Using machine learning in network traffic analysis for penetration testing auditability

Tiko Huizinga
thuizinga@os3.nl
Supervisor: Alex Stavroulakis

November 14, 2019

Abstract

Pentesters perform an authorized simulated attack on a computer system to find vulnerabilities. They need to log all the actions they do during a test. Many pentests happen over the network and pentesters use different tools to execute their tests. During this research, we explore the possibility of automatically recognizing these tools by only looking at the network metadata. We use machine learning to create a model which performs this classification. To create such a model, it needs example data for the classes it needs to learn. This research covers the whole part of the machine learning process, from creating traffic samples to preprocessing the data to actually training and testing a model. The results of this research show that preprocessing and classification can happen fast enough to perform live during a pentest. For the limited number of classes we defined, the resulting model is highly accurate. Further research is needed to create a new model with enough classes for a pentester to actually be able to classify most or all of their traffic.

1 Introduction

Penetration testing, short pentesting, is the task of evaluating the IT security of any system by finding and exploiting vulnerabilities on that system. In the pre-engagement phase of a pentest, the pentester and the client agree on the scope of the test. This is written down in a document called the Permission to Test document. The scope defines the limitations of what the pentester is allowed to do on the system and how pentesters should spend their time[9].

For the auditability of the pentest, a pentester needs to record their actions. This is useful for three scenarios. The first scenario is when a pentester wants to prove that he performed certain actions. The second scenario is when a pentester wants to prove that he did not perform certain actions. Finally, it can be used as reminders for the pentester what he did in what order to help him write a report.

Pentesting is a creative process which is often done with a combination of automated tools and manual investigation. These different tools create different sources which can be used to create auditability. Examples of such sources are the command line history, log files from tools, screen shots, and manual notes[10]. The problem is that each tool leaves different traces on different locations. One thing all the tools have in common (in a pentest over the network) is that they send data over the network. A pentester could then capture their network data from their own machine or all pentesters could connect to a central server which does the classification for them and automatically combines the results if desired.

For this research, we focus on using this network traffic as a source for pentesting auditability. More specifically, we will apply machine learning on the network metadata (IP and TCP headers). The reason for this is that there are many different types of tools and we believe that each tool behaves differently on metadata level. An example of this is a port scan sending a constant stream of packets to the same IP address but to different ports whereas normal web browsing would contain smaller bursts of traffic to different destinations with larger intervals between destinations. Often, there are different tools for the same purpose and using the same method. A machine learned model might classify network traffic from these different tools in the same class but this does not have to be a problem and might even be useful because often, the task a pentester executes is more interesting than which tool he used.

In a best case scenario, a pentester would run a tool during the pentest that records their network traffic and classifies it live. This tool would create an overview of which action was performed at what time by the pentester. If a pentester creates a new class of traffic which has not been learned by the model yet, the model can be updated by uploading a file containing only traffic from of this type.

We realize that there are other methods for classifying network traffic like Intrusion Detection Systems (IDS). A specific example of an IDS is Snort. This is a rule based system which means that a human usually creates a rule for each type of traffic that needs to be logged/detected. These rules look for specific patterns or strings in a packet header or payload. A deep understanding about the attack is required to create such a rule. Our idea is that many different classes of traffic can be recognized by only their metadata fingerprint. The reason we do not look at the payload, is that machine learning uses statistics on numbers and pattern recognition in text data (like the payload) is a whole new field of study.

1.1 Research question

This research will not create a tool which has all the functions of the best case scenario as described in the introduction. We will focus on creating a Proof of Concept (PoC) which is able to preprocess network traffic and create a machine learned model that is able to classify different specified classes of traffic using pre-captured data.

The research question we want to answer is as follows:

How applicable is using machine learning in network traffic classification for pentesting auditability?

To answer this qustion, we define the following sub questions:

1. *Is preprocessing and classifying fast enough to do it live?*
2. *What is the accuracy of the model?*

2 Related work

2.1 Pentesting auditability

Maritsas and Tsiridis define one of the sources of auditability data as the command line in- and output. They propose a framework on how pentesting auditability should be done among a team. They create a prototype implementing this framework to enhance collaboration, action recording and documenting. To record the actions of a pentester, they create a tool which captures all the command line in- and output including timestamps[10].

Doorn and Spithoff create a network analysis tool for pentest auditability based on TCP metadata. They define characteristics for port scans and for reverse TCP shells by manually analyzing the data. Characteristics they look at are TCP flags, SYN numbers and timing. They achieve 100% accuracy in recognizing a port scan or a reverse shell. Unfortunately, they do not provide data about the number of false positives this method produces. Because of this, it is difficult to determine the value of this tool.[8]

2.2 Network traffic classification using machine learning

Zhang et al. propose a non-parametric approach to classifying network traffic[11]. The new framework they propose is Traffic Classification using Correlation (TCC). They use Bag of Flows, BoFs to correlate traffic flows which are generated by the same application. They define a flow as successive IP packets with the same five-tuple (src ip; src port; dst ip; dst port; protocol). They also propose a novel nonparametric approach based on Bayesian decision theory for the BoF model-based traffic classification. Finally, they use different nearest neighbor (NN) techniques to implement TCC.

Ali et al. use different machine learning algorithms (J48 Tree, Nave Bayes, Random Forest, Support Vector Machine) to classify a public data set containing different network attacks (nmap, lan dos, buffer overflow, ftp_write and more)[1][4]. Nmap was best classified with J48, Random forest and SVM and significantly worse with Nave Bayes. The public dataset they use dates back to 1999.

The dataset contains three different types of data:

- Basic features of individual TCP connections

- Content features within a connection suggested by domain knowledge
- Traffic features computed using a two-second time window

Pacheco et al. created a survey on how machine learning is used in network traffic classification[5]. During this research we will use the taxonomy of machine learning as they describe it in the following steps: Data collection, feature extraction, feature reduction, algorithm selection and model deployment. For data collection they describe three possibilities. The first one is real traffic which has the advantage of being the best representation of real world traffic which is what you aim for during data collection. The disadvantage is that in many cases, this traffic is private. The second one is traffic generation. This means that a researcher simulates real traffic conditions by modeling real interactions through scripts. The last way to generate traffic is emulation which aims at setting a scenario close to a real one and generate the traffic manually. During this research, we will be emulating pentest traffic because this is the fastest valid approach and data collection is not the main goal of this research.

For feature extraction, they propose statistical based features such as packet length, graph based features which uses a graph of nodes in the network, time-series based features like inter arrival times between packets and hybrid combinations of these. During our research, we will use both statistical and time-series based features.

3 Method

In most machine learning research, the general method is very similar. The goal of machine learning is to create a model that makes predictions or decisions without explicitly being programmed for that task. This requires a dataset which can be used to train a model. The closer the dataset resembles the actual data on which predictions must be made, the higher the accuracy of the model. A dataset can contain a lot of noise and that is where preprocessing and feature engineering come in. In machine learning, features are the properties of data which you want to use to predict your result. The next step is to select an algorithm which makes it possible to learn or create a model. Finally, this model should be tested to verify its performance. The next subsections describe these steps in more detail. The results of these steps are shown in Section 4.

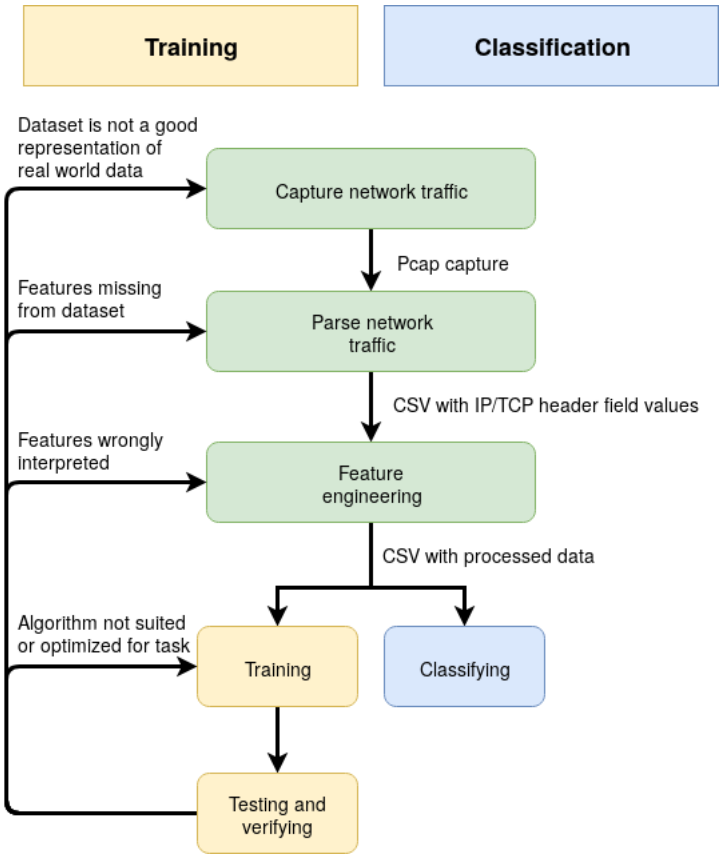


Figure 1: Flow chart of machine learning and classifying process.

3.1 Data source

The quality of the dataset is very important to gain desired results. Some research use predefined or existing datasets. A very popular dataset is the Knowledge Discovery and Data Mining Tools Cup (KDD Cup) from 1999[4]. This dataset is not a raw packet capture but already preprocessed data. The features in this dataset can be split in three categories: basic TCP features, features with domain knowledge and features based on a two-second time window. An advantage of using a pre-existing dataset is that if a popular dataset is used by multiple researchers, it provides a fair way to compare their results. A disadvantage of using an existing dataset is that the chance is low that it is a good representation of the real world scenario where the model will have to operate. In this research, the goal is to create a PoC for pentesters to help them classify their data. We want pentesters to be able to define their own classes of data. Because of this, we will create our own data sets for training and testing

the model. This allows us to execute the machine learning process from start to end and to modify and improve the process after reviewing the results.

We want our model to be able to make a prediction between the following classes:

3.1.1 Dirbuster

Dirbuster is a tool that uses a wordlist to brute force directories and files on web and application servers.[1] This means that the number of packets per second will be high, server side ports will usually be 80 or 443 and the tool will not connect to other ports or other destinations.

3.1.2 Nmap

Nmap is a network discovery and port scanning tool.[2] It sends specially crafted IP packets to the target(s) and based on the (lack of) response, determines if the target is online and which ports have services listening and responding on them. For our PoC, we use Nmap to only scan a single target using three different techniques, giving us three classes of data.

SYN scan The Nmap SYN scan is the default and most popular scan type. This scan crafts and sends TCP SYN packets to a number of ports on the target (1000 by default). If the port is open, the target sends a SYN/ACK packet back and Nmap sends a TCP Reset (RST) packet to close the connection before it is fully established. If the port is closed, the target will reply to the SYN with a RST packet. If the target does not respond at all to the SYN packet, Nmap sends a number of retransmissions, based on packet loss statistics, before continuing with the next port.

ACK scan The Nmap ACK scan is used to see if a firewall is active on the target ports. Nmap sends an ACK packet to the target. Default TCP/IP behavior would be for the target to send a RST packet. Lack of this response on a host that is confirmed to be online, indicates that this port might be filtered using a firewall.

TCP connect scan The Nmap TCP connect scan tries to establish a full TCP connection to the host: It sends a SYN, expects a SYN/ACK and sends an ACK to establish the connection. This scan is used for the same purpose as the SYN scan but then for users that lack permissions to handcraft packets. These packets are crafted by asking the Operating System to establish a connection, which is an operation non-superusers are allowed to do. After establishing a connection, Nmap immediately closes the connection by sending a RST.

[1] `https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project`
[2] `https://nmap.org/`

3.1.3 SSH

Establishing a secure shell (SSH) connection to the target is also common during a pentest. By default SSH listens on TCP port 22. For each character that is typed in an SSH command prompt, four TCP packets are generated:

1. Client ⇒ Server: Character 'C' typed.
2. Server ⇐ Client: ACK
3. Server ⇐ Client: Character 'C' appeared on the screen.
4. Client ⇒ Server: ACK

Depending on the server, the second and third message might be combined for optimization purposes.

3.1.4 Web browsing

For the final class, we captured the packets of a user browsing the web on both HTTP and HTTPS webistes.

3.1.5 Capturing packets

For each of these classes, we create a packet capture using tcpdump. To make sure we only capture the relevant packets, we filter tcpdump on the host IP address of our target. This ensures that we will not capture noise generated by other applications running on our host device.

These captures are the source for our ground truth. Ground truth are the labels corresponding to datapoints considered/assumed to be true used for training and testing. Not per definition always true, as it is a measurement that can contain errors. The cleaner and more complete the ground truth is, the better the algorithm will be able to learn the correct behavior.

The results of the data gathering step are multiple packet capture (pcap) files, each labeled with their class name. These files are binary data and not yet ready for machine learning, as that needs numerical input. Parsing of these pcaps to numerical data is part of the preprocessing step.

3.2 Preprocessing

3.2.1 Parsing PCAP

During the preprocessing task, we parse the raw pcap output from tcpdump to numerical values to create valid input for any machine learning algorithm. This research uses network traffic meta data as data source. Specifically, we focus on the IPv4 and TCP headers. We have to choose between either TCP or UDP and either IPv4 and IPV6 because we have to create a data structure that is the same for each packet. TCP and UDP have different header fields and parsing them will result in different columns. Adding both to the same

data structure would lead to empty fields for one of the two for each packet and machine learning algorithms can not handle empty fields. A possible solution for this could be to fill the empty fields with zeros but for our PoC we leave this out to reduce the complexity. We choose TCP and IPv4 over UDP and IPv6 because these are the types supported by all the classes of data we choose.

As the payload of the different types of packets will have different formats, this research focuses on the values of the TCP and IP header fields. We will use Scapy, a network packet manipulation tool running on Python, to parse the packets to their values. Appendix A and Appendix B show the values from the IPv4 and TCP headers as Scapy parses them.

One problem with the output from the parsed pcap, is that not all data is numerical. IP addresses are saved as strings and the options field in the IP header is an array printed as a string as well. The first problem with the IP addresses will be fixed during the feature engineering step. For the IP options field, we choose to not differentiate between the different options in our feature set but, for each packet, count the number of IP options that are set. As the IP options field is rarely used, a tool that does use it, can be identified by it.

3.2.2 Feature engineering

When we have parsed the data, we have a table where each row is a packet and each column corresponds to a TCP/IP header value. All these columns are called features and they represent the measurable properties of a packet.

The next step is to transform the features from their current value to more relevant values. This is called feature engineering. The features, as they are in the table, do not tell us everything we could know about those packets even though some information is actually in the table of data. If a human sees many packets per second, they would recognize this but, based on the current features, a computer wouldn't. The context in which a packet is measured is important to recognize which tool might have created the packet. We define the context based on time. That is why we will modify the following features:

Time The timestamp of a packet does not give us any indication of which class a packet belongs to. However, it does give us a way to create context for new features like: number of packets with the same destination IP address as this packet in 0.2 seconds before and after the current packet. This method originally comes from [4] where they use 2 seconds, but we choose 0.2 seconds because some of our data captures last less than 10 seconds. Having a total of 4 seconds (2 before and 2 after each packet) of context time would take up a large portion of that scan, resulting in too small differences per packet.

IP addresses (source & destination) The IP addresses are not related to the class of traffic because in each scenario, the addresses will be different from the addresses being used for training the model. However, the number of packets within a set time frame coming from the same source or going to the same destination does actually say a lot about the class of traffic. Tools which

are brute forcing anything over the network, like an Nmap scan or a dirbuster attack, send many packets to the same host in a short amount of time.

TCP ports For classifying between different types of traffic, the source and especially the destination ports are useful because generally the same application runs on the same port. To recognize portscanners, it is also interesting how many packets within a certain timeframe went to the same port. That is why we introduce the fields same src_port and same dst_port. These fields contain the number of packets in 0.2 seconds before and after the current packet with the same source or destination port.

The following pseudocode shows how the number of packets with the same source address before and after each packet are counted. This works the same for the other context features. It is important that this algorithm is efficient because calculating these features must be done for both training and using the model. If it has a higher than linear time complexity, it would not be possible to calculate these features live during an intense packet capture. During this algorithm, we keep a list which has a count for each possible value of 'source_ip' which is updated for each packet. The algorithm loops over all the packets in the network data N once. Within this for loop, there are two while-loops; one to manage the beginning of the 'context time frame' and one to manage the end. Within these while-loops, the indices 'before' and 'after' are increased by one and the new values are kept during the start of the next loop. This means that after the algorithm has finished, the three values b, c, a have gone from zero to $|N|$ (the numer of rows in N). All the individual operations in this code are $\mathcal{O}(1)$: accessing value in array, accessing and modifying value in hash table. This makes the total time complexity of this algorithm $\mathcal{O}(N)$.

Algorithm 1 Create context feature: same_src

INPUT: Two dimensional array of network data $N[][]$ where each row is a packet and each column is a feature. Context time T is the time in which packets before and after the current packet are counted.

OUTPUT: Two dimensional array of feature-engineered network data F containing original features and the feature same_src.

c is the index of the current element in list N

b (before) is the index of the first element in N where $N_c.time - N_b.time \leq T$

a (after) is the index of the last element in N where $N_a.time - N_C.time \leq T$

src_count is a hash table with default value 0. It keeps the number of times each source address is present within the context time.

F is a copy of N with extra column *same_src*

for all packets N_c in N **do**
  if $C > 0$ **then**
    $src\_count[N_{c-1}.src] += 1$
    $src\_count[N_c.src] -= 1$
  end if
  while $N_c.time - N_b.time > T$ **do**
    $src\_count[N_b.src] -= 1$
    $b += 1$
  end while
  while $a + 1 < |N|$ and $N_{a+1}.time - N_C.time < T$ **do**
    $a += 1$
    $src\_count[N_a.src] += 1$
  end while
  $F_c.same\_src = src\_count[N_c.src]$
end for
return F

3.3 Training

To train the model, we first have to choose an algorithm. We use the Python library scikit-learn which has many machine learning algorithms built in. They published a flowchart on their website to help determine which algorithm is best for the task at hand[3]. Following this chart, the first algorithm we end up with is: Linear Support Vector Classification (SVC). The path we follow here is: (More than 50 samples: Yes), (Predicting a category: Yes), (Do you have labeled data: Yes), (Less than 100K samples: Yes). Besides this, many related works include SVM as one of the classifiers for network traffic classification[2][6][7].

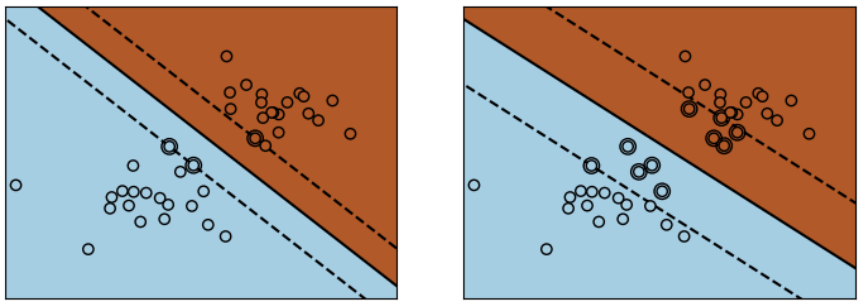
So we will train the model using the Support Vector Machine (SVM) algorithm. SVMs are well known supervised binary classifiers. Supervised means

[3] `https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html`

that learning the model requires training data which is already classified. A binary classifier is able to classify between two classes. Of course, we have multiple classes and thus need multi class classification. For N classes, this is possible by creating a classifier between each combination of classes. With four classes A, B, C, D, this is a classifier for: (A,B), (A,C), (A,D), (B,C), (B, D), (C,D). The pattern here is that there are $(N-1)+(N-2)+...+1$ classifiers, which can be rewritten to $n \cdot (n-1)/2$. This is known as the handshake problem.

SVM creates a hyperplane of $d-1$ dimensions in an d-dimensional space where d is the number of features of the dataset. In a two dimensional space, this would look as shown in Figure 2. The dotted lines are the margins. To calculate the margin in Figure 2b, the distance from each of the points to the hyperplane is calculated and the margin on each side is the distance from the closest point to the hyperplane. In Figure 2b, you can see that there are datapoints within the boundaries of the margin. This is possible because of the so called penalty term (we call this term C). A low small penalty term allows more datapoints to fall within the margin, resulting in a larger margin. A larger penalty term makes it less likely for datapoints to fall within the margin, resulting in a smaller margin.

If all the training data is correctly classified, a larger penalty term would lead to a more accurate classifier. If the training data contains errors or outliers, it is preferable to decrease the penalty term so that they do not influence the hyperplane too much. Choosing a large penalty term, could lead to overfitting on the training data. Overfitting is the problem where the model is too specific for the training set and the training set is not a good representation for the real world. An example of how this could happen in our dataset is when the training data contains outliers like packets from another tool. With a high penalty term, the model tries to find a hyperplane that still includes those packets. With a lower penalty term, these packets are excluded from the model.



(a) Small margin, large penalty term C (b) Large margin, small penalty term C

Figure 2: SVM hyperplanes [4]

Other than the penalty term, the 'kernel' is a parameter which can be tuned for SVM. Figure 2 shows a linear kernel, which generates a linear hyperplane (in 2D dimension a straight line). Other kernels are a polynomial kernel and the 'Radial Basis Function' (RBF) kernel. The polynomial kernel creates a polynomial hyperplane and the RBF kernel can create circular hyperplanes, which is usefull if class A is a cluster of points and class B are data points all around this cluster. As explained at the start of this section, our PoC is written in python using the scikit-learn svm.svc implementation with a linear kernel.

3.4 Testing and verifying

We split our dataset into a smaller trainingset and a larger testset. The model will be created using the trainingset. To test the accuracy of the model, we use the F1-score which is the harmonic mean of the precision and recall. This is a widely used metric for the performance of classification models [11].

In binary classification, there are two classes usually called Positive and Negative. The classification metrics are then created using a confusion matrix shown in Table 1.

		Predicted class	
		Positive	Negative
Actual class	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Table 1: Confusion matrix for binary classification

Precision or Positive Predicted Value (PPV) is the number of True Positives divided by the sum of True Positives and False Positives:

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

Recall or True Positive Rate (TPR) is the number of True Positives divided by the sum of True Positives and False Negatives:

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

Looking at Table 1, the precision only uses values from the first column and the recall uses the values in the first row.

For multiclass classification, the confusion matrix grows to a table of $n * n$ with n being the number of classes. The precision and recall are then calculated per class. A True Positive is an item where the predicted and actual class is the same. False Positives for class c are all the items where the predicted class is c and the actual class is not c. False negatives for class c are all the items where the actual class is c but the predicted class is not c. This way, the precision and

[4] `https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html`

recall can be calculated for each class. The F1-score is the harmonic mean of the precision and recall:

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

The Nmap scans for the top 1000 ports contain between 2000 and 3000 packets. From all our classes, these have the fewest number of packets. Large discrepancies between the number of packets per class can lead to bias in the model [3]. To prevent this, we choose to cut off the datasets of our captures for this PoC after the first 3000 packets. Per class, all the packets come from the same capture, which could lead to overfitting. To test for this, we also check our model with separate captures. The results of this are shown in Section 4.2.1.

4 Results

This section contains the results for the experiments based on our method and research question. To make reproduction of our results possible, the code of our PoC is online available on GitHub[5]. During these experiments we focus on speed and accuracy to be able to say something about applicability.

4.1 Capturing & preprocessing

In this section, we show for each class the number of packets that we use during this research and the time span in which they were captured. If these times are longer than the preprocessing and classification times, this would indicate that during a pentest, preprocessing and classification could be performed live.

Class	**#Packets**	**Capture time(s)**	**Preprocessing time(s)**
Dirbuster	3000	4.50	2.24
Nmap ACK	2641	8.37	1.72
Nmap SYN	2451	5.90	1.63
Nmap TCP	2306	17.00	1.5
Web browsing	3000	333.46	2.26
SSH	3000	122.63	2.33

Table 2: Preprocessing times for each data class

Table 2 shows that the 'brute force classes' have significantly shorter capture times than the web browsing and SSH classes and thus more packets per second. For all the classes the preprocessing times are shorter than the capture times. This makes the idea of live preprocessing and classification possible.

In Section 3.2, we showed that preprocessing has a linear time complexity, meaning that double the number of packets, would double the time needed for preprocessing. This makes it scalable for a larger number of packets.

[5] `https://github.com/THuizinga/Pentest-network-traffic-classification`

4.2 Model learning

This section shows the process and results of creating the machine learned classification models. We find flaws with the first model we create and try to fix them in the next model.

4.2.1 First training

For the experiments in this section, we train the model using the six data classes as mentioned before, different values for error term C (0.1, 1, 10) and two amounts of packets (150, 300) as training data. Table 3 shows the most important results for the trainingset of 150 packets and Table 4 shows the results of training with 300 packets. The total dataset is here 15000 packets and the data that is not being used for training, is used for testing the accuracy and calculating the F1-score. Splitting the data in separate training and test sets is done randomly and we repeat each experiment 5 times to get reliable results.

	C=0.1	C=1	C=10
Training time (s)	9.6766	9.5322	3.361
Classifying time (s)	0.0398	0.042	0.0394
Average F1 score	0.892	0.896	0.892

Table 3: Overview results of size trainingset = 150 packets

	C=0.1	C=1	C=10
Training time (s)	118.525	42.3954	28.2364
Classifying time (s)	0.045	0.0442	0.0446
Average F1 score	0.942	0.96	0.958

Table 4: Overview results of size trainingset = 300 packets

The most important conclusions we can make from these results is that a higher value for the error term C leads to a faster training time with minimal differences in the F1-score. A larger penalty term means that the algorithm allows less items to fall within the boundaries of the hyperplane, and thus choosing a hyperplane with a smaller boundary.

We can also see that doubling the number of packets to train the classifier, resulted in more than linear growth of training time. This is expected because the SVM algorithms training time is more than quadratic with the number of input samples. Relatively, the training time for $C = 1$ increased the least.

Testing with new unbiased dataset The previous accuracy tests were done by randomly splitting the data in a test- and trainingset. Each class of data from this dataset contains packets of one capture. This means that the highly accurate results might partially be due to overfitting on the data. To test if

this is the case, we create a new packet capture of an Nmap SYN scan with a different target.

Dirbuster	Nmap ACK	Nmap SYN	Nmap TCP	SSH	Other
1716	39	48	1	37	159

Table 5: Nmap SYN scan classification results without overfitting

Table 5 shows that most packets have been classified as Dirbuster traffic even though it was a SYN scan. Only 48 out of 2000 packets are classified correctly. This indicates that the high accuracy scores shown previously are indeed an indication of overfitting. One possible cause for this overfitting is that there are features which are capture-specific. Because of this, in the next model we create, we will exclude the following features:

- IP Identification
- IP checksum
- TCP source port
- TCP destination port
- TCP checksum

The reason we also exclude the TCP ports, is because although the ports of our target are fixed in our experiments, the ports from where we run our tools are chosen by the OS and different each time.

4.2.2 Training without capture-specific ports

Now that we have removed theses features, we will create a new model with the same dataset. The results of this training are shown in Table6.

The first thing we notice is that the model performs much better with an average F1-score of 0.99. This is a great improvement and shows that the model with the capture-specific features indeed used those features to make the predictions. Now we will again test our model with newly created Nmap scans on a different destination compared to the trainingset.

Testing with new dataset Table 7 shows the result of classifying an Nmap SYN, ACK and TCP scan with a different destination than used in the trainingset. As we can see, the recall values for the ACK and TCP scans are similar to the values in Table 6. The Nmap SYN scan has a lower performance than before. This is mainly due to 114 packets being classified as a TCP scan and 71 packets as a Dirbuster scan. We will look more closely to the packets which are misclassified as TCP.

The model is a combination of classifiers for each combination of two classes separating them. We can inspect the classifier between Nmap SYN and Nmap

	Precision	Recall	F1-score	Support
Dirbuster	0.99	1	0.99	2841
Nmap ACK	0.99	1	0.99	2508
Nmap SYN	1	0.98	0.99	2320
Nmap TCP	1	0.99	0.99	2197
Web	0.99	0.99	0.99	2859
SSH	0.99	0.99	0.99	2854
Average	0.99	0.99	0.99	15579

Training time (s)	10.172
Number of training packets	819
Classification time (s)	0.040
Number of classification packets	15579

Table 6: Training results without capture-specific features

	Predicted class					
	Dirbuster	Nmap ACK	Nmap SYN	Nmap TCP	Web	Recall
Nmap ACK	0	2802	1	20	0	0.99
Nmap SYN	71	21	2563	114	2	0.92
Nmap TCP	8	6	0	2584	2	0.99

Table 7: New Nmap scan classification results

TCP. For each feature, it has a weight of how important that feature is for that classifier. The feature with the highest weight between the Nmap SYN and TCP scan is *same sequence number*. This feature counts the number of packets with the same sequence number 0.2 seconds before and after the current packet.

If we look at Figure 3, we see that most of the values for *same sequence number* for the Nmap SYN scan are very different from those of the TCP scan, but a small portion falls within the small range of the TCP scan. We believe this could be one of the causes for the misclassification.

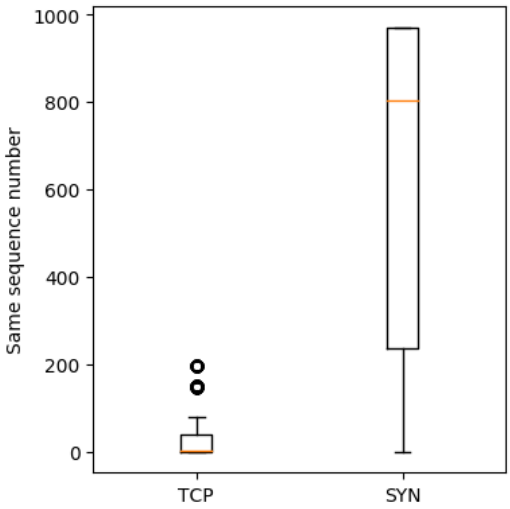


Figure 3: Distribution of 'Same sequence number' for Nmap TCP and SYN scans

5 Conclusion

5.1 Classifying time

The results in Section 4.1 show that the time to preprocess the data was always at most half the time in our experiments than the time needed to capture the data. The time needed to classify 15579 preprocessed packets is 0.04 seconds. This is a clear indication that during a pentest, data could be processed and classified live so that the pentester can see the results during and directly after the test without delay.

5.2 Model accuracy

In Section 4.2, we show that the accuracy of the model greatly depends on the features that are used. It is important to avoid features that are capture-specific. Such features result in a model that takes longer to learn, is less accurate and has a higher possibility to overfit on these features. We also tested the model with new captures to a different target (just like different pentests have different targets). This test shows that there has been some overfitting in the original results but that the classification still had an average recall of 0.97. This is comparable with the results from [1].

5.3 Applicability for pentesters

For a pentester, these results would need to be converted to a list with tools that are ran with corresponding timestamp. This could be done by counting the number of packets within a certain time frame that are classified as class X. For each tool the (minimum) expected number of packets would need to be known to create a threshold. If the number of packets counted for X come above this threshold, the conclusion would be that tool X has been run.

The lowest support value in our final model is 0.92. Setting this threshold to 90% of the expected (minimum) number of packets should be a safe value to identify different classes of traffic. This method would still have the risk of misclassification if many packets are misclassified as one class but it does reduce noise of all the single misclassifications. In this form, we do believe that machine learning could be a useful addition to pentesters for their auditability, but only as a supporting tool which results still need to be verified by the pentesters themselves.

6 Discussion

6.1 Unknown classes

The model needs to be trained for each tool a pentester uses during a test. If the model has to classify traffic from an unknown class, it would be classified as the known class with the most resemblance. It is not possible to create one 'other' class that catches all unknown classes because if each class is a cluster in a multi-dimensional graph, 'other' traffic would be on multiple sides of different classes, making it impossible to create a correct hyperplane separating the classes.

6.2 Running multiple tools parralel

The network captures used in this research, always contained traffic from one specific tool. If, during a pentest, multiple tools are ran at the same time, this influences the time-based features. This could make it look like one tool is sending many packets per second which can impact classification.

6.3 Differences in network speed

Differences in network speed between training and real world data might result in misclassification because the number of packets per second are different. This could be partially fixed by having one feature containing the number of packets per second and the other time-based features being the a ratio to that value. Another feature type that could be interesting are context features which are not based on time but on a set number of packets before and after the current packet.

6.4 Metadata versus payload inspection

This research focuses on metadata because the IP/TCP headers always have the same format which makes them easy to parse. The six classes we defined during this research were very good distinguishable based on metadata only. In a real world scenario, there might be many more classes which are harder to distinguish from each other by metadata. Using domain knowledge about those classes, would then be a solution. This does take a bit away from the idea of machine learning for which no domain knowledge is needed but a combination of the two might give a powerful tool.

7 Future work

7.1 Diversify training data

The training data is currently not very diverse because it contains one capture per class of data. A more diverse trainingset with captures on different targets and on faster and slower networks would make a more complete representation of real world data.

Further the dataset needs to be expanded to include all the classes of data a pentester needs to classify.

7.2 Other algorithms

This research focussed on Support Vector Machines with a linear kernel. Research on other machine learning algorithms would be useful to see if that results in faster or more accurate models. Suggestions for other algorithms are: K-nearest neighbors, SGD classifier and decision trees.

7.3 Extend feature set

The current feature set could be extended with relative time based features and context features based on a set number of packets before and after the current packet as described in Section 6.3. Furthermore would features based on the payload of a packet be useful as described in Section 6.4.

References

[1] Abdinur Ali, Yen-Hung Hu, Chung-Chu (George) Hsieh, and Mushtaq Khan. Detailed analysis of network attack detection accuracy using machine learning algorithms. In *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, pages 58–62. The Steering Committee of The World Congress in Computer Science, 2018.

[2] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):16, Jun 2018.

[3] H. He and Y. Ma. *Class Imbalance Learning Methods for Support Vector Machines*, pages 83–99. IEEE, 2013.

[4] S. Hettich and S. D. Bay. KDD Cup 1999 data. Irvine, CA: University of California, Department of Information and Computer Science, 1999. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

[5] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Communications Surveys Tutorials*, 21(2):1988–2014, Secondquarter 2019.

[6] M. Shafiq, Xiangzhan Yu, A. A. Laghari, Lu Yao, N. K. Karn, and F. Abdessamia. Network traffic classification techniques and comparative analysis using machine learning algorithms. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 2451–2455, Oct 2016.

[7] Syed Shah and Biju Issac. Performance comparison of intrusion detection systems and application of machine learning to snort system. *Future Generation Computer Systems*, 80:157–170, 03 2018.

[8] Marko MB Spithoff and Henk H.P.M van Doorn. Pentest accountability by analyzing network traffic & network traffic metadata. 2018.

[9] Penetration Testing Execution Standard. Pentest Standard: Pre-engagement, 2014. `http://www.pentest-standard.org/index.php/Pre-engagement#Introduction_to_Scope`.

[10] Alexandros Tsiridis and Stamatios Maritsas. Penetration testing auditability. 2016.

[11] Jun Zhang, Yang Xiang, Yu Wang, Wanlei Zhou, Yong Xiang, and Yong Guan. Network traffic classification using correlation information. *IEEE Transactions on Parallel and Distributed systems*, 24(1):104–117, 2012.

A All fields parsed from the IP header

Field (feature)	**Possible values**
Time (Seconds since epoch)	1562072053.48703
IP version	4
Internet Header Length	5 to 15
Type of Service/ DiffServ	0 to 255
Total length	20 to 65535
Identification	0 to 35535
Flags	0 to 5
Fragment offset	0 to 65528
Time To Live (TTL)	0 to 255
Protocol	0 to 255
Chekcsum	0 to 65535
Source address	String: e.g.: 127.0.0.1
Destination address	String: e.g.: 127.0.0.1
Options	Array as string: [ ]

Table 8: IPv4 header fields output from Scapy

B All fields parsed from the TCP header

Field (feature)	**Possible values**
Source port	0 to 65535
Destination port	0 to 65535
Sequence number	0 to 4294967296
Acknowledgment number	0 to 4294967296
dataofs	5 to 15
reserved	0
FIN flag	0 or 1
SYN flag	0 or 1
RESET flag	0 or 1
Psh flag	0 or 1
ACK flag	0 or 1
Urg flag	0 or 1
Ece flag	0 or 1
Cwr flag	0 or 1
Window size	0 to 65535
Checksum	0 to 65535
Urgptr	0 to 65535

Table 9: TCP fields parsed with scapy

C All features

ip_version
ihl
tos
ilen
IP flags
frag
ttl
IP protocol
same_src
same_dst
ip_options
same_src_port
same_dst_port
same_seq
dataofs
reserved
same_tcp_flags
fin
syn
rst
psh
ack
urg
ece
cwr
window
urgptr

Table 10: All 27 features used for learning