pairs. One can then get rules from the derived decision tree. [HBV07]

Such a generic technique is extremely valuable when it is essential to extract rules from the model as in firewall configuration. Other generic techniques for rule extractions exist but they mostly come in the form of algorithms similar to decision trees, that is algorithms that can learn directly from data. [HBV07] The additional step of deriving a better performing intermediate classifier thus only makes sense if the rules derived from the data itself perform worse than the ones derived from the intermediate classifier.

However, investigating algorithms that perform better than decision trees is not worthless. Effectively separating the building of the classifier from the rule extraction presents the advantage of modularity. If progress is made either in the field of classification, or in rule extraction, that component can be upgraded independently from the other. This is the reason why it is important to evaluate the performance of supervised machine learning algorithms other than decision trees for packet and flow classification.

6.2 Naive Bayesian classifier

The Naive Bayesian classifier is based on the usual assumption of independence in bayesian inference. We simply consider all features' influence on the output to be independent and thus consider the probability of a given output given different observed values as the product of the probabilities of observing the same output given each of these values with the probability of said value (see property 6.4). Such classifiers have been used with a lot of success in antispam filters. [Don19b]

It would be somewhat surprising if similar performance could be observed in firewalls as features in both packets and flows are not independent and considering them as such could ignore the most interesting patterns. However, it should be noted that in email filtering, the words are not independent either and they still perform well, notably on the important criterion of the false positive rate. [Don19b]

Let us consider the input vector as a set of random variables. Each feature corresponds to a random variable and so does the output class. We can thus consider frequencies of the possible values for the features over a given learning set. We can also simply compute the frequency of a given output class conditioned upon the value of a given feature. Assuming independence of the features, we can simply compute the probability of a given class conditionally to a set of features values as proportional to the product of the individual prior probabilities with the probability of the class itself. [Don19b] [NS11]

$$\begin{aligned} &P(\text{Class} = c \mid X = \{x_1, x_2..., x_n\}) \\ &\propto P(\text{Class} = c)\, P(X_1 = x_1 \mid \text{Class} = c) P(X_2 = x_2 \mid \text{Class} = c) \ldots P(X_n = x_n \mid \text{Class} = c) \end{aligned} \tag{6.4}$$

Classification is thus quite simple. One just calculates the prior probabilities for each feature value in the learning set and the frequency of each feature's value (in the LS). Then, new inputs can be classified by finding the class with the maximal probability for that given set of features (/ inputs) values.[GW19] Mathematically this means maximizing the posterior probability , that is finding $c*$ such that[GW19]:

$$c* = \arg\max_c P(\text{Class} = c \mid X = \{x_1, x_2..., x_n\}) \tag{6.5}$$

This (6.5) can be easily computed by considering each possible value of *Class*, applying property 6.4 and normalizing the results so that the sum for the posterior probabilities of each class amount to 1. The value of *Class* yielding the maximal value used as index for this probability vector is $c*$ and will thus be the class assigned to this set of features.

In terms of interpretability, although the model does not transform directly into rules, at least not the type of rules that we typically use in firewalls, it is not entirely opaque. The classifier can, for any combination of features, find the output class but can also give a measure of certainty of that result in the form of its probability.

This may be interesting in the case of binary classification as performed by a firewall. One may decide to only drop packets that are assigned the class "Malicious" if they are assigned that class with a certain threshold of certainty (e.g. 70%). This would allow to diminish the false positive rate at the price of an increase in the false negative rate. The reverse trade-off can obviously also be done as well.

One is off probably better not playing with said thresholds as both true and false negatives can be costly and dangerous but nevertheless, this measurement is available for interpretation. In case of failure of the classifier, one may use it as an indication of what went wrong.

6.3 k-nearest neighbors (k-NN)

In k-NN classification, the class assigned to a given sample is the one that is the most common among its k "nearest" neighbors. The intuition behind such an algorithm is that inputs that are close should have a similar output. [GW19]

Several metrics are possible for the distance on which this algorithm relies heavily. For continuous input variables, Euclidean distance is common. Other metrics, such as Hamming distance, are more adapted for discrete features. [NFS12]

Euclidean distance between two vectors $\mathbf{x}$, $\mathbf{y}$ can be written as:

$$d_{\text{euclid}}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_k (x_k - y_k)^2} \tag{6.6}$$

The lesser-known Hamming distance is defined as:

$$d_{\text{hamming}}(\mathbf{x}, \mathbf{y}) = \sum_k (x_k \neq y_k) \tag{6.7}$$

where we use the usual computer science convention for the predicate, i.e its value is 1 if true, 0 otherwise. The advantage of this metric is that it measures "overlap" between the vectors. [NFS12] This is often an adequate choice for vectors of discrete variables as it often makes little sense comparing discrete values apart from an equality check.

In its simplest form, the training of the algorithm consists simply in keeping the learning set in memory. Classification of new inputs is done by finding the k points of the LS that are the closest to the features according to whatever distance metric we are using and taking the mode of the classes of the so-called neighbors. [GW19] [NS11]

For a given input vector $\mathbf{x}$, and a distance metric d, we have:

$$\text{Neighbors}(\mathbf{x}, d) = \text{First}(k, \text{Sort}(LS, f(\mathbf{y}) = d(\mathbf{x}, \mathbf{y}))) \tag{6.8}$$

where $f(y)$ is the function used as a key for sorting. Sort is a function returning the ordered version of a vector ordered using the function f, First(k, $\mathbf{x}$) returns the first k elements of the ordered vector $\mathbf{x}$. [GW19] Once said vector is found, one only needs to find its mode for the class, that is the most output common value.

As such, the algorithm is limited in performance by the size of the learning set. Luckily, the algorithm can be used as such in a more scalable fashion by removing objects from the stored LS that removing does not affect (much) the predicted values and also the points that should be removed due to being statistical outliers. [GW19] This does not affect the procedure in any way apart from this additional step.

The models originated from this algorithm are easy to use and can have decent results in terms of performance, although they depend a lot on an adequate distance metric. It is intuitive that similar inputs yield similar outputs.

However, this somewhat lacks interpretability. Apart from the original intuition, the model itself does not enlighten the relationship between inputs and outputs in a new way different from this original intuition. Rule extraction techniques will be precious for extracting additional meaning from this model. In the case of firewall configuration, they will be essential.

6.4 Random forests

Natural extension of decision trees, random forests are built quite intuitively by constructing several random decision trees on a part of the learning set and averaging their output (or taking the mode in the context of classification). [GW19]

These trees are random regarding both the cutoff point used for their depth and selection of features used as splitting criteria (the splitting criterion is no longer the best, but only the best among a random subset of features). They usually outperform decision trees [GW19] at the cost of ease of interpretability.

In the case of classification, averaging outputs simply means taking the mode of the output of the different classifiers considered. Concretely, to build a random forest, one builds k random decision trees from part of the learning set[GW19]: $T_1, T_2, ...T_k$.

To classify a new input, one asks each of the tree classifiers $T_1, ..., T_k$ to classify the input x and gets a k-sized output vector $\mathbf{y}$. The most frequent value in the vector $\mathbf{y}$ will be the output associated to x.[GW19] We can write this as:

$$y = \text{mode}(\mathbf{y}) \tag{6.9}$$

$$\mathbf{y} = (T_1(x), ..., T_k(x)) \tag{6.10}$$

where we denote the function associating a value and its associated output according to a tree classifier T_i as $T_i(.)$.

In terms of interpretability, this model is somewhere between the decision trees and something completely opaque. It does not translate directly into rules, but it is composed of decision trees which are each interpretable individually.

6.5 Ensemble methods: stacking

Similarly to what is done with random forests, which combine different models with an equal weight built on some part of the learning set, a similar technique can be used with any set of models. Performance could possibly be improved by combining 2 or more of the best performing considered models. [GW19]

To proceed with this, one must first study the performance of different candidate classifiers. Once several best-performing classifiers have been identified, one uses one or more of these classifiers, trained on the entire training set. One of these classifiers is to be used as the "final" classifier, that will yield the output for a given set of input values. The other ("intermediate") classifiers are used to strengthen the efficiency of the final classifier by allowing it to use their respective outputs as an additional feature to use for training and classification. [GW19]

Classification is obtained by first making each intermediate classifier predict the output for the given inputs then finally ask the final classifier to predict the output given the original features and the output of the additional classifier as additional inputs. This is shown on figure 6.2. Training is done in the exact same fashion. Each intermediate classifier is trained on the entire original training set, and the final classifier is trained with the same data to which the output of the intermediate classifiers for the inputs of each element of the dataset. [GW19][NS11]

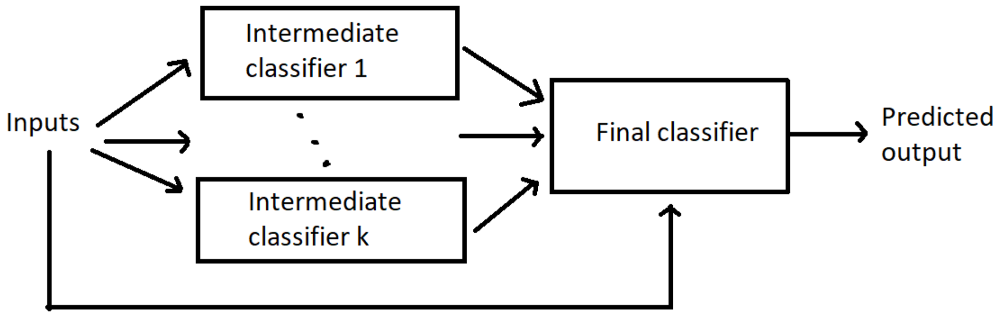

Figure 6.2: Classification by stacking

This simple technique is very powerful as it allows to combine any number of all types of classifiers. Such combination can result in better results than any of the single individual models. It was notably used by the winning team in a competition by Netflix in 2009 to rank shows by user interest.[GW19]

Stacking can be one of the most effective machine learning techniques but its efficiency relies heavily on the combined classifiers. In terms of interpretability, it is almost completely black-box. Interpreting the final classifier, when it is possible, could give some insight on the influence of each intermediate classifier, but if they are themselves black-box, it would be difficult to draw any interesting conclusions.

6.6 Summary of the different considered techniques

Table 6.1 contains a summary of the different considered models and their respective advantages and disadvantages with respect to their possible use for firewall configuration.

At this point, the next logical question to answer is whether decision trees should be used directly to translate captured packets and flows data into rules or if they be used as a rule extraction technique to extract rules from another better-performing model.

This question cannot be answered without evaluating the performance of each of these algorithms for classification and the performance of the derived tree classifier for every technique but the decision trees themselves. With all of these measurements, the efficiency of each technique will be evaluated, and this will also yield an idea of the cost of rule extraction with decision trees.

If some algorithm outperforms decision trees a lot, but yields poor results after rule extraction, the question may remain unanswered. We could then maybe conclude that looking into other rule extraction techniques may be interesting, but there are few general purpose techniques of the sort, i.e. techniques that do not assume any particular underlying model. Most research on rule extraction is actually focused the special of neural networks which are not considered here due to their heavy computational cost, the difficulty of tuning them, and their need for very large datasets that are not always available with real captured traffic. General purpose rule extraction techniques are few, and rarely outperform decision trees much. [HBV07]

However, if decisions trees outperform results of other models prior to tree derivation, it will be obvious that one ought to derive trees directly from the data instead of using a different classifier first. The following chapter will evaluate the performance of the classification for each of the considered scenarios. Whatever the outcome of this performance analysis, the best chosen technique's derived tree can be used to derive rules which can be written using the syntax of a classical rule-based firewall, e.g. a **mmb**-based firewall.

Supervised learning algorithm	Advantages	Disadvantages
Decision trees	- Completely interpretable: translates directly into rules. - Can be used as a rule extraction technique for other models.	- Usually less efficient than other techniques
Naive Bayesian classifier	- Has proven efficient for the similar problem of email filtering. - Probabilities can be interpreted.	- Based of unsatisfied assumptions: no theoretical reason to work - Requires rule extraction techniques
k-nearest neighbors	- Easy and intuitive. - Usually performs well.	- Not particularly interpretable in any meaningful way. - Requires rule extraction techniques.
Random forests	- Usually more efficient than decision trees - Each tree in the forest is completely interpretable	- The forest itself is not really interpretable - Requires rule extraction techniques
Stacking	- Combining several models that perform well can yield a model that performs even better.	- Not interpretable - Requires rule extraction techniques

Table 6.1: Advantages and disadvantages of each supervised machine learning algorithm considered

7 Supervised machine learning: methodology and results

7.1 Model evaluation

The purpose of the performance study is to determine which technique can be used to derive the best-performing model for classification. For this purpose, the need for a performance metric arises. Such a performance metric is the accuracy of the predictions. This is simply the proportion of well-classified data points among the available data set.

This metric is adequate if one wishes to limit both the false positive and false negative rates. Alternatively, for binary classification, any of true positive and true negative rates could be used as a metric in applications where minimizing either false positive or false negative rate is more important than the other. [GW19] [Don19b] In the particular case of traffic filtering, both types of errors can bear a great cost. Accuracy should thus be the metric of choice.

All of these metrics can be derived from the so-called confusion matrix [GW19][Don19b], which is defined as:

$$\begin{pmatrix} TP & FP \\ FN & TN \end{pmatrix} \tag{7.1}$$

where TP is the number of true positives, FP is the number of false positives, FN is the number of false negatives, TN is the number of true negatives.

We thus have:

$$\begin{aligned} \text{Accuracy} &= \frac{TP + TN}{TP + FP + FN + TN} \\ \text{True positive rate} &= TP/(TP + FN) \\ \text{True negative rate} &= TN/(FP + TN) \end{aligned} \tag{7.2}$$

There is however a point to clarify: which data should be used to compute these rates. It is not adequate to train a classifier on the entire available data set and then see how well the derived model predicts that same data. This technique introduces an undesired bias and will cause overfitting. [GW19] [NS11]. Overfitting is a phenomenon where the derived model predicts very well the available data but will perform poorly on unseen data. This phenomenon is very common as supervised machine learning techniques derive all of their knowledge from the available data. [GW19]

To mitigate the problem of overfitting when evaluating the performance of a model, one can simply randomly split the (shuffled) data in 2 groups: the training (or learning) set, and the testing set. The training set is used to train the algorithm, then the derived model is used to predict the output for the testing set. The predicted and real outputs for the testing set are then used to compute an adequate performance metric (e.g. the accuracy). [GW19]

7.2 Model selection

Evaluating the performance of a given classification technique is not enough, one can derive many models using a same classification algorithm and dataset. The algorithms used in supervised machine learning always present one or several parameters whose value can be specified. The chosen values influence the model derivation and can thus lead to models of varying performance. Such parameters are referred to as hyperparameters. [GW19] [NS11]

When trying to tune these hyperparameters, one may think of training models with candidate values with the training set, evaluate their performance on the testing set and keep the better performing one.

This approach is however invalid, as it once again brings bias leading to overfitting of the testing set. Intuitively, it would be very wrong to choose values based on their performance at predicting the testing set, then evaluate the performance of the chosen model with the testing set itself. One must keep in mind that the objective is not the optimize the performance of predictions on known data but rather find the model that will best perform on unknown data and evaluate the capacity of the model to predict this unknown data by using known data. [GW19]

A simple way to avoid introducing such bias, assuming enough data is available is to further subdivide (randomly) the learning set into a smaller learning set and a newer set used for validation. [GW19]

This goes as follows:

1. Randomly subdivide the dataset in 3 parts: learning set (LS), validation set (VS), testing set (TS)
2. Train models on the LS for candidate hyperparameters values,
3. Select the combination of values that yields the best prediction performance on the VS
4. Retrain a model with these hyperparameters with the data from both LS and VS
5. Evaluate the performance of the model on the testing set. This yields the estimated model performance.
6. The final model is obtained by retraining the algorithm with the selected hyperparameter values on the entire dataset LS + VS + TS.

This technique was chosen for selecting and evaluating models for per-packet classification as much more data points are available when considering each packet individually instead of regrouping them into flows.

When less data is available, further subdividing the available learning set is not desirable. Doing so diminishes the statistical power of the data and the quality of the associated predictions from any derived model. An alternative way to select hyperparameter values comes in the form of cross-validation. [GW19]

Let us describe the process known as k-fold cross validation. The learning set is randomly subdivided into k parts. For a given hyperparameter value, performance is evaluated by training the classifier with $k-1$ parts and evaluating the performance on the remaining part.[GW19] This is done for each part and the results are averaged.

In this way, the bias of selecting based on performance of the testing test is avoided while still keeping most of the training set during this evaluation step. Common values for k are 3 and 10. They are known to yield better performance than other values. [GW19]

In the case of a very small dataset, it is interesting to consider $k = n$, which is known as the leave-one-out [NS11] method as only one data point is excluded from the training set at each step.

For model selection of per-flow classification models, 10-fold cross validation was used as it is more adapted to the smaller learning set obtained by grouping packets into flows.

Let us detail the process of model selection and evaluation in this case:

1. Randomly divide the dataset in 2 parts: learning set (LS) and testing set (TS)
2. Randomly divide the LS in 10 equal parts
3. For each set of hyperparameter values considered:
   - For each of the 10 parts, train a model on the remaining 9 parts and evaluate its performance at predicting the remaining part
   - Average the 10 performance metrics
4. Train a model with the entire LS using the best performing set of hyper-parameters values and evaluate its performance at predicting the TS
5. The final model is obtained by using the values found to train a model on the entire dataset LS + TS

7.3 Additional technical details, results, and discussion

7.3.1 Implementation details

The packets and flows data was featurized as described in the previous chapter using a simple custon script written in Python with the scapy library. This library offers native support for packet capture (.pcap) files and could thus be used for going through the list of captured packets and extracting from each packet the desired information.

For per-packet classification, this only consisted in changing each packet into the list of its features. All necessary information was found directly in packets. Time between packets can be computed using the packets' timestamps.

For per-flow classification , as flows were each saved in their own capture file, they can be each treated essentially in the same way as what was done previously. For each packet in the flow, the needed information (timestamps, values of fields...) can be collected directly and any information that needs to be derived from this value (e.g. averages).

The derived data is saved in csv files. Each packet/flow corresponds to a line of the associated features. The data is stored in a different file depending on the associated class (legitimate or malicious).

These files are provided as input for the classification scripts. Classification is done with the algorithms provided in the scikit-learn Python framework. This library offers efficient implementations of each of the algorithms that are considered in this comparative study.

7.3.2 Discussion of the choice of features

Most candidate features that were kept should have a positive influence on the efficiency of the classification. This will be shown to be true later in this section. However, we must never forget the final objective which is configurating a firewall. An opaque classifier is of no interest as no action can be taken to rectify its potential mistakes and it cannot be used in the context of classical rule-based firewall.

Features for per-packet discrimination	Features for per-flow discrimination
Destination address	Pair of communicating ports
Destination port	Transport layer protocol
Source port	Average TCP/UDP checksum
Transport layer protocol	Average time between 2 packets
TCP/UDP checksum	Average payload length
Time since previous packet	Proportion of packets with TCP PSH flag
Payload length	
TCP PSH flag	

Table 7.1: Reminder: chosen features

In the case of per-packet classification, two features that are not really ordinary are the TCP PSH flag and the delay between packets. These 2 features are very descriptive of legitimate and malicious traffic and their differences as was illustrated in the preliminary statistical analysis of traffic performed in a previous chapter. Although, those should reduce the classification error, it is unclear whether they should actually be included.

The inter-packet delay is pretty interesting as it is a reflection of the sending rate but it could lead to some unforeseen issues. For example, in case of congestion on the network, it may vary greatly which would result in this criterion becoming completely unreliable. It is thus only reliable in networks whose use conditions don't vary much, which is a dangerous assumption.

For per-flow classification, the average delay is more robust due to being an average, but it presents the same problem. It is thus better to not actually use this delay for actual configuration, at least in the case of per-packet discrimination. Another important consideration is that this is not usually implemented in firewalls (e.g **mmb**) and bears the additional cost of keeping state of inter-packet time which means saving information for each received packet. This could bear a cost on classification speed, which is extremely important.

Similarly, the use of the TCP PSH flag should improve the classification power as it bears information related to the desired packet buffering in the TCP protocol. Statistics performed in chapter 4 have shown that the control of buffering through the TCP PSH flag was very different depending on the malicious character, or lack thereof, of the considered traffic. However, blocking packets depending on whether or not the TCP PSH flag is true may completely break some applications. If the flag is set, it means that the sender does not want the packet to be buffered, it ought to be sent directly to the application layer.

The sender application is unlikely to change that behavior upon seeing its packets being blocked. It will consider them lost and resend them identically, which will result in packets being dropped until the TCP connection is considered broken. Thus using this flag for packet discrimination would result in all applications using the flag being essentially blocked, which would be problematic to say the least. Even though the flag is susceptible to reduce the classification error, this metric is computed on saved data. Data that would not exist if the connections were broken. It would thus probably be better to not keep this feature for actual firewall configuration. This problem does not exist however when talking about the proportion of PSH flags per flow, which is computed for the entire flow.

Per flow-classification should be somewhat more robust. All packets belonging to a same flow should in principle be classified in a same way. It indeed makes little sense to think of a session between a client and server as partly malicious. This is reflected in the training data in which all communication emitted during a same session is marked in the same way (legitimate/malicious).

However, many proposed features are expressed in the form of averages and proportions. This is not something that is often implemented in firewalls (e.g **mmb**) and keeping track of this additional state could influence classification speed. This presents the advantage of additional robustness, and less risk of breaking connection-oriented (including all TCP) applications by dropping only some of their packets. For this to happen, the metrics considered would need to change greatly over the connection's life, which could happen but is still much less likely.

7.3.3 Hyperparameter tuning

Hyperparameters' values were selected as described in previous section. Only the most important hyperparameter for each algorithm was considered. For building decision trees, whether directly or as a rule extraction technique for other models, this hyperparameter was the depth for post-pruning of the tree. The parameter is the same for random forests (a cutoff depth for each tree in the forest). For the k-nearest neighbors technique, this parameters is the number of neighbors considered.

The stacking classifier uses the best values found in each of the classifiers it is built on top of.

The Bayes filter does not really offer any hyperparameter. In the case where samples can have missing features (e.g. words count in a text), a smoothing parameter can be introduced when computing likelihoods to avoid null probabilities, but it is useless here.

7.3.4 Precisions on the stacking classifier

The stacking classifier uses as final classifier the best performing algorithm and the 2 second best as intermediate classifiers. This ended up being a decision tree as final classifier and k-NN and a random forest classifier as intermediate classifiers.

7.3.5 Discussion of potential feature pre-processing

Some classification algorithms require feature pre-processing. Among those considered, the naive Bayes filter requires it. The algorithm considers features as discrete random variables. If the set of values considered is continuous, they must be discretized in some way. Time delays were discretized by order of magnitudes (ranges of powers of 10). It still bears information about the sending rate, but can be exploited by the algorithm.

Checksums belong to a discrete space, but it is extremely large. A different discretization is thus a good idea for the Bayes filter. For example, they can be replaced by whether or not the checksum is null (0 if null, 1 otherwise). Null checksums mean the field is unused, which is the most important information of the feature. This was shown empirically during the experiments to improve efficiency of each algorithm considered but is essential for the Bayes filter. Such pre-processing will also avoid dropping a connection randomly for having an unusual checksum.

An interesting observation is that with the chosen features as such, decision trees perform extremely well for per-packet classification (accuracy = 0.997) but the best depth found through model validation is very large (= 43). This means a large set of precise rules, which could indicate something went wrong. Looking at the obtained tree, one sees a lot of tests on the source and destination ports.

Hyperparameter	Per-packet classification	Per-flow classification
Decision tree maximum depth	43	7
k-NN #neighbors	1	1
Random forest maximum tree depth	35	8

Table 7.2: Best hyperparameter values for models created without port pre-processing

This is particularly bad. Although source and destination ports can help identify the application, there is no reason they should help classify with that much efficiency. The ports alone mean nothing. This phenomenon can be explained by the fact that the source port (on the client side) is usually random. This means that in the dataset generated with the data from the testbed, each flow will essentially have at least one unique port. Decision trees are thus able to regroup packets into flows with port data. This, however, is not really an actual decision criterion. The source ports are random. They will not be able to identify an almost identical flows on unseen data as they will have changed.

There are a few possibilities to avoid this problem. The first one is to post-prune at a "sub-optimal" depth. However, the first tests in the tree could still be inadequate and without data from the validation stage, it is impossible to know what the adequate depth would be due to overfitting of the testing set.

An alternative would be to stop using port data altogether, but the impact on performance would probably be far too big to consider. Ports are the most common criterion used in firewalls for a good reason. Most common applications use a well-known port on the server-side. Blocking all unused ports is good practice. This is impossible without referring to ports.

The most reasonable solution would be to pre-process the dataset both for packets and flows. Although the problem is not that blatant in per-flow classification, classification based on random non-reproducible data is always wrong. Such a pre-processing could be to only keep values of well-known ports (< 1024) and represent all others with a single value (e.g. 1024) If it is known that a port outside of this range is in use (non randomly) for some application, it is always possible to assign it its own value outside of the excluded range.

Table 7.3 shows that the pre-processing of ports solves the issue by removing entirely the random data.

Hyperparameter	Per-packet classification	Per-flow classification
Decision tree maximum depth	10	8
k-NN #neighbors	4	1
Random forest maximum tree depth	10	13

Table 7.3: Best hyperparameter values for models created with port pre-processing

This ensures that no choice is made based on non-reproducible random data. This allows for correct model validation and selection according to the best practices in machine learning. This is why this option is preferred. Results obtained with such pre-processing will later be contrasted with the results obtained without it. This is a prime example of the pitfalls one must avoid when trying to infer rules from real data. One should avoid introducing non-reproducible characteristics as decision criteria. In network data, one must care for the eventual presence of random data (e.g ports) in the dataset but also for anything that is dependent on network conditions (e.g congestion).

7.3.6 Precision on the k-NN classifier

The metric used is the Euclidean distance as only some features are discrete, using something like the Hamming distance does not make sense. Pre-processing checksums by considering all non-null checksums as 1 allows to avoid nonsensical comparisons. Closer hashes do not indicate closer data.

7.3.7 Results and interpretation

We can see in table 7.2 that there is a clear overfitting of the training set in per-packet classification, especially in the case of the random forests and random trees. The problem is caused by the presence of random ports in the dataset as mentioned earlier. It is not normal to have such depth in trees, this means that the rules are fairly intricate. Only looking at a single neighbor in k-NN is somewhat similar, whatever the closest point in the learning set is for a given input vector, their output should be the same. For k-NN however, this does not necessarily indicate something is wrong, but in conjunction with tree depth, this is an additional reason to look carefully at issues in the produced models. These results essentially indicate that something was wrong in the considered learning, testing and validation sets considered so that the testing and validation data were not really representative of unseen data.

The problem does not appear for per-flow classification but random data is still being used for predictions, which should be avoided. The values of tables 7.2 and 7.3 are obtained without using the inter-packet delay and PSH flag as features for per-packet classification, but the same problem is observed when they are included.

Heatmap 7.1 (next page) represents the estimated accuracy for different depending on the classification algorithm, the approach (i.e. per-packet or per-flow) and whether this is the accuracy of the classifier itself or the rules that are derived from it.

Heatmap 7.1-1 is built with data to which no port pre-processing was applied. To the contrary, heatmap 7.1-2 was built using the same data to which port pre-processing was applied.

Heatmap 7.1-3 is built from data from which PSH flags and inter-packet delays have been removed, then applying port pre-processing.

Heatmap 7.1-3 show that the accuracy when using the data containing random ports is very high. Although this cannot be interpreted in any way as a success, one can see that high accuracy rates can be obtained with such algorithms in circumstances where the data from which the patterns are inferred is sufficiently descriptive for each output class.

One should however keep in mind that firewall configuration is a difficult problem. Attacks often look very much like the usual traffic from the application they are targeting. Simple classification, based essentially on fields in the headers and other simple information from the packet is unlikely to reach such levels

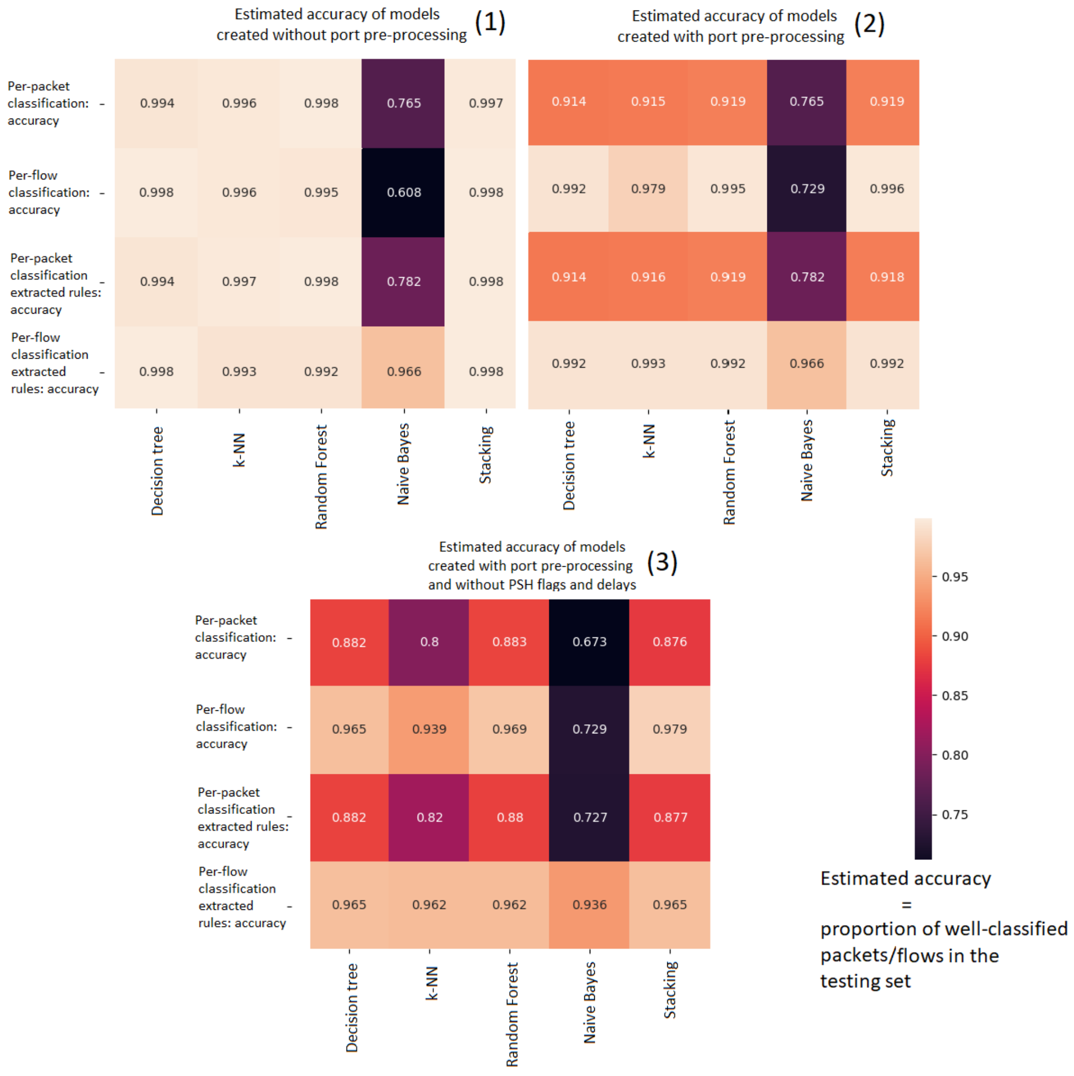

Figure 7.1: Heatmaps of the estimated accuracy of models depending on the data used to build them

of accuracy. What level of accuracy can be achieved in this way is yet to be determined but results obtained here were too optimistic in nature.

Another observation than can be made immediately is that the Naive Bayes algorithm is a poor performer, most likely because of its unsatisfied assumption of independence of features prevents it from detecting the most complicated patterns.

Heatmap 7.1-2 contains the estimated accuracy of models built with all selected candidate features, including the PSH flag and time delay for per-packet classification (and their equivalent in terms of average/proportion for per-flow classification), where port pre-processing has been applied.

The performance with this data which is more realistic and should help better predict what would happen if an actual firewall were configured using this technique. It may however seem counter-intuitive that pre-processing data is the best approach to deriving rules.

First of all, if said (re-)configuration is to happen on-the-fly, then the pre-processing must be fairly simple and have a limited impact on configuration speed. Secondly, and more importantly, once again, one must not forget the objective of configurating a firewall.

A good firewall is a firewall that does not have a lot of impact on the speed of the flows going through it. Ideally, it should operate at line speed. To do so, it does not have much time to compute anything. This real-time requirement results in the fact that most firewalls only accept rules based on data directly available in the packets. If these criterion are derived data, it is something that can be derived quickly.

Thus, pre-processing data may seem like it is contradictory since it will result in rules on derived data rather than what can be found directly in the packet. However, all pre-processing that was considered is fairly simple (and in particular the very important port pre-processing that avoids making choices based on random data). It is thus easy to reverse such processing directly in the deduced rules to produce equivalent rules that will classify traffic in the exact same way that the original rules but only refer to data that is readily available in packets. This will be one of the main configuration issues tackled in the next chapter dedicated to concrete firewall configuration. The computing cost of this preprocessing is also acceptable and will not hinder configuration speed as it is linear with the size of the dataset that must already be considered entirely at least once, and in practice several times, by machine learning algorithms.

One must however be careful, packets and flows can only be classified correctly if the firewall does not interfere with established connections. The use of PSH flags and delays in per-packet classification will definitely do so, as explained earlier.

A similar but less obvious phenomenon could affect classification per-flow. The features considered mostly come in the form of averages and proportions. These do not present an obvious implementation as discrimination criterion and give rise to many difficult questions to answer for concrete actual firewall configuration. How are these averages and proportions computed, over which amount of packets?

If an attacker sends a single packet containing a malicious payload and the average is computed over 10 of these, then can the decision to end the connection be done before the malicious packet reaches its destination? If the protocol is connection-oriented, a firewall cannot wait for 10 packets to arrive as they most likely will not ever be sent if no response is received.

Criteria that are problematic from this point of view in the case of per-packet classification are to a lesser degree also problematic with a per-flow approach. For example, the PSH flag and inter-packet delay could cause such problems in the context of per-packet classification. It is less likely that considering proportions of packets marked with PSH flags over a few packets leads to suddenly dropping the connection, but it cannot be totally excluded. The average delay over a few packets of the flow is more stable than a simple inter-packet delay, but if unusual congestion is observed over the entire duration over which this average is computed, the exact same problem will occur.

Heatmap 7.1-3 shows what performance can be attained when omitting these 2 criteria from our features. Ports and checksum pre-processing are in use. In these conditions, no connection should be suddenly interrupted.

What can we learn from tables heatmaps 7.1-2 and 7.1-3 ? First, they show that both per-packet and per-flow classifications are actually viable. There are actually convincing reasons to believe that machine learning-based classification can be applied successfully for the problem of firewall configuration.

Another somewhat surprising observation is the impressive efficiency of decisions trees. Decision trees yield the simplest and most interpretable models since they translate directly into rules but they are among the most efficient models considered here. Only the random forest and stacking classifier have such high accuracy and they both incorporate decision trees in their classification process. This makes decision trees very efficient when used as an extraction technique for the other models.

However, the use of decision trees as an extraction technique, although efficient, seems superfluous compared to building a decision tree directly from data. Effectively, using another classifier prior to building a decision tree only serves to eliminate some data deemed unrepresentative of the first model. If the first model were much more efficient than the decision tree, it might help with its accuracy, but in conditions where decision trees perform better or as well as the original model, it makes little sense.

Nevertheless, using one technique or the other makes little difference. Predictions are made with any classifier then you extract a tree from it (unless it already is a tree).

Using models built with decision trees, whether directly, or as an extraction technique for some other classifier, is the exact same in terms of outcome. Both yield a tree to translate into rules. In terms of accuracy of the predictions, both have shown to be roughly equivalent.

Thus, it does not matter too much if one does one or the other, but since decision trees are very fast both for training and predictions, and since the same is not necessarily true for all algorithms considered (e.g. random forests), building decision trees directly from the data seems to be the most adequate course of action.

As was discussed earlier, heatmap 7.1-3 is the most representative of the accuracy that could be reached using this type of automatic configuration. Apart from what was already discussed the most important information that can be extracted from this table is that the estimated accuracy of per-packet classification would be 88% while per-flow classification is estimated to reach 96% accuracy.

However, one must note that these numbers do not tell the whole story. As explained, the criteria used for per-flow classification are based on averages and proportions. When predicting on the considered dataset, these averages were pre-computed using the entire flow. In practice, this is impossible to do for a firewall, that will need to compute such values based on a subset of packets. This is worrying for several aspects, e.g. stability, reproducibility, and loss of accuracy.

Using a per-flow classifier for firewall configuration would likely require many modifications to an existing firewall to incorporate discrimination based on moving averages (or finding different criteria that bring similar efficiency). One would also need to deal with the imprecision brought by those moving averages (sudden need to close a connection, the issue of the first packets needing to go through and its potentially terrible consequences).

As such, to recommend a per-flow approach rather than a per-packet approach, it would be necessary for the per-flow approach to perform very significantly better than the per-packet approach. Let us look at the results we got in heatmap 7.1-3 and try to draw a conclusion.

With accuracy as our efficiency measure, let us consider the error rate.

$$\text{error rate} = 1 - \text{accuracy} \tag{7.3}$$

The error rate corresponds to the proportion of packets (resp. flows) that are misclassified. Heatmap 7.1-3 gives an error rate of roughly 12% at best for per-packet classification and at best an error rate of roughly 4% for per-flow classification. Considering both figures as comparable, one would conclude that per-packet classification makes 3 times more errors than per-flow classification.

This would however be unfair as the estimate for per-flow classification performance does not account for the very important issues concerning the feasibility of implementation. Both figures can be seen as an upper bound on performance, but the figure concerning per-packet classification should be a tighter one.

Let us state an obvious fact. Both figures are too high. Allowing 4% of attacks in is too much, and blocking 4% of legitimate traffic also is. When both can have terrible financial consequences, one wants to avoid these errors as much as possible. What can one do confronted with such a fact?

A possibility would be to give up completely on machine learning-based classification for firewall configuration. This is obviously the most uncharitable interpretation of these results.

A more reasonable outlook at the situation would be to argue these results could still be improved. Machine learning is an ever evolving field from which the experiments performed only scratched the surface. Neural networks were not really investigated due to their complexity and their heavy computational requirements, but if one were to look into neural networks, many research is oriented towards translating such networks into rules. [HBV07]

Maybe another featurization could be considered since it would seem that performance here is mostly limited by the expressivity of rules expressed with such features as the contrast between heatmaps 7.1-1 and 7.1-2 illustrates.

Modern firewalls may perform deep packet inspection. This is much more complicated than what was considered in this performance study but if one were to extract information related to the application layer, this information may be used to create very expressive features to describe traffic.

It would be wrong to look at the error rates and conclude that they are too high for the techniques developed here to have any practical interest. It is true that as such, the imprecision of traffic discrimination of the configured firewall would be too high to rely on. Nevertheless, it is a good starting point. One may generate such rules from known legitimate application traffic and previous attacks and then check into a test environment if the filtering is adequate.

If some rule is too strict or too lenient, the system administrator may change it accordingly. This is not ideal, true automatic configuration should be done without trade-offs and human intervention, but this would require unreasonably low error rates that most likely cannot be attained without more advanced techniques.

If this is not a reasonable fully automatic firewall configuration method, it is almost certainly already a decent technique to diminish the burden of firewall configuration. From desired allowed and denied traffic examples, an administrator can easily generate a reasonable amount of rules. The best tree depth found was 10.

The number of rules extracted from a binary decision tree (e.g built by scikit-learn) is upper-bounded by the number of leaves in the tree. Since the whole packet space is covered by the decision tree, one only needs to keep all rules regarding either accepted or blocked packets (resp. flows). The number of leaves is itself bounded with respect to the tree's depth. For a full binary tree, the number of leaves is related to the depth of the tree by this formula:

$$\#\text{leaves} = 2^{\text{depth}} - 1 \tag{7.4}$$

This is the upper bound on the number of leaves. This means that for a decision tree of depth 10, the upper bound on the number of rules is 1023. In practice, since only roughly half of them need to be kept and since decision trees aren't usually full, the actual number would be lower than this. A tree of depth 10 generated from the per-packet data considered for the experiment has 171 leaves, 101 of which are ACCEPT rules. Depending on the approach chosen, this would yield either 101 or 70 rules used to configure the firewall.

The number of rules considered is in practice relatively easy to review exhaustively for a system administrator looking for a particular issue. There is no doubt that if complete automation is not yet attained, the proposed techniques can at least be used to heavily simplify the configuration task.

7.3.8 Considerations on replies in training data

In everything that was done previously, the training data contained both requests and replies. It makes sense to consider the reply to a malicious request as malicious traffic since it could lead to sending information to an attacker.

However, detecting the reply to a malicious request is more difficult than detecting the malicious request itself. A legitimate application will send very similar responses to most requests, including those originated from attacks.

When considering a per-flow approach, replies must be considered, since the entire concept of using flow-related data is to view traffic as exchanges between two communicating hosts. Replies are thus important.

In contrast, in the per-packet approach, the presence of replies in the dataset is more questionable. Replies to malicious requests are harder to detect that the requests themselves. Since they are very similar to replies to legitimate requests, they do introduce a form of noise that could increase the error rate. Additionally, working only with requests to build the rules cannot result in any new trade-off. Indeed, it is obvious that no response can be sent to a request that was not received. Thus, blocking all malicious requests is enough to also block all malicious replies by preventing them from ever existing.

Thus, it is interesting to check if the intuition that only working with requests in the training data actually reduces the error rate. The per-packet classification error rate is of 12% when considering all traffic. Only considering requests, which is easy to do since one host acted as a server while the other acted as client/attacker, the error rate drops to 9% (for decision trees).

While this is not a panacea, it would seem that not considering replies helps the accuracy of the predictions, and by extension the derived rules. One should however note that separating requests and replies is not always obvious if all communicating prefixes contain servers and clients. However, in the common case of protecting a data-center from incoming attacks, it makes sense to only consider incoming traffic for training in the case of per-packet classification.

7.3.9 Conclusion on the comparative study and outlook

What we can conclude from this performance analysis is that automatic firewall configuration is actually possible. Decision trees are an efficient and simple way to derive rules from featurized packet and flow data. They work well in comparison with other classification algorithms and can also be used in conjunction with them as a rule extraction technique. This was however not found to be significantly better than building trees directly.

Per-flow classification was found to be somewhat more efficient than per-packet classification on the data considered. However, both are not directly comparable as per-packet classification can be implemented without much additional concerns given a firewall can be configured with rules using combinations of equality and inequality constraints on the considered features while per-flow configurations would require building firewalls keeping track of moving averages which present many challenges and would be very sensitive to the considered "packet-window" on which the average would be computed.

However, due to the many problems of this approach (first packets always go through, difficulty of choosing a window...), it is unlikely this could be used as such in practice. Maybe some hybrid approach could be considered. Per-packet classification could for example be enabled in conjunction with per-flow classification with moving averages. This would avoid allowing the first packets always going through while still taking advantage of flow data.

Considering all of this, a per-packet approach seems to be the way to go. Nevertheless, given a firewall that can handle rules based on moving averages, both approaches would essentially be equivalent. How the actual configured firewall would handle incoming traffic would of course be very different, but configuration itself would be almost identical. The decision tree would be translated into conjunctions of equality and inequality constraints and the associated action, which is equivalent to the output of the classifier for the given conjunction. For combinations considered malicious be the classifier, the associated action would be DROP, otherwise ACCEPT.

Then, the last step would be to translate these derived rules into rules that are syntactically valid for the firewall of interest. None of these techniques are inherently dependent on the type of classification (per-packet / per-flow) chosen. This technique is actually very generic and could essentially be used to consider any firewall given the firewall actually allows sufficient expressivity for writing conjunctions of inequality and equality constraints on the chosen features.

The method is also totally independent from the actual choice of features. If one wanted to configure some advanced firewall based on deep-packet inspection, the technique can be used unchanged given one can incorporate application-layer data into the featurization.

Considering that the per-flow approach presents too many problems to be implemented as such, while the simpler per-packet approach can be, and also taking into account that the per-packet approach is the first step towards hybrid approaches, it only seems natural to put the emphasis on per-packet classification.

The remainder of this thesis will be devoted to deriving rules from the per-packet decision trees and using them for configurating the **mmb**-based firewall.

8 Practical firewall configuration

8.1 Structure of the binary decision tree

The approach chosen for rule derivation is to use decision trees. The tree can be built directly or with the help of another opaque classifier, but the output is the exact same: a binary decision tree.

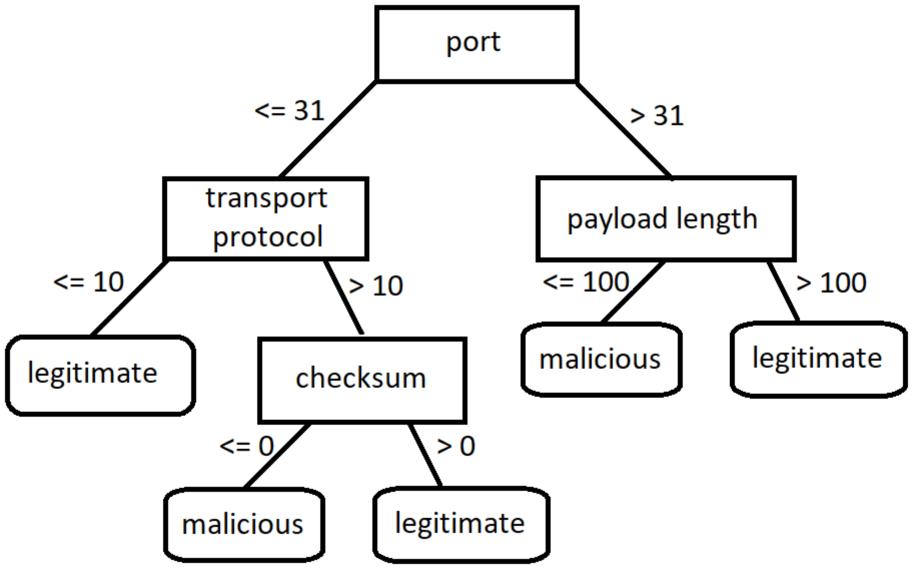

Figure 8.1: Example of a binary decision tree for packets

Let us discuss the structure of binary decision trees. To each non-terminal node is associated an attribute (feature) and a given threshold for said attribute. The left child is a tree containing all data points (of the considered learning set) for which the given attribute is lesser or equal to the given threshold, while the right tree contains all points for which the attribute is greater than the threshold. The leaves contain the class that is associated to each point that follows the path the root to the leaf, i.e. points of the learning set that meet all of the constraints of the branches on that path.

The training set used to build the tree is considered representative of the entire state space. New data is classified exactly like the data used to build the tree. A given data point is classified by placing it at the root of the tree and then following the path determined by the point's attributes values. It eventually reaches a leaf which is labelled with the class that is assigned to that given point.

8.2 Rule extraction from the tree

Each leaf effectively determines a rule. The form of any of these rules is as follows:

$$\text{constraint}_1 \ \wedge \ \text{constraint}_2 \ \wedge \ \ldots \ \wedge \ \text{constraint}_k \implies \text{action} \tag{8.1}$$

The constraints in the conjunction are equality and inequality constraints on the path from the given leaf to the root of the tree and the action is either accept or drop the packet depending on the leaf value (respectively legitimate or malicious).

Such a form is typical in firewall configuration in which it is common to describe all characteristics of a given packet and what the associated action should be.

In firewalls, however, it is typical to have priorities in the rules. If several rules apply, then the first one in the list of rules is the one that has precedence over the other.

This does not really matter here as the rules really never overlap. As rules correspond to the constraints on the path from a given leaf to the root, they at least have one contradictory constraint when compared to another rule in the rule set. If they did not, that would mean that both compared rules correspond to a same path in the tree (i.e. conjunction of constraints) and thus they could not originate from different leaves.

The following pseudocode describes an algorithm for extracting the complete set of rules from a binary decision tree.

```
function get_rules(tree)

        rules = RuleSet()

        for leaf in tree.leaves

            rule = Rule(action = leaf.class)
            node = leaf
            parent = leaf.parent

            while parent is not None

                isLeftChild = True

                if parent.rightChild == node
                    isLeftChild = False

                if isLeftChild
                    rule.addConstraint(
                                    attribute = parent.attribute,
                                    type ="<=",
                                    threshold = parent.threshold)
                else
                    rule.addConstraint(
                                    attribute = parent.attribute,
                                    type =">",
                                    threshold = parent.threshold)
                node = parent
                parent = node.parent

            rules.addRule(rule)

        return rules
```

Once the rules recovered, they can be used for configurating an actual firewall. However, this presents some additional challenges. First of all, these rules are not yet written in the actual syntax of the given firewall of interest. They need to be translated into something that is valid in that context.

This could be as simple as replacing the features' names by their correct name in the context of a given firewall's syntax, printing things in the right order, and adding a few required keywords. However, there can be additional considerations to take into account before being able to generate the rules in such a simple fashion.

The features that were chosen could for example not be entirely expressible with the grammar of the firewall of interest. In such case, there is a need for an additional translation into criteria that do exist in that context.

Maybe some feature does exist as an available attribute in the firewall's grammar but only equality constraints are allowed while inequality comparisons are not possible.

In the previous chapter, the benefits of pre-processing training data, in particular regarding ports, were discussed. From such pre-processing can arise new challenges. If one builds a tree based on features that are pre-processed, then the rule apply to pre-processed data and not to the original data. A strategy to deal with rules from derived data is thus necessary. Two scenarios are possible.

The first possibility is to use the rule as such and the firewall must perform itself the necessary derivations on incoming traffic to apply the rule on the derived data. This is possible but requires the firewall to offer the functionality to write rules on such derived data, which is rarely the case.

One could modify the given firewall to incorporate said criteria but this presents limitations. This approach is complicated as it requires modifying the firewall software itself, and it may not always be possible. If one wants to configure a firewall for which the source code is unavailable or it is illegal to modify, it will not be an option. The derivations that have to be performed must also be quick enough to avoid introducing new bottlenecks that would reduce the operating speed of the firewall.

The other possibility is to translate the rule back into one or several rules that are actually expressible using the criteria that are already available in the firewall's grammar. This approach does not require any modification of the underlying firewall software.

It bears the limitation of needing to be able to reverse the rules to apply to the original data, which could be non-obvious or even impossible with a complex pre-processing. If however, this is possible, this probably is the best bet as it is almost certainly easier that modifying the firewall itself, and the firewall software's optimization to deal with the criterion that it offers is taken advantage of.

There are still however a few obvious drawbacks to this approach. The type of rules with constraints on pre-processed data from which rules on original data can be derived are a smaller subset of possibilities. This is effectively a limitation of the expressivity of the feature space.

Another potential problem could come in the form of unreasonably increasing the number of rules. If one rule on pre-processed data can be expressed with a much larger number of rules on the original values, then number of all rules using that criterion will be multiplied by that large constant.

Once all of these additional considerations addressed, configuring the firewall truly consists in simply writing all of the rules so that they follow a valid syntax for the firewall of interest. This is straightforward but non-generic. Even given two firewalls that allow writing rules as inequality constraints with the exact same attributes, if their syntax differ, one will need to respect it.

The remainder of this chapter will be devoted to to configuring a **mmb**-based firewall while dealing with the pre-processing that was described in previous chapter.

8.3 Configurating mmb with a decision tree

8.3.1 mmb rule format

What is needed is to be able to express a conjunction of equality and inequality constraints and associate with an action: ACCEPT or DROP.

With **mmb**, this can be done easily with the following syntax [Ede+19]:

To add a rule and associate with an action, one uses the command

- mmb <add-keyword> <match> [<match> ...] <target> [<target> ...]

Let us detail this form. The rule consists of a single <add-keyword>, one or several <match> elements, which correspond to the constraints and one or several <target> elements which correspond to the action to be taken upon matching.

The <add-keyword> element indicates whether the rules are to be applied on a per-packet (add, add-stateless) or on per-flow basis (add-stateful). [Ede+19]

- <add-keyword> = add | add-stateless | add-stateful

As **mmb** is not merely a firewall but also a generic middlebox configuration tool, many values are possible for <target>. In this context however, the only relevant possible values are *accept* and *drop*.

The <match> element has the following definition.

- <match> = <field> [[<cond>] <value>]

Each <match> item corresponds to a comparison between a field and a possible value. <cond> can be either "<", "<=", ">", ">=", "==" in most cases. In specific cases, where inequality comparison is uncommon because it makes relatively little sense, only "==" is available. This is the case for IP addresses and prefixes for which only equality can be checked.

8.3.2 Challenges in configurating mmb from a decision tree and their solutions

An algorithm was described to extract a set of rules. Each rule's individual form is a conjunction of equality and inequality constraints associated with an action. This format is very close to what is required by **mmb**'s syntax. Let us discuss all difficulties in going from the former to the latter.

Features and corresponding mmb attributes

The main difficulty in configurating a firewall using **mmb** from a decision tree comes in the form of being able to express the rules that have been extracted from the tree using only the attributes that are readily available in **mmb**'s grammar and the comparison operations that apply to them.

Luckily, we can express rules in a format that is close to what can be extracted from a decision tree, as we can associate an action and associate with a combination of equality and inequality constraints. The potential for issues comes first from the potential pre-processing and secondly from the possibility of missing attributes or inequality comparison operators on these attributes.

Feature	**mmb** attribute	Inequality available?
Transport protocol	ip-proto	Yes
Destination address	ip-daddr	No
Destination port	tcp-dport udp-dport	Yes
Source port	tcp-sport udp-sport	Yes
TCP/UDP checksum	tcp-checksum udp-checksum	Yes
Payload length	None	N/A

Table 8.1: Features and corresponding **mmb** attributes

Table 8.1 lists the different features, their **mmb** equivalent attribute if there is any, and whether or not inequality comparison are possible (equality always is). We will only consider the features of the final model of per-packet classification that was discussed in chapter 7.

It is easy to see that, although most attributes can be used as such (after considering a few caveats relative to pre-processing), there are still a few features that will require some additional translation of the rules to be expressible with **mmb**'s attributes.

Each necessary transformation of the rule set will now be addressed, no matter whether if it is due to the reversal of some pre-processing or due to the limitations of **mmb**'s expressivity.

IP destination address

IP addresses are treated by the decision tree algorithm as if they were integer numbers. While this makes perfect sense, as IP(v4) addresses are technically 32-bit integers, it is rare to think of an IP address is being smaller or bigger than another. (The same reasoning remains valid with IPv6.)

That sort of comparison is never really made because there are no many situations where this would be meaningful. However, the only way that the binary decision tree algorithm reduces the state space is by introducing inequality constraints.

What is commonly done with IP addresses, is regrouping them into meaningful prefixes. A given range of IP addresses can be expressed by a certain prefix (its first bits). Fixing the first bits of an IP address to a certain value is equivalent to setting a lower and upper bound on that number. With sufficient tests on the address, the decision tree algorithm could attain isolate a given prefix. Such comparisons are thus not meaningless, just fairly uncommon, as specifying the prefix is much more natural when manually dealing with IP prefixes.

If modifying the firewall code is possible, then such comparisons can be introduced and no other action is required. This is however not the approach that is chosen here.

It is always possible to enumerate all IP addresses (or prefixes) in the considered dataset that are actually encompassed by a given rule and creating as many rules as there are addresses/prefixes and checking for equality, leaving all other constraints on other attributes and the associated action unchanged.

This is obviously equivalent if and only if the training data set actually encompasses all possible values. In the dataset created on the testbed, only two addresses were possible, in which case the solution is obvious.

However, even with a dataset containing many IP addresses, this solution is also possible. It mostly relies on a choice of pre-processing. For the results to be meaningful and the number of rules to remain reasonably low, there must not be too many possibilities.

One could imagine regrouping all external addresses under a single value. Internal addresses could also be regrouped into meaningful prefixes (e.g. the different departments of a company's network). In this way, the number of prefixes/addresses to enumerate remains low and the rules that are created bear an obvious meaning about the origin of the traffic they apply to, which is good for interpretability.

Decision tree: continuous values in the rule set

A small problem with the binary decision tree algorithm, as implemented in the Scikit-learn library, is that it consider all features as having continuous values. This means that the values of the thresholds in the extracted constraints may sometimes have a fractional parts. While it is mathematically valid to define an integer in comparison with a real-valued constant, it is not something that anyone would expect to be implemented in a firewall's grammar.

Two forms of constraints are possible:

1. attribute $\leq t$
2. attribute $> t$

For both type of constraints, the integer points that satisfy them are exactly identical if the threshold t is rounded down. They can thus be rewritten.

1. attribute $\leq \lfloor t \rfloor$
2. attribute $> \lfloor t \rfloor$

Rounding all of the constraints' thresholds does not change the semantics of the rules but it is a necessary operation for configurating firewalls that do not handle real values (e.g **mmb**).

Multiple tests of the same attributes

There is no real problem with having several constraints on a same attribute. If a feature is very expressive, it is possible that most constraints in a given extracted rule concern the corresponding attribute. This is a direct consequence of the structure of a decision tree.

However, it means that some rules can be simplified. Let us consider the following set of constraints.

- {ip-proto == 6, tcp-dport $<$ 50, tcp-sport $\leq$ 25, tcp-sport $\leq$ 23, tcp-sport $>$ 10}

It is easily observed that one constraint can be removed while still describing the same set of packets.

- {ip-proto == 6, tcp-dport $<$ 50, tcp-sport $\leq$ 23, tcp-sport $>$ 10}

This can be done because if more than one upper or lower bound is imposed onto a given variable, then it is perfectly equivalent to only keep the stricter one.

Each rule can thus be simplified by removing all redundant constraints. For each attribute in the rule's constraint set, two bounds must be found: the maximal lower bound and the minimal upper bound. All constraints involving other bounds are redundant and can be omitted from the constraint set.

Rules involving constraints on the transport layer protocol

Depending on the transport layer protocol in use, the traffic has a very different form and characteristics. It is natural that the transport layer protocol appears in most extracted rules' constraints sets as they are one of the most basic essential features of traffic. For example, to allow HTTP traffic on a simple firewall, one allows TCP on port 80.

There are 2 main transport layer protocols: TCP (ip-proto=6) and UDP (ip-proto=17). Those are the only values for ip-proto considered in the dataset generated on the testbed.

It is entirely valid, and accepted by **mmb**, to have a constraint of the forms {ip-proto $\leq$ t} or {ip-proto $>$ t}. However, this is inconvenient, as it includes a large set of protocols which are not really relevant for configurating a firewall.

Application traffic is generally either TCP or UDP and if something else is needed, this typically for a specific reason that requires some manual configuration (e.g. setting up an IP in IP tunnel, allowing the machine to be pinged through ICMP). Such scenarios are dealt with manually by a network administrator.

As ip-proto values other than TCP and UDP are not relevant to automatic firewall configuration, they can simply all be dropped with the exception of the ones that are manually permitted.

The constraints on ip-proto in the rules that are extracted from the decision tree are however inequality constraints. Each rule can only contain at most one of these due to the nature of the decision tree algorithm. Splitting the training set more than once using this criterion would not improve classification, so it is never done.

When present, the constraint regarding ip-proto for a given rule will only be satisfied by either TCP (ip-proto = 6) or UDP (ip-proto = 17). It can thus be replaced by an equality constraint with the one of these 2 values that satisfy the inequality constraint.

This presents only positive consequences as this excludes unwanted protocols from the extracted rules. The decision tree could not possibly produce the same result as it only produces inequality constraints and no example of these other values were present in the dataset. The relevant traffic remains classified in the same way.

Rules wihout constraint on the transport layer protocol

It is entirely possible that some rules extracted from the decision tree present no constraint on the transport layer protocol. However, most features (e.g ports, checksum, payload length) correspond to a different field in the packet depending on whether TCP or UDP is used.

While it is acceptable to treat those in the same way for machine learning, the firewall needs to know what value it is testing against the constraints of a given rule.

A simple solution comes in the form of replacing each rule that does not contain a constraint on the network protocols by two rules: one only for TCP and one for UDP. The new rules have exactly the same constraints with the addition of either {ip-proto = 7} or {ip-proto = 16}.

Specialization of features into actual attributes

The extracted constraints are independent of whether TCP or UDP is used as a transport layer protocol. However, as explained above, to configure **mmb**, they need to be specific to either TCP or UDP. Due to the modifications discussed above, all rules now have a constraint on the transport layer protocol: either {ip-proto = 7} or {ip-proto = 16}.

Each attribute can thus simply be replaced by its specialized version (e.g checksum becomes tcp-checksum or udp-checksum depending on the value of the constraint on ip-proto).

Pre-processing of checksums

Processing checksums prior to using them for machine learning removes nonsensical comparisons between them. Closer checksums does not give any indication that the packets are actually closer in content. The pre-processing that was proposed consists in representing non-zero checksums by 1, and 0-checksums by 0. This was shown to improve the classification accuracy.

However, the rules derived apply on pre-processed checksums, not checksums themselves. Let's go over what could potentially go wrong and how this can be dealt with.

Four possible constraints are possible (after rounding).

Three of them are {checksum $\leq$ 0}, {checksum > 0}, {checksum > 1}. The first two pose no problem whatsoever because they have the exact same meaning whether the checksums are pre-processed or not.

It should be noted that {checksum > 1} should not appear as a constraint since it would be a bad split for the tree since no (pre-processed) training data respects that constraint. For this exact reason, any rule with this constraint can be discarded as it cannot describe any valid pre-processed packet. This means that the constraint is meaningless as it applies to traffic that cannot exist

Similarly, the last possible constraint {checksum $\leq$ 1} should not appear either. It is a bad split as it is complementary to {checksum > 1} (all packets satisfy this constraint). This means that at a given node in the tree checking for this threshold for the checksum, all training data would go to the left son and nothing to the right son.

If the constraint were to be chosen anyway, for example by deriving similar rules on data that was pre-processed in the same way with an other algorithm than binary decision trees, then it is not really difficult to deal with.

The constraint {checksum $\leq$ 1} means that the checksum is either 0 or greater (because of the pre-processing of anything non-null into 1), that is {checksum $\geq$ 0}. The constraint can thus be rewritten in this way or removed entirely. This effectively means that this is valid independently of the checksum, which is exactly why this would be a bad split.

In practice, this means that rules based on pre-processed checksums can be used without (or almost without) any modifications.

Pre-processing of ports

Port pre-processing was shown to be important to ensure the validity of the predicted models and associated rules. Without pre-processing port data, random ports are present in the training dataset.

This phenomenon is due to the fact that in a communication between a client and a server, the client's sending port is generally random while the port that server is listening to is generally a well-known standardized port.

If a model is built from data containing such random ports, it is likely that the exact same packets sent from a different random port would not be treated in the same fashion. Since this is highly undesirable to introduce such inaccuracies of unknown impact, a simple solution was proposed, in the form of simple port pre-processing.

The simplest form of such a processing is to assimilate all non-standard port values into a single value. For example, all non-standard ports ($\geq$ 1024) are represented by the single value 1024.

In this way, all non-meaningful random data is eliminated from the training dataset in an interpretable fashion. All standard ports are used for machine learning while randomly generated ports are considered equivalent in the absence of additional information.

Once again, similarly to how the checksum pre-processing influenced the semantics of the constraints on checksums, this also modifies how the constraints on port are to be interpreted. As always with constraints extracted from a binary decision tree, two forms are possible.

1. port $\leq t$
2. port $> t$

The possible thresholds t always lie within the range of values that are found in the training dataset. As the tree is built on pre-processed data, this means that we strictly have $t \in [0, 1024]$. Since all thresholds that represent integer numbers are to be rounded down, we only need to consider all $t \in \{0, ..., 1024\}$.

The good news is that it is very easy to modify the rules so that they apply to unprocessed ports. One must first notice that for $t\ in$ $\{0, ..., 1023\}$, the meaning is unchanged for both types of rules.

It is worth noting that, similarly to the observations made for checksums, using 1024 as threshold would be a bad split and should never happen by using a binary decision tree algorithm. All (pre-processed) data points would satisfy the constraint of type 1 (port ≤ 1024) while none would satisfy its counterpart (port > 1024).

However, if some other rule extraction algorithm were to derive such constraints from data that was pre-preprocessed in this way, then it can also be easily dealt with.

The possible constraint (on pre-processed data) {port > 1024} describes a class of traffic that does not exist, any rule involving it can thus be simply discarded.

The complementary constraint {port ≤ 1024} matches all traffic and can thus be replace by {port ≥ 0} or simply removed from any rule it appears in.

This simple pre-processing can thus be used with little to no modification of the derived rules. However, one may argue that although this pre-processing allows to derive rules from data with random ports, it may lose some information about non-random use of non-standard ports.

A solution was proposed to this problem. If such non-standard ports are in use for some application, then it is known to the system administrator that would be in charge of the self-configured firewall. They can be excluded from the pre-processing of ports and thus have a distinct value each.

This is a little more complicated than the simpler variant, but few modifications of the derived rules are also necessary. If a rule contains constraints such as {port $> t_1$, port $\leq t_2$}, then it must be replaced with one or several rules corresponding to which non pre-processed ports this interval actually represents.

What this means is dependent of the actual non-standard ports that were not assimilated to 1024 and of the thresholds t_1 and t_2 but the modifications to the rule set remain fairly few and simple.

Payload length

It was shown that constraints on any field that is found in packet headers, and thus consequently in **mmb**, can be expressed as valid packet constraints, even if some of the corresponding features are pre-processed prior to feeding them as input to a machine learning algorithm.

However, a feature that is of great interest is the TCP/UDP payload length. This feature is not something that is found directly in packet headers. Its interest resides in the fact that it is one of the few low-level and simple pieces of information related to the application layer. No matter whether if it is legitimate or not, traffic is usually sent to an application.

Attacks targeting an application may have a typical behaviour at any networking layer below the application. Their only difference could consist in the form of the application payload. Do they contain typical data for the application, too much of it to cause a buffer overflow attack, or is maybe the request malformed in some way to make the receiving application crash? All of these are possible and could be invisible at the transport layer.

Payload length is a very descriptive low-level feature about the application layer. It is not directly a field in packet headers but it can be derived from data that is found in such headers. It likely would not be very difficult to allow writing rule with this criterion by modifying the firewall's code and grammar.

This is not the approach that was chosen here but what follows will make clear how this could be achieved anyway. Computing this feature is very much dependent on the actual transport protocol. The distinction between TCP and UDP traffic is primordial here.

The key principle is that the payload starts where the TCP/UDP header ends. Where exactly this header ends is and the length of what remains is information that can be derived from information that is is readily available in IP, TCP and UDP headers.

Payload length: UDP packets

Let us start with the case of UDP packets, as it is the simpler of the two cases. The reason it is simpler is two-fold, the UDP header has an explicit length field and its header's size is fixed.

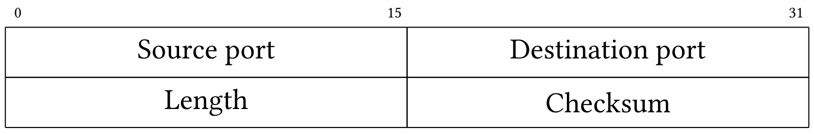

Figure 8.2: UDP header

The UDP header can be observed on figure 8.2. We can see that the UDP header is made of 4 fields of 2 bytes (16 bits) each. A field which is of great interest is the length field.

This field is not equivalent to the payload length. It corresponds to the length of the entire UDP datagram including the UDP header.

This means that the payload length can be expressed directly as a function of this field. More precisely we can write:

$$\text{payload-length} = \text{udp-length} - 8 \tag{8.2}$$

This means that for constraints of the form,

- payload-length $\leq t$
- payload-length $> t$

we can write equivalently:

- udp-length $\leq t + 8$
- udp-length $> t + 8$

This is a simple transformation of the rules extracted from the tree into equivalent rules using only attributes that are available in packet headers and consequently in **mmb**. There is thus no real drawback to the approach of keeping the firewall unchanged for rules about UDP traffic.

Payload length: TCP packets

The case of TCP traffic is more complicated. TCP's header size is variable, so it is impossible to treat it as a constant like it was done for UDP. Furthermore, the TCP header does not have any length field comparable to the one found in UDP's header.

To be able to express TCP's payload length, we will need information from both TCP and IP headers. Both headers are of variable length, which complicates the matter. They are displayed next page (figures 8.3 and 8.4).

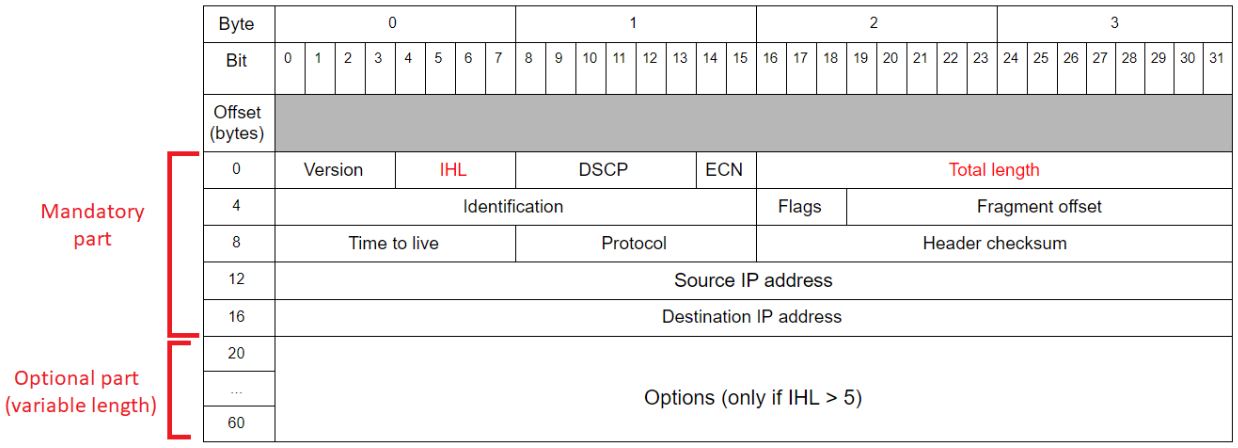

Figure 8.3: IP(v4) header, inspired and adapted from the figure in https://en.wikipedia.org/wiki/IPv4 in the header structure section

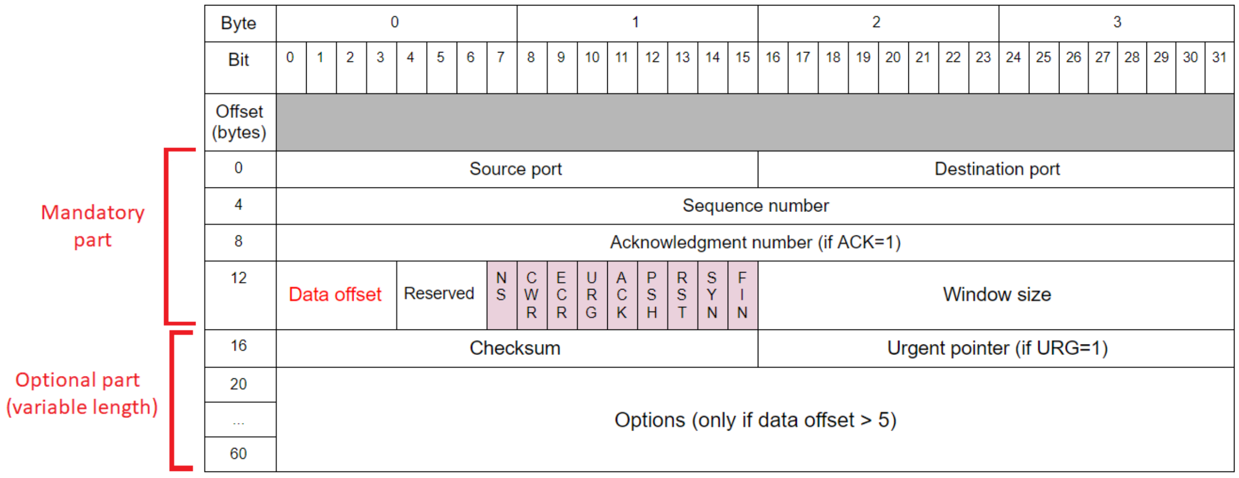

Figure 8.4: TCP header, inspired and adapted from the figure in https://en.wikipedia.org/wiki/Transmission_Control_Protocol in the TCP segment structure section

Figures 8.3 and 8.4 illustrate how the content of these 2 headers can be used to derive the length of the payload that follows. Three fields that are of particular interest are highlighted in red.

In the IP header, the two fields of interest are:

- The IP total length (ip-len in **mmb**'s formalism): this 16-bit field's value represents the total length of the IP packet in bytes (including the IP header)
- The IHL (Internet Header Length, ip-ihl in **mmb**'s formalism): this represents the IP header's length in 32-bit words.

In the TCP header, a last field of interest for the task at end is found:

- The data offset (tcp-offset in **mmb**'s formalism): represents the TCP header's length in 32-bit words.

It is possible to recover the TCP payload length by subtracting the length of both the IP and TCP headers from the total IP length. This is a consequence of the structure of the IP packet used in the context of TCP, this is illustrated on figure 8.5.

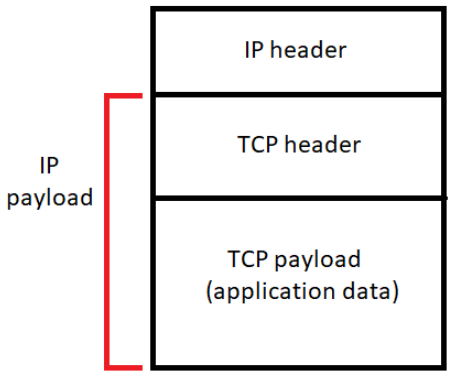

Figure 8.5: Structure of the IP packet in the context of TCP

We can thus formally write:

$$\text{payload-length} = \text{ip-length} - \text{ip-header-len} - \text{tcp-header-len}. \tag{8.3}$$

Using only values that are available in the TCP and IP headers (and consequently in **mmb**'s grammar), this can be rewritten as:

$$\text{payload-length} = \text{ip-length} - 4\ \text{ip-ihl} - 4\ \text{tcp-data-offset}. \tag{8.4}$$

This is an easy way to compute the payload length that could easily be done by a firewall to allow writing rules with the payload length as an attribute on which constraints can be set.

However, it is still possible, without modifying the firewall, to rewrite rules containing constraints on the TCP payload length by making use of the inequality shown above.

Let us say that we have the following subset of constraints in a given rule.

$$\{\text{tcp-payload-length} \leq a, \text{tcp-payload-length} > b\} \tag{8.5}$$

This is equivalent to (due to identity 8.4):

$$\{\text{ip-length} - 4\ \text{ip-ihl} - 4\ \text{tcp-offset} \leq a,\ \text{ip-length} - 4\ \text{ip-ihl} - 4\ \text{tcp-offset} > b\} \tag{8.6}$$

The IP IHL field has a minimum value of 5, and, as it is a 4-bit field, its maximum value is 15. The exact same goes for the TCP data offset. This means that if we consider all of the possible pairs of values for these 2 arguments, there are $11 \times 11 = 121$ possibilities.

It is thus possible to express a rule with constraints on the TCP payload length as 121 rules in which these constraints have been replaced by constraints on the total IP length by considering the value of the IP IHL and TCP data offset as constants.

For example, if we assume ip-ihl = 5 and tcp-offset = 5 (minimum values), then 8.6 can be rewritten as:

$$\{\text{ip-length} \leq a + 40,\ \text{ip-length} > b + 40,\ \text{ip-ihl} = 5, \text{tcp-offset} = 5\}. \tag{8.7}$$

Although this is obviously equivalent, one should note this has an important impact on the number of rules. Any rule with one or more constraint on the TCP payload length is replaced by 121 rules. If the number of such rules is high, it could result in an excessive number of rules.

However, the impact of this method was acceptable on the example considered. Using rules extracted with a decision tree from the data generated on the testbed using only data pertaining to requests and the features from table 8.1, the initial number of extracted rules (prior to all the modifications discussed in this section) was 39.

After applying everything that was discussed in this section including the potentially costly modification discussed above, the number of rules was 4604. This is much bigger but can easily be handled by **mmb** which was designed with scalability in mind and can handle 10^5 rules without a significant impact on performance [Ede+19].

One must also note that the rule set is redundant (as it covers the entire packet-space), only considering either DROP or ACCEPT rules is sufficient. Considering only DROP rules, only 1819 rules remain.

This covers all that is needed to configure the **mmb**-based firewall by using decision trees with the features of table 8.1.

This shows that it is possible to configure a firewall by deriving rules from labelled traffic data. Feature pre-processing was shown to have interesting applications to improve the quality of the derived rules and it was shown that, as long as there is an obvious way to rewrite rules on pre-processed data, applying such pre-processing is not problematic.

8.4 Conclusion: challenges in configurating real firewalls with such techniques

The first challenge comes in the form of the expressivity of the available features. As the simple example of the payload length illustrated very well, it may be possible but somewhat difficult to express a chosen feature with the attributes that are readily available to express constraints in the firewall's grammar.

However, realistically, the rule format is too restrictive to permit writing rules that are equivalent to features which correspond to even a slightly complicated function of the attributes which are readily available.

This limitation illustrates very well, that when considering what can and cannot be a feature of traffic, what the firewall offers is very well the limiting factor. As the different scenarios considered in the machine learning experiments have shown, the error rate with simple features such as the ones considered here is too high to be used for truly automatic firewall configuration.

This remains true regardless of hyperparameter tuning by following the best practices in machine learning and remains true for every classification algorithm that was considered.

What was however shown to reduce greatly the classification error is to use better features. Delays and PSH flags were not used because they may disrupt normal communication and are thus not really usable as such in practice, but they did help reduce the classification error. Similarly, flow-based models using averages as features presented many problems to be usable in practice, but their error rates were smaller.

Since configuration based on rules using features that are not available on the firewall of interest is either difficult and impractical or straight-up impossible, the only realistic solution is for the firewall itself to offer the ability to write constraints directly on all features considered by the classifier.

The other main issue is the important reliance on the format of the rules. The techniques that were studied will only ever work for configurating a firewall accepting a very strict format. One must be able to associate an action with a conjunction of inequality and equality constraints. Any other rule format is inadequate and would require a firewall-specific translation that may or may not be possible.

At this point, it is very clear that the automatic configuration techniques that are discussed in this thesis, are very much reliant on the capabilities and grammar of the configured firewall.

If such techniques are to be used and relied on, major developments regarding standardization are going to be needed. This essentially calls for a standardization of a rule format that is similar to the one of **mmb**.

However, since classification performance is still an issue, there is little motivation for such a standard to be adopted.

Automatic firewall configuration is possible but much more research is going to be needed in order to obtain acceptable error rates, so that this configuration can really be performed automatically and on-the-fly.

Two problems have been shown to be critical to the adoption of such techniques:

1. Finding features of traffic that will lead to an acceptable error rate and being able to build a firewall that can use constraints on said features.
2. The configured firewall must accept a specific rule format (conjunction of inequality and equality constraints.

There is no doubt that if such techniques work efficiently, the associated rule format will also eventually be adopted. It would thus most likely be interesting for further research to focus on finding better features that can be computed fast enough for a firewall to use them and still operate at line-speed.

Part IV

Conclusion

9 Conclusions and Outlook

The objective of this master thesis consisted in finding techniques to configure firewalls automatically in order to avoid the error prone process of firewall configuration. Traditionally, firewalls are configured by a system administrator defining explicitly every type of traffic that is allowed to go through it. In order to avoid manually writing such rules, firewall configuration was reformulated in a supervised machine learning framework.

This research and its results answer a lot of questions on how it is possible to use machine learning techniques to configure a firewall and which techniques can be used to achieve this objective. A generic technique to achieve this objective was found, in spite of the many difficulties specific to firewall configuration. Let us go over the learning outcomes that were highlighted in this work. Several matters need discussion.

A firewall consists in a program classifying packets in two categories: legitimate or malicious. In possession of labelled dataset, i.e. a set of packets for which the category is known, this looks very much like a typical classification problem, for which a variety of well-known algorithms already exist. The approach that was studied consists in treating firewalls as a classification problem to see how classification algorithms can be re-purposed for firewall configuration.

The first question one might ask is if there are any problems obtaining such a dataset. Obtaining legitimate examples is easy and straightforward. One may just record the traffic from legitimate applications they want to pass through the firewall.

Obtaining examples of what is not allowed could seem more difficult. It consists in pretty much all traffic that is not emitted by a legitimate application. In practice, to record traffic of the sort, one may use recorded traffic from previous attacks, which could for example come from an Intrusion Detection System (IDS).

Building representative data of both classes of traffic is thus not conceptually difficult but one must note that nothing that is not found illustrated in this data will be picked up on by the firewall. This means that if some new attack is different in every way from known attacks, it is unlikely to be considered by malicious by an automatically configured firewall.

If we now assume that such a dataset exists and is representative of both classes of traffic, one may assume that this pretty much comes down to a simple classification problem and applying out-of-the-box supervised machine learning algorithms, it is directly possible to configure a firewall from these examples.

In reality, this is much more complicated, and these techniques cannot possibly be applied as such without solving many problems that are specific to firewalls and traffic data. Let us go over these and how they were addressed.

The first significant issue to address is how recorded traffic is to be rewritten as a set of features. Two approaches were proposed: a per-packet and a per-flow approach. The main difference between both approaches is what will be considered as a data point by the classification algorithm. In the former, a data point is a packet, and in the latter, a data point is a flow.

Defining what is a data point is not enough, one must change it into a set of features, which is essentially a vector of input variables. What are exactly those variables is very important for the performance of the classifier.

Once the problem of changing data into pairs of features and output class is solved, comes in the very important issue of interpretability. Most classification algorithms offer little to no interpretability. This means that once "trained" from a set of labelled data, they become a sort of opaque classifier object that can be used from a vector of inputs (the features) can infer the output. What they however do not offer is insight on the relationship between inputs and outputs.

However, firewall configuration is not just another typical classification problem. In firewall configuration, interpretability is key. An opaque classifier cannot simply be incorporated into a firewall. If one were to do it, this would require building a firewall specifically to work with such classifiers. The classifier would need to be called for each new incoming packet, which could make high-performance traffic discrimination difficult depending on the underlying algorithm.

If it is not enough to warrant the necessity of interpretability, the issue of error handling must be considered. If something goes wrong with the automatic configuration process and some traffic gets handled differently than it ought to, let us say some legitimate traffic is wrongfully blocked, how is a system administrator supposed to rectify that issue if there is no useful way in which the classifier structure can be understood and edited?

One may suggest to retrain the algorithm with data that is more representative of that particular type of traffic. This may or may not work. There is indeed no guarantee that the training dataset is itself perfectly classified. In the cases where it does not work, there would be no way to rectify the situation. It is unacceptable in practice to have that little control. There is thus an actual need, both for performance and practicality reasons, for such techniques to be used only for configurating classical rule-based firewalls.

However, as pointed out, most classification algorithms are opaque and do not offer any possibility of being translated into rules. Rule extraction from opaque models is possible but is a relatively new field and there are not many truly generic techniques. Most of these techniques are generally specific to a particular type of classifier and most research in that domain has been about neural networks. Neural networks were not addressed in this thesis due to the difficulty of tuning them and their high computational complexity, which is problematic in the context of firewalls. This could however be an interesting subject for further research.

An algorithm that can be used easily to build rules from data is the decision tree algorithm. It has the ability to learn rules directly from data. This means that it can be used directly as a classifier using the training data or as a generic extraction technique for other classifiers.

Both approaches were considered, but no significant advantage was found in using it to extract rules from another classifier than using it directly. The idea of splitting the problem of configuration in two sub-problems that are classification and rule extraction is however interesting. This additional modularity of the solution could allow improvements in any of those fields to improve the quality of the overall solution.

Whatever approach is chosen, the outcome is almost exactly the same: a decision tree that can be translated into rules of a very specific format: a conjunction of equality and inequality constraints on the features' values. This format is adequate to configure a firewall and an example for configurating a solution based on **mmb** (Modular MiddleBox) was proposed.

Several possible sets of features were proposed, both for the per-packet and per-flow approach. The error rate was shown to depend greatly on the set of features. The per-flow approach presented many obstacles to generate an usable configuration. The per-packet approach was shown to be usable in practice. Although its error rate was somewhat higher, some of the machine learning experiments that were performed have shown that the error rate improves greatly when using more expressive features.

The set of features that was used to build a valid configuration for the **mmb**-based firewall does not allow for an acceptable error rate. However, including additional features like TCP PSH flags and packet delays reduce greatly the error rate. These were not used because this would result in preventing normal communication in some cases, but this illustrates that a more expressive feature set allows for smaller error rates.

Although these results are encouraging, none of them can be applied as such in real-world networks. What currently lacks is better set of features and firewalls that accept them in their grammars. Finding features that are expressive enough so that they can yield acceptable error rates will certainly be challenging.

The main reason why the current error rates are unacceptable is because the features are not descriptive enough to be able to distinguish legitimate and attack traffic in all relevant cases. That information is found mostly at the application layer. It is thus likely that improving the feature set relies on deep packet inspection (DPI), which is even more complicated in the context of encrypted traffic, i.e. 85% of web traffic nowadays.

What the future of automatic firewall configuration is made of will be a combination of several factors. It is relatively obvious that any improvements both in classification algorithms and rule extraction techniques for such algorithms will be of great help for the specific problem of automatic firewall configuration.

However, improvements in the related sub-fields of artificial intelligence will not be enough to make manual firewall configuration a thing of the past. Much more research dedicated to finding better features of traffic and building firewall that can use rules based on them is going to be necessary. Unsupervised machine learning techniques such as clustering would also deserve some investigation as they may allow an easier creation of labelled data for training the algorithms.

Automatic and on-the-fly firewall configuration and reconfiguration will one day be a reality but the road ahead both in artificial intelligence and in this brand new field is still very long.

Future works could focus on finding more expressive features, for example based on Deep Packet Inspection, to describe traffic efficiently. Using them would require modifying an existing firewall, or tool that can be used as a firewall (e.g **mmb**), in order to take advantage of such features while still operating at line speed.

Another subject that could be of interest to future works could be the use of clustering and other unspervised machine learning algorithms to build labelled training datasets from unlabelled data. Being able to rely on unlabelled data would make the configuration process even simpler as it would permit configuration from any recorded traffic data, without initially knowing which packets are legitimate or not.

Bibliography

[Don19a] B. Donnet. *Lecture: INFO0045: Introduction to computer security, Part 2, Chapter 4: Network attacks*. Université de Liège. 2019.

[Don19b] B. Donnet. *Lecture: INFO0045: Introduction to computer security, Part 2, Chapter 5: Spam*. Université de Liège. 2019.

[Ede+19] K. Edeline, J. Iurman, C. Soldani, and B. Donnet. **mmb: Flexible High-Speed Userspace Middelboxes**. In: *Proc. ACM Applied Networking Research Workshop (ANRW)*. July 2019.

[Ede19] K. Edeline. **Characterizing and Modeling of Transport-Based Middleboxes**. PhD thesis. Université de Liège, Oct. 2019.

[fdi] fd.io. **What is VPP?** In: *https://wiki.fd.io/view/VPP/What_is_VPP%3F*. Modified: May 2017, Accessed: October 2020.

[GW19] P. Geurts and L. Wehenkel. *Lecture: ELEN062-1: Introduction to machine learning*. Université de Liège. 2019.

[HBV07] J. Huysmans, B. Baesens, and J. Vanthienen. **Using rule extraction to improve the comprehensibility of predictive models**. *SSRN Electronic Journal* (2007).

[Inv] Federal Bureau of Investigation. *2020 Internet Crime Report*. https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf. Published: March 2021, Accessed: May 2021.

[Kal] Kali.org. **Kali Default Non-root User**. In: *https://www.kali.org/news/kali-default-non-root-user/*. Published: 2019-12-31, Accessed: 2020-09-20.

[McK] Martin McKeay. *Parts of a whole: effect of COVID-19 on US Internet traffic*. Akamai blogs, https://blogs.akamai.com/sitr/2020/04/parts-of-a-whole-effect-of-covid-19-on-us-internet-traffic.html. Published: April 2020, Accessed: May 2021.

[NFS12] M. Norouzi, D. Fleet, and R. Salakhutdinov. **Hamming Distance Metric Learning**. In: *25th International Conference on Neural Information Processing Systems*. NIPS'. 2012, 1061–1069.

[NS11] M. Narasimha Murty and V. Susheela Devi. **Pattern Recognition: An Algorithmic Approach**. Springer, 2011.

[Rap] Rapid7. *Metasploit framework manual pages.* https://docs.rapid7.com/metasploit/. Accessed: February 2020.

[Sec] Offensive Security. *Exploit Database.* https://www.exploit-db.com/. Accessed: February 2020.

[Ser] IBM Managed Security Services. *IBM Security Services 2014 Cyber Security Intelligence Index*. Published: June 2014, Accessed: May 2021.

[Tru79] G. V. Trunk. **A Problem of Dimensionality: A Simple Example**. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1:3 (1979).

[Ver14] R. Verbruggen. **Creating firewall rules with machine learning techniques**. Published in Radboud University's online article repository. MA thesis. Nijmegen Netherlands: Kerckhoffs institute Nijmegen, 2014. URL: https://www.ru.nl/publish/pages/769526/roland_verbruggen.pdf.