Master Thesis

Radboud University Nijmegen

Detection of Cryptominers in the Wild

Author:
Lars Deelen
L.Deelen@student.ru.nl

Supervisor:
dr. Veelasha Moonsamy
Radboud University
email@veelasha.org

Daily Supervisor:
Axel Ehrenstrom
KPMG
Ehrenstrom.Axel@kpmg.nl

Second Reader:
dr. ir. Eelco Herder
Radboud University
e.herder@cs.ru.nl

June 25, 2019

Abstract

Cryptomining malware takes up significant resources and computing power, with protection methods either focusing on behavioral analysis on individual systems or static network traffic signatures. This thesis aimed to find a network based solution that used behavioral statistics of miners for detection. A custom malware lab was created to perform controlled data collection of network traffic from cryptominer samples. Next, an automated pipeline was built to label the data and generate statistical features from the observed traffic. To explore for solutions, the data was scaled and correlated using a variety of methods for feature selection, resulting in fifteen mutations of the data. Finally, six different machine learning classifiers were trained on ranges of parameters to find the optimal tuning, resulting in a total of ninety combinations of classifiers and data mutations. The Naive Bayes worked best, minimizing false negatives with a recall score of 98.11% but a precision of only 19.55%. The potential for an improved system is shown without retraining the classifiers, that is already capable of increasing the precision to 99.69%, but lowers recall to 50.40%.

Contents

List of Figures

List of Tables

List of Used Software

Al-Khaser (version 0.77) A application that uses common tricks found in malware to detect virtualized environments [18] .

Anaconda (version 2018.12) A distribution for easily installing and managing machine learning and data mining packages, as well as Jupyter Notebooks [3] .

Cuckoo Sandbox (version 2.0.6) An open-source automated malware analysis system [24] .

DB Browser for SQLite (version 3.11.1) An application with a Graphical User Interface to easily interact with SQLite databases [54] .

Jupyter Notebook (version 5.7.4) An open-source web application for code, visualizations, text and more in a single document [36] .

Matplotlib (version 3.0.3) A Python plotting library that allows for easy creation of graphics and plots [29] .

Ninite (version unknown)A package manager to easily download and install popular Windows applications [61] .

OpenVPN (version 2.4.6) An open-source solution for creating Virtual Private Network connections [30] .

Pandas (version 0.24.2) A high-performance data analysis library for Python [42] .

pfSense (version 2.4.4-p3) A free and open-source firewall and router platform with many features, including support for OpenVPN connections [40] .

REMnux (version 7) A Linux toolkit for reverse-engineering and analyzing malware [82] .

Scapy (version 2.4.2) A Python library capable of crafting and dissecting network packets [7] .

scikit-learn (version 0.20.3) A Python library with many machine learning algorithms for preprocessing, dimensionality reduction, classification, regression, clustering, and model selection. [53] .

Snort (version 2.9.12) An open-source intrusion prevention system that can analyse network traffic using customizable rule sets [10] .

SQLite (version 1.0.109.0) A free, small, and fast SQL database engine [68] .

Tor (version unknown) An application for anonymously routing traffic over a network of volunteer servers on the internet [31] .

VBoxHardenedLoader (version 1.9.0) A tool for hardening VirtualBox virtual machines, making it harder for applications to detect the isolated environment [75] .

VirtualBox (version 6.0.8) An application to run and manage virtualized operating systems in an isolated environment on top of an existing operating system [48] .

Whonix (version 14.0.1.4.4) A security-focused Linux distribution that forces all network traffic over Tor [70] .

Glossary

Bitcoin The first widely used peer-to-peer electronic cash system.

conversation All network traffic between two IP addresses.

cryptocurrency Electronic cash based on cryptographic calculations and proofs.

cryptomining The process of verifying and adding new transactions to a blockchain using cryptographic challenges as proof of work. This process is computationally intensive.

flow All network traffic between two IP address-port combinations using a single transport layer protocol (UDP or TCP).

GBT getblocktemplate protocol.

KNN K-Nearest Neighbors.

mining Short for cryptomining.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

VM Virtual Machine.

Chapter 1

Introduction

Historically, one would exchange physical objects with others for objects of the same perceived value. A layer of abstraction was introduced by the concept of currencies as intermediary objects that represented a value, most commonly in the form of a coin. This was then abstracted to bank notes, which represented any number of coins. The dependency on all physical representations was solved by technological advancements, allowing us to store a digital number as the representation of our wealth. But with each of these transitions, the need for organization and infrastructure increased. Third parties, such as governments and banks, have been responsible for all financial transactions and the underlying infrastructure. A paper by Satoshi Nakamoto [46], noted the trust that was placed in these organizations:

> *"While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. [...] With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party."* [46]

The same paper also introduced a design for a peer-to-peer electronic cash system: the Bitcoin. The fundamental principle behind the Bitcoin, is that all actions rely purely on cryptographic proofs instead of trust. Computers have to solve a cryptographic challenge that requires a lot of computations. This process is known as *mining* - short for *cryptomining*. In addition, privacy can be achieved by using anonymous keys, whereas banks would need to be trusted with keeping all customer information private.

The possibility to maintain anonymity while using Bitcoin has attracted the attention of criminals [16]. In September of 2013, the *Cryptolocker* ransomware was the first ransomware to allow affected users to pay with Bitcoin [35]. But more importantly, resources of other systems could be hijacked and used in the gathering process of cryptocurrencies. Any processor, from an Internet of Things devices to corporate data centers or cloud servers, could contribute valuable computing power. Botnets have been abusing infected computers since at least as early as August 2011 [79], when Arbor Networks and Kaspersky Lab found malware with three embedded Bitcoin mining

applications. This was just two years after the Bitcoin was introduced. The embedded mining applications turn out to have become far more popular than ransomware campaigns [39].

Naturally, random malware infections can contribute to a large but unorganized mining network. However, organized botnets are also being used for cryptocurrency mining [55]. The major advantage of malware infections is the ability to gain persistence on infected systems as a process, which allows for mining at any moment that the machine is operational. While the average workstation might be operational for limited periods, corporate networks and data centers remain continuously operational and are therefore more valuable targets [8].

Mining in browsers is a relatively recent development, with the CoinHive service launching in August 2017 and a cryptojacking infection using this service already found in September 2017 [65]. Just a few weeks after CoinHive launched, 220 of the 100K most popular websites already used the service [43]. Note that this service did not yet have an opt-in mechanism [62] at that time, making most websites automatically start mining without asking the users for consent. The drive-by nature of this type of mining makes it easy to temporarily hijack resources without needing to infect the system and even use the GPU through WebAssembly for increased mining power [37]. However, there is a major limitation to cryptojacking: the average page visit, and therefore mining time, is generally limited to a few minutes [13].

Extended periods of mining will significantly heat up a machine. This thermal stress can cause damage to the hardware components if cooled insufficiently [1], or significantly increase power consumption to keep up with computation and cooling demands [77]. These side-effects of the malware result in financial damages for organizations. But as companies implemented smarted detection techniques, malicious miners implemented smarted evasion techniques too. In addition to generic detection evasion techniques, cryptomining specific strategies had to be implemented. Examples of these techniques include limiting the computing power taken up by the malware to prevent the host system from noticeably slowing down, or pausing mining operations when users open resource monitoring applications such as the Windows Task Manager. However, one mechanism of the malware is impossible to hide. At some point, the malware needs to communicate with the cryptocurrency network to retrieve information or to submit blocks.

In this thesis, a framework for detecting cryptominers on the network level is introduced. It aims to make as few assumptions as possible, and avoids using static knowledge that might become irrelevant as cryptominers change or adapt to current detection methods. Section 2 introduces background information on cryptomining. Section 3 covers the methodology. Section 4 presents the results. Section 5 discusses the results. Section 6 describes observations made during the research. Section 7 outlines future work. Section 8 discusses related work. Section 9 gives the conclusion.

Chapter 2

Background

2.1 Network Traffic Flows

There are several ways of describing the data streams between computers. However, experts and literature tend to disagree on details or interpretations. To avoid confusion, I will use the definitions used by Narang, Hota & Venkatakrishnan [47]. They used two levels of abstraction for communications: *conversations* and *flows*. The former describes the most basic case, where all traffic between two IP addresses is grouped together. The latter describes the 5-tuple that is most commonly used: the two IP addresses, the two ports, and the transport layer protocol. *Conversations* do not differentiate between traffic from separate applications on the two endpoints, while *flows* tries to make a distinction. As *flows* look at the transport protocol and ports, this can be viewed as shallow packet inspection or stateful packet inspection.

2.1.1 UDP

The User Datagram Protocol (UDP) [56] is a stateless communication protocol that provides bare-minimum connection capabilities. A UDP message contains a small header with the source port, the destination port, the packet length and a checksum value. This is enough to route the message over the internet, but also means that any control or verification mechanisms have to be handled by the applications on the endpoints, as reliable delivery is not guaranteed in the protocol.

UDP connections are ideal for one-way traffic, where the sender does not necessarily care about loss of messages, or for time-sensitive communication where messages become irrelevant once newer information is received. Common uses include DNS or DHCP traffic, where conversations consist of a single request and a single response message, or audio and video traffic, where consistency and continuity are more important than completeness.

2.1.2 TCP

The Transmission Control Protocol (TCP) [57] keeps track of the current state of the connection and provides reliable and ordered transmission of the data. It takes care of splitting data streams into segments that are small enough to be transmitted. It also keeps track of any missing or duplicate messages. For messages that do not arrive in time, retransmission is requested. The

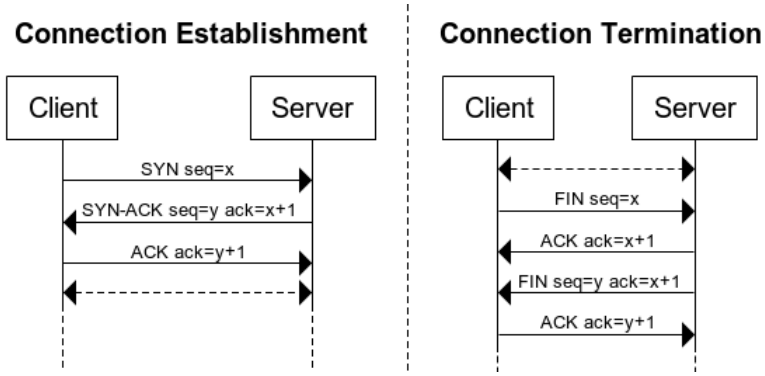

Figure 2.1: Schematic overview of TCP connection establishment and termination handshakes. Note that the client and server use different sequence numbers, and acknowledge messages using the last received number from the other party.

transmission speed and message buffer is also adjusted to compensate for the apparent network congestion. Messages that arrive out of order are sorted before delivery to the application.

These features require some overhead. This starts with the three-way handshake to ensure synchronisation between the two parties. The client sends a SYN message with a random sequence number to the server. The server acknowledges the client's SYN and sequence number, and sends its own sequence number. Finally, the client acknowledges the server's sequence number. Both parties now know the first sequence number used by the other, which is increased with each following message. During transmission, received messages are acknowledged using their number. This allows TCP to detect missing messages and signal retransmission requests. Finally, another handshake is performed to synchronise the termination of the connection. Both parties send and acknowledge each others' FIN message. A brief overview of the handshakes is shown in Figure 2.1.

2.2 Cryptocurrencies

The cryptomining process consists of a few general stages that are similar for most cryptocurrencies. A high-level overview of this process for Bitcoin will be given, but differences do exist between cryptocurrencies.

2.2.1 Bitcoin

First, a transaction is made by listing inputs and outputs. The inputs are references to unspent funds owned by the user. Ownership of these funds can be proven if the public key of the user was included in the previous transaction, and the funds have not been referenced in newer transactions. The outputs list the public keys of recipients and the amounts of funds to be transferred. An additional output can be included to return change to the user. Left-over funds are claimed by miners processing the transaction. See Figure 2.2 for a visual representation.

Transactions are bundled in blocks. To link blocks together, a reference to the previous block is included. See Figure 2.3 for a visual representation. At this point, the mining process begins. In order for a block to be accepted, the hash of the block needs to be of a valid format. In the

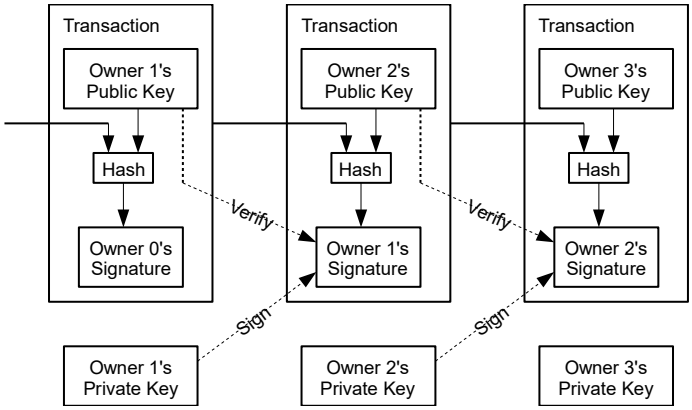

Figure 2.2: Visual representation of transaction chains. The public key references the new owner. Ownership is verified with public keys from previous transactions. Unspent funds have not been referenced in newer transactions. Transactions with multiple inputs reference multiple previous transactions. Transactions with multiple outputs reference multiple public keys. Image by Satoshi Nakamoto [46]

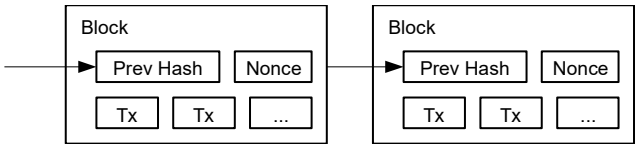

Figure 2.3: Blocks with the included transactions and the hash of the previous block. The nonce is used to influence the hash of the current block, which must satisfy the difficulty requirement. The process of trying various nonces is known as mining. Image by Satoshi Nakamoto [46]

case of Bitcoin, the first n characters of the hash need to be zeroes. To achieve this, a nonce is included in each block. Miners compute the hash of the block for various nonces. The more zeroes are required in the hash, the less valid hashes exist and thus the harder it is to find a nonce that satisfies the requirement. It then serves as proof of work, as others can trivially verify that the block now satisfies the difficulty requirement. The difficulty is adjusted to ensure that the entire mining network finds such a nonce every ten minutes on average. This means that difficulty increases as more miners join, and decreases as miners leave the mining network. Rewards are handed out to the miner that mines a block. This gives miners an incentive to add more computing power, increasing their share in the mining network and thereby increasing their chances of mining a block. Once a block has been mined, it is published and added to the blockchain. The mining network then starts mining the next block.

2.2.2 Altcoins

Since the release of Bitcoin in 2009, many more cryptocurrencies have been invented. All these alternative coins, commonly abbreviated to *altcoins*, share some characteristics in one way or another. However, a few coins have introduced innovative concepts that are worth noting. The list

below features a few randomly selected coins that introduced noteworthy ideas to cryptocurrencies. Note that many more altcoins have been invented, with many introducing their own novel features or creative innovations. However, listing them all would go far beyond the scope of this thesis.

- Litecoin [80] (7 October 2011): considered to be the first successful altcoin, it made small changes to the Bitcoin. Most notably, it replaced the hashing algorithm with scrypt, introducing memory requirements that dedicated hardware could not yet handle. This was the first effort to prevent hardware solutions from dominating mining operations of a cryptocurrency.

- Dash [17] (18 January 2014): the X11 hashing algorithm was introduced, which uses a sequence of 11 different hashing algorithms. Dedicated hardware would need to become significantly more complex to handle X11 efficiently. Additionally, all 11 hashing algorithms would need to be broken simultaneously for the X11 algorithm to be broken.

- Monero [74] (18 April 2014): the CryptoNote protocol introduced both unlinkability and untracability of transactions, hiding the identity of both sender and receiver. It also introduces the CryptoNight hashing algorithm, which is an adaptation of scrypt as used by Litecoin. However, it deals with optimizations and configurations that still made GPUs more effective than CPUs. Not only does this halt dedicated hardware, but also GPU mining rigs that were built for their performance over normal CPUs.

- Ethereum [81] (30 July 2015): provided smart contracts that can execute arbitrary code in an isolated sandbox, run by community members. It uses the Ethash proof-of-work algorithm and Keccak-256 hashing to create tasks that require a large amount of memory, resisting the development of dedicated hardware.

- Zcash [28] (28 October 2016): introduced individual optional anonymity for both parties, through zero-knowledge proofs known as zk-SNARKs. It also used the Equihash algorithm, which is based on the Generalized Birthday Problem and is a different approach than most coins take. It still relies on the fact that a lot of memory is required, preventing development of dedicated hardware.

2.3 Cryptomining Protocols

Chances of mining a block are relatively low, and getting a significant share of the computing power in the network is infeasible. This is why miners have grouped together in mining pools. They cooperate in an organized fashion that can be used to minimize duplicate computations and increase effectiveness of the pool. Profits from mining a block are shared among all participants in the pool, usually proportional to the computing power contributed to the pool by the participant.

The Bitcoin Wiki lists five protocols that are used for pooled mining [73]: Bitcoin Binary Data Protocol, BitPenny, getblocktemplate, getwork, Stratum Mining Protocol. At the time of writing, only the getblocktemplate protocol (GBT) and the Stratum protocol are still in use. Both GBT and Stratum use JSON Remote Procedure Calls for communication between the client and the pool server, and Stratum even uses large parts of the message specification of GBT. The major difference lies in the fact that GBT uses HTTP requests, while Stratum uses direct TCP connections.

2.3.1 getwork

The basic protocol first used for pooled mining, is the getwork protocol [15]. Even though it is no longer used, its successors still follow the same design principles. Getwork uses JSON Remote Procedure Calls to request a block to mine, or submit the result of a computation. Messages are sent as HTTP requests and responses. An empty request signals the server that a new task should be sent back. A block of 256 bytes encodes the task that the miner should work on, and another 64 byte block encodes the target value.

Miners can add a header to the request with a list of non-standardized extensions that they support. The server can add the corresponding headers to the response in order to better manage the miner. Extensions can be used to (temporarily) redirect the miner to a different server, instruct a miner to work on a partial task, or manage how the parties communicate.

Because of the little control that could be exercised by the server, the bad scalability, and the unofficial attempts at fixing various issues, the getwork protocol was replaced by the getblocktemplate protocol (Section 2.3.2).

2.3.2 getblocktemplate

In 2012, the getwork protocol was replaced by the GBT protocol [14]. It was designed to incorporate the unofficial extensions into a standard format as a list of identifiers, while keeping the JSON format. In addition, the tasks distributed to the miners contain more information and options that allow the miner to have more control over the mining process.

The GBT protocol does not send a specific task for the miner to work on, but gives the transactions that the miner needs to work with. The server can mark some transactions to be required, forcing the miner to include them in the block. A miner is then free to choose which of the optional transactions will be included. If the server marks the template as mutable, a miner is also allowed to alter the template parameters or add transactions. Because of the options and transactions, GBT messages will almost always be larger than getwork messages.

Miners are allowed to ask for new transactions at any time. In practice, miners often request work even if their current task is still valid and has not yet been invalidated by the server. Since each transaction includes a fee for the miner who processes it, keeping the task up-to-date with the maximum amount of transactions also maximizes the reward in case a miner is able to generate a block.

One drawback of the GBT is that it is still based on HTTP requests, like getwork. Even though miners can create as much work as they can handle, they are also responsible for maintaining connection to the server. It is not possible for the server to push a message to the miner, although tricks such as HTTP Long Polling provide workarounds. This was fixed with Stratum, which is described in Section 2.3.3.

2.3.3 Stratum

The Stratum protocol [49] was first introduced as a custom protocol for a mining client, but shared many similarities to getwork and GBT. The main difference is that GBT replaced the HTTP connections with TCP connections. This gets rid of the tricks and workarounds that used to be necessary, which simplifies communications and allows the server to push messages to miners whenever necessary.

The GBT protocol is used internally, but with a few slight modifications. First of all, only transaction hashes are included instead of full transaction details, and all transactions have to be processed. This makes messages significantly smaller, but also takes away the option for miners to pick transactions. The assumption is that most miners will want to maximize the fees from mining, automatically accepting all transactions. The second notable modification, is that miners can subscribe to notifications from the server and authorize multiple workers on a single connection. In practice, this means a single connection can serve multiple independent miners.

2.4 Machine Learning

In machine learning, there are two general approaches to training a model: supervised and unsupervised. The difference lies in whether there is knowledge about the correct output used during the training process.

In unsupervised learning we do not take any knowledge into account, whether this is available or not. Instead, the records in a data set are compared with each other and some metric for similarity has to be defined. From that metric, a method for splitting the data set has to be defined. But without guidance on what is right, the configuration of unsupervised algorithms has a direct influence on the results. Usually, the algorithm is trained and results are interpreted by humans. Parameters are then tweaked in an effort to nudge the outcome to better match the expected results, although this process does introduce an inherent bias.

In a supervised learning environment, the true labels of the records can be used to calculate the performance of a model. Improvements can be measured, allowing for optimization. However, the performance metric has a strong influence on how the model will be trained. Optimizing for different metrics can result in very different models. Regardless, varying types of learning can be used. Some algorithms may try to calculate absolute rules, while others incrementally adjust parameters.

In this project, labels are available to the algorithms. Therefore, only supervised learning algorithms will be described in the rest of this section. Section 2.4.1 describes the K-Nearest Neighbors algorithm, Section 2.4.2 describes the Random Forest algorithm, Section 2.4.3 describes AdaBoost, Section 2.4.4 describes the Naive Bayes algorithm, and Section 2.4.5 describes Support Vector Machines.

2.4.1 K-Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm [2] is an intuitive clustering algorithm. When classifying a record, the distance between it and all records from the training set are calculated. The distance can be calculated in a number of different ways using different metrics, but the most common metric is the sum of the euclidean distances per attribute. After calculating the distances, the closest k records are selected. The classes of the selected records are used in a vote. A number of ways exists to cast votes, a common method being the simple majority vote. Weighted methods often base the weight of the vote on the distance between the records. In some situations, it is possible for a record to receive equal scores for multiple classes. KNN classifiers should define how to handle such cases.

2.4.2 Random Forest

To understand the Random Forest algorithm [27], it is essential to understand the underlying principle of the Decision Trees. A Decision Tree takes the data set and splits it based on an attribute. The attribute is chosen based on information gain (Section 2.5.1) to maximize the classification score. It may then recursively split each part further based on other attributes. The decisions on how to split the data set may be encoded and visualized as branching nodes in a tree, and leaves encoding the class decision. Note that a decision tree may be arbitrarily large, and might be pruned to generalize the classifier and encode the majority class in a leaf.

The Random Forest classifier extends the concept of the Decision Tree classifier by combining the decisions from multiple randomly initialized Decision Trees. While the individual Decision Trees might come to different classifications, the Random Forest returns the majority vote among the trees.

2.4.3 AdaBoost

AdaBoost [19] has some similarities with the Random Forest classifier that might help understand how it works. Instead of building complex and independent Decision Trees, AdaBoost uses weighted and dependent decisions. The algorithm starts by picking the attribute that maximizes information gain, which can be seen as a decision tree with only one decision. The classification performance is calculated and the decision is given a weight. Then, the weight of each record is updated based on its current weight and the score of the last decision. Incorrectly classified records increase in weight, and correctly classified records decrease in weight. The next iteration of the algorithm once again looks for the decision that maximizes information gain, but now with respect to the weights of the records. The decision is recorded, the score is calculated and the weights are updated again. The final classifier takes a majority vote with the weight of each decision taken into account.

2.4.4 Naive Bayes

The Naive Bayes classifier [41] classifies records based on statistical knowledge of other records. Per class, the probability that a record belongs to this class is calculated, with the highest probability being used to classify the record. First, all records belonging to a class are selected. For each attribute, the percentage of selected records with the same label as the record under classification is calculated. These percentages are multiplied, along with the percentage of records in the entire data set belonging to the selected class. This gives the probability that the record under classification belongs to the selected class. The probability is not yet normalized, which is irrelevant for classification but can be done to calculate the confidence that the record belongs to the selected class.

The Naive Bayes classifier is formally defined as $p(C_i|x) = \frac{p(C_i)p(x|C_i)}{p(x)}$ with x as feature vector. Note how the denominator does not depend on any class, visualizing that normalization is not necessary for classification. Also note that $p(C_i)$ and all individual elements of both $p(x|C_i)$ and $p(x)$ can be precomputed, significantly reducing the cost of classification.

Typical Naive Bayes classifiers make use of discrete attributes, making it trivial to calculate the class probabilities per attribute. Continuous attributes can be discretized using binning, where values are mapped to intervals at the cost of accuracy as the bins introduce artificial clustering of the values. However, it is possible to estimate the probability more accurately using Gaussian probability distributions [32].

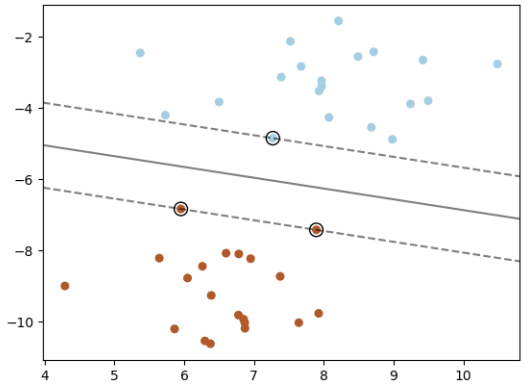

Figure 2.4: Example of the optimal separation line for a support vector machine on a two-dimensional data set. Image by the scikit-learn developers [53]

2.4.5 Support Vector Machine

Support Vector Machines [11] are based on the intuitive sense that plotted data can be split into groups using lines or (hyper)planes. The optimal case would split the data, but also maximize the distance to all data points. Records are classified based on what side of the line or (hyper)plane they are on. This principle works well on high-dimensional data, but it is computationally intensive to find a solution as the (hyper)plane for an n dimensional data set has $n - 1$ dimensions. In addition, hyper-parameters for the training algorithm itself need to be tuned, making training even more expensive.

The algorithm starts off with a random guess at the best line or plane to separate the data. The parameters are trained iteratively by computing the error of the separation plane, and adjusting accordingly. Calculating the error for all samples, especially if many dimensions are involved, can be costly. An alternative method is the Stochastic Gradient Descent, that computes the error for a random subset of records and adjusts based on this partial calculation. In practice, this method requires more iterations to converge, but reaches an optimum in less time. In this thesis, both a standard Support Vector Machine and a Support Vector Machine using Stochastic Gradient Descent were used.

Multidimensional cases are difficult to visualize or imagine, but the basic principle remains the same for any number of dimensions. If the data set would be one-dimensional, the records could be plotted on a number line where the optimal separation would be somewhere in between the means of the groups. For the two-dimensional case, as visualized in Figure 2.4, a line separates the groups. Note that it does not need to be linear, but could be polynomial at the increased cost of having more parameters to tweak (see Figure 2.5). Finally, the three-dimensional case would have a plane or surface that separates groups. Support Vector Machines can also be used for multi-class data, which is also visualized in Figure 2.5.

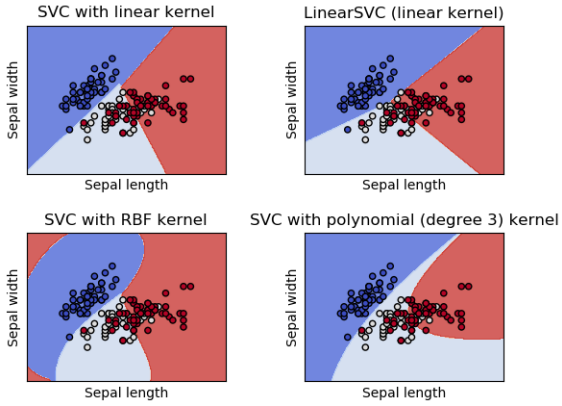

Figure 2.5: Example of the optimal separation line for a support vector machine on a two-dimensional data set with multiple classes. Both linear and polynomial separations are visualized. Image by the scikit-learn developers [53]. Note that *SVC* stands for *Support Vector Classification*.

2.5 Dimensionality Reduction

The curse of dimensionality [6], as it is commonly known, notes that achieving statistical significance becomes harder when the number of dimensions in the data set increases. Combining this knowledge with the fact that some machine learning algorithms require statistical significance to make accurate predictions, as well as the significant increase in computation time that occurs for many algorithms, and it is no surprise that there is a need to minimize the data while keeping the most relevant information. In order to achieve this, several strategies and techniques exist. In this section I will discuss information gain (Section 2.5.1), Principal Component Analysis (Section 2.5.2) and Correlation Feature Selection (Section 2.5.3).

2.5.1 Information Gain

The metric of information gain describes the change in knowledge about a data set after splitting it based on some arbitrary decision. We start by looking at the Shannon entropy [63] of our data set. It is defined as $Entropy = -\sum_{i=1}^{n} P_i \log_2 P_i$, and can be seen as an impurity measure for the set. Intuitively, when picking a random item from a high purity set, we have a high probability of picking the majority class. And when picking a random item from an evenly distributed set, we have a lower probability of correctly predicting which class we will pick. This can be translated to classification problems. When classifying records, we can group records by label and calculate the impurity of the prediction by looking at the original classes. When picking a random item of a certain label, we hope to have a high probability of finding an item of the corresponding class. In order to achieve such predictions and groupings, all decisions should aim to minimize entropy, further refining the sets until we are satisfied with the purity of our labels. This is done by using Kullback-Leibler divergence [38], also known as information gain, which is defined as $Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_V|}{|S|} Entropy(S_V)$. This compares the purity of the

original set and the new subsets, and weighs each subset by its relative size. Any decision that improves the overall purity of the predictions results in a positive value, namely the information gained with the decision. Algorithms often pick the decision that maximizes the information gain to most efficiently improve results with a single decision, and define lower bounds on the information gain to prevent the number of decisions from exploding without significantly improving predictions.

2.5.2 Principal Component Analysis

Principal Component Analysis [20] redefines a data set as a set of linear combinations of attributes, known as principal components. These components represent the major axes of variance in the data set, and can be used to transform linearly dependent attributes into linearly independent combinations. Intuitively, a person's height and weight are correlated, meaning that defining a healthy weight is impossible without taking their height into consideration. But two people of the same height could have a different weight, as well as two people with the same weight having a different height. PCA will find this correlation and transform it into the first principal component, and transform the distance from this correlated combination into the second principal component. Imagine such a scenario when looking at Figure 2.6, and it becomes clear that the second principal component says something about the distance from the expected weight or height.

First, the data set is centered on the mean and attributes are scaled to unit variance. Then the points are projected on a line through the origin and the sum of squared distances from the projected point to the origin is calculated. The first principal component is the unit vector that aligns with the line that maximizes the distance, which correlates to the axis with the greatest variance. Each following principal component is then orthogonal to the previous ones, each describing a decreasing amount of variance. When reducing dimensionality, selecting the first n principal components ensures the maximum amount of information is kept.

2.5.3 Correlation Feature Selection

Correlation Feature Selection is a method described by Mark Hall [26]. The basic principle is to select the feature that correlates most with the learning labels, but least with all other features. Intuitively, duplicate features show equal correlations to the learning labels but do not add value to the data set. Similarly, closely correlated features (such as distance in kilometers and distance in miles) still add dimensions without efficiently describing new variance in the data set. It uses a heuristic called merit, which is defined as $M_S = \frac{k\overline{r_{cf}}}{\sqrt{k+k(k-1)\overline{r_{ff}}}}$. M_S is the merit of feature subset S, k is the number of features in subset S, $\overline{r_{cf}}$ is the mean feature-class correlation for features f in S, and $\overline{r_{ff}}$ is the mean feature-feature correlation for features f in S. Looking at the formula, we see that a feature set with high feature-class correlations would maximize the numerator. Simultaneously, adding too many features or features that show significant correlations to each other, maximizes the denominator and thereby the cost of the feature set. One can start with an empty feature set and grow the set by adding the feature that maximizes the merit, or start with the full feature set and shrink it by deleting the feature that minimized the merit. These strategies are known as forwards selection and backward elimination respectively. Depending on what correlation metric is used, or what strategy is implemented, it is possible to obtain different feature sets.

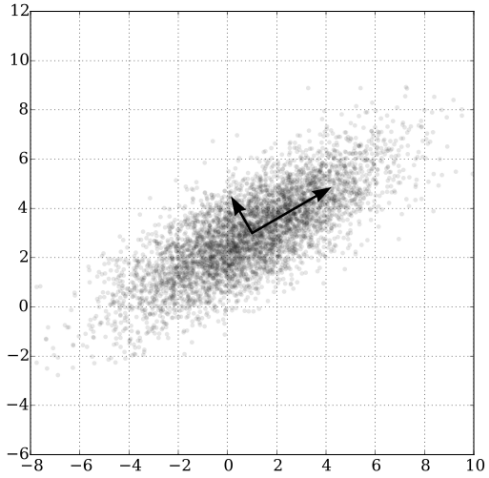

Figure 2.6: An example of a principal component analysis, with the first principal component showing the linear combination of attributes with the greatest variance. In dimensionality reduction, the last component is discarded to reduce the amount of data while keeping as much variance as possible. Image by Nicols Guarn [25]

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

Table 2.1: Standard confusion matrix for binary classification. Rows represent predicted classes and columns represent actual classes. Correct and incorrect classifications are found on the diagonals.

2.6 Performance Metrics

The performance of a classifier will always be quantified in terms of correct and incorrect labeling. The most basic form is binary classification, for which the possible classes are Positive and Negative. For both classes, a classifier can make a correct and incorrect prediction, resulting in four scenarios. This is commonly visualized as a Confusion Matrix as shown in Table 2.1. While training classifiers, the results from the confusion matrix are commonly condensed into a single value metric that focuses on two or more of the metrics in the matrix. In this section, some of these scoring metrics and their drawbacks will be discussed. The information in this section is based on work by Powers [58].

2.6.1 Precision

Precision describes the ratio between true positives and predicted positives. Intuitively, precision is the percentage of predicted positives that are actual positives. Precision is defined as

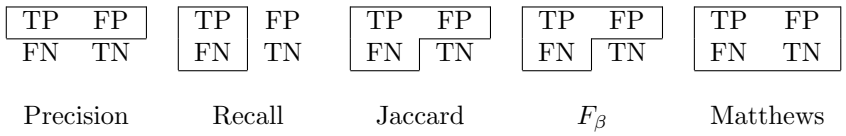

Table 2.2: Relations between the confusion matrix and scoring metrics visualized. See Table 2.1 for an explanation of the confusion matrix.

$precision = \frac{TP}{TP+FP}$. Precision is most useful when the cost of false positives is significantly higher than the cost of false negatives. In terms of this miner detection, this metric would show how many flagged systems turn out to contain cryptomining software after inspection. Optimizing for this metrics would mean minimizing the amount of systems unnecessarily checked for cryptominers. The downside of this metric is that precision does not give insight into how many actual positives were missed. Perfect precision can be achieved by a single correctly predicted positive, regardless of the number of false negatives.

Table 2.2 displays how the precision score relates to the confusion matrix and other scoring metrics.

2.6.2 Recall

Recall is quite similar to precision. However, recall looks at the true positives and actual positives. This can be interpreted as the percentage of actual positives that is predicted as positive. The formula is recall $\frac{TP}{TP+FN}$. Contrary to precision, recall is most useful when the cost of a false negative is significantly higher than the cost of a false positive. In terms of mining detection, recall would describe the percentage of active miners that is found and flagged. Optimization would focus on minimizing the hidden costs of miners that remain active and undetected. Recall does not provide insight into the number of false positives, leading to the scenario where unnecessarily predicting only positives would always lead to a perfect recall score.

Table 2.2 displays how the recall score relates to the confusion matrix and other scoring metrics.

2.6.3 Jaccard Similarity Coefficient

Expanding upon the precision and recall metrics, the Jaccard Similarity Coefficient looks at all relevant true positives and all incorrect predictions. Jaccard similarity is calculated using $\frac{TP}{TP+FP+FN}$. As both false positives and false negatives can result in unnecessary costs, this metric can be interpreted as the percentage of costs spent on correct miner detections in comparison to the total amount of costs caused by miners and miner prevention. However, Jaccard similarity does not differentiate between the cost of false positives and false negatives. For miner detection, false positives might cost less than false negatives which result in miners remaining undetected.

Table 2.2 displays how the Jaccard Similarity Coefficient relates to the confusion matrix and other scoring metrics.

2.6.4 $\mathbf{F}_\beta$

The F_1 score is the harmonic mean of the precision and recall scores. The formula is $F_1 = 2 * \frac{precision*recall}{precision+recall} = \frac{2*TP}{2*TP+FP+FN}$. The generalized form is $F_\beta = (1+\beta^2) * \frac{precision*recall}{(\beta^2*precision)+recall} =$

$\frac{(1+\beta^2)*TP}{(1+\beta^2)*TP+\beta^2*FN+FP}$. The parameter β allows control over the balance between precision and recall over the score. This can be used to model the balance in costs between false positives and false negatives and optimize for a more realistic score of the detection method. Intuitively, recall is β times as important as precision, or precision is $\frac{1}{\beta}$ times as important as recall. Note that $F_0 = precision$ and $F_\infty = recall$. However, The F_1 metric does not take into account true negatives and can result in over-optimistic scores.

Table 2.2 displays how the F_1 score relates to the confusion matrix and other scoring metrics.

2.6.5 Matthews Correlation Coefficient

On most data sets, the Matthews Correlation Coefficient is similar to the F_1 metric. The formula is $MCC = \frac{TP*TN-FP*FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$. It takes into account all four classification types as well as any imbalances, and gives more information about the confusion matrix from a single score. Imagine a classifier that classifies all actual positives as negatives, and all actual negatives as positives. The F_β score would be 0, which shows the lack of true positives but can not show the performance on true negatives. The MCC would be -1, showing that the predictions are perfectly inversely correlated to the actual classes. As another example, imagine a perfectly random guessing on a perfectly balanced set of binary classes. The expected confusion matrix would be $TP = FP = FN = TN$, which results in an F_β score of 0.5. The MCC would be 0, which represents the random nature of the classifier. As a final example, imagine $TP = FP = FN, TN = 0$. In this case, F_β would still be 0.5 while the MCC of -0.5 would correctly show the loose inverse correlation. The drawback of the MCC is that there is no control over the influence of false positives and false negatives, which might be desirable from a business perspective.

Table 2.2 displays how the Matthews Correlation Coefficient relates to the confusion matrix and other scoring metrics.

Chapter 3

Methodology

This section describes the full process from collecting miner samples to the trained models. The process is split into three steps. The first step is selecting the miners, described in Section 3.1. The second step is the collection of data in Section 3.2. The third step is the preprocessing of the gathered data in Section 3.3. The fourth step is the diversification by applying several scaling and correlation methods in Section 3.4. The final step is the learning and validation of the models in Section 3.5.

3.1 Sample Selection

Miner samples were provided by VirusTotal [76] through an academic agreement. Temporary download access to an academic repository was granted, containing samples from October 2017 to July 2018. Sample selection was performed by analysing all reports in the repository provided by VirusTotal, using SQLite [68] and DB Browser for SQLite [54]. All detection names were extracted from the reports, filtered for unique entries, filtered once more for mentions of the terms *coin* or *mine*, and finally checked by hand for entries unrelated to cryptocurrency mining. The list of manually removed names can be found in Appendix A. The resulting list of relevant detection names was used to filter for reports mentioning any of the selected names. The corresponding samples were used for data collection. All samples were provided as encrypted zipped (7z) files.

The results of the sample selection can be found in Section 4.1.

3.2 Data Collection

A Windows 10 laptop with an Intel Core i7-7600U processor and 32 GB DDR4 RAM was used as host for the data collection lab. VirtualBox [48] was installed using instructions from VBox-HardenedLoader [75] to harden the installation and make sandbox detection more difficult for the samples. We refer the reader to Section 6.2.1 for remarks on the use of VBoxHardenedLoader. Inside VirtualBox, an internal network was created for the VMs so that they can not connect to the host system or the internet. Three VMs were created: a base analysis VM, an FTP server, and a gateway. Each of these will be discussed in Sections 3.2.1- 3.2.3. An overview of the virtual network is shown in Figure 3.1.

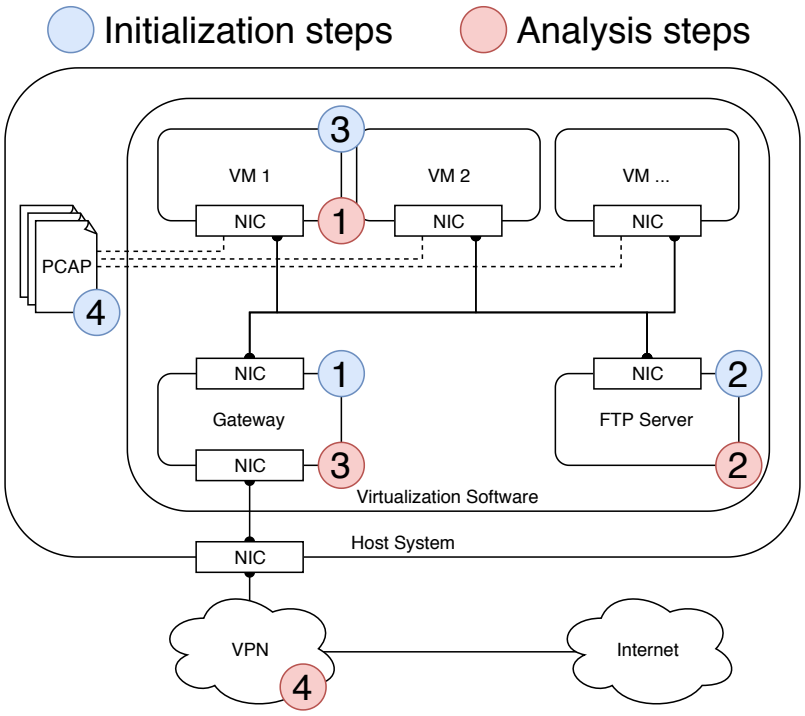

Figure 3.1: The data collection lab is a virtual environment that is controlled by virtualization software. A gateway and an FTP server provide infrastructure for the analysis VMs. Network traffic is logged to PCAP files by the virtualization software. The gateway routes all traffic over a VPN connection, to prevent reputation damage for the network that the host is connected to.

A script on the host system, called the controller, managed the virtual network and all VMs. First, the gateway and FTP server were started and given time to finish loading. Then, n linked clones of the base analysis VM were created (e.g. VM1, VM2, etc). VirtualBox was configured to capture all traffic from the analysis VMs to unique local PCAP files on the host. This acted as a virtual hardware tap on the virtual network interfaces, and was invisible to the VMs. Next, the controller started the analysis VMs. After waiting for fifteen minutes, the analysis was assumed to be complete and all analysis VMs were forcibly shutdown if they were still active. Finally, the analysis VMs were deleted. The controller then restarted the cycle back at the cloning from the base analysis VM. Refer to Section 6.2.2 for remarks on the PCAP collection.

The results of the data collection can be found in Section 4.2.

3.2.1 Analysis VM

A Windows 7 VM with two cores, 2 GB RAM, and a 100GB HDD was created as a base analysis VM. These hardware specifications were based upon results from Al-Khaser [18], which will be discussed later in this paragraph. Next, the VM was configured using instructions from VBox-HardenedLoader [75], with one adaptation. The use of the Tsugumi driver was abandoned, and

the *hidevm* script was edited to reflect this change, as the driver kept crashing. After applying the script, the hardening was verified using Al-Khaser [18], which reports the results of typical VM detection techniques. The results can be found in Section 4.2.1.

After installing Windows 7 on the VM, Windows Defender, Windows Updates, and User Account Controls were disabled. 7-Zip [51] was installed to unpack the samples during analysis. A script was set to run on startup. First, the script waited for the system to fully load. Then, the script connected to the FTP server to download and delete a single sample. The script then closed the FTP connection, unpacked the sample, and ran it. The script waited ten minutes, before forcibly shutting down the VM. Finally, the script would disable itself before running the sample. This prevented the script running twice and downloading a second sample, if the first sample would restart the VM. Refer to Section 6.2.3 for remarks on the script.

Several frameworks were installed on the VM to help samples to run, and common applications were installed to make the VM resemble a normal desktop. All frameworks and applications were installed using Ninite [61]: 7-Zip, Adobe Air, Adobe Shockwave, Apache OpenOffice, Apple iTunes, Document Foundation LibreOffice, Dropbox, GIMP, Google Backup and Sync, Google Chrome, IrfanView, Microsoft OneDrive, Microsoft Silverlight, Microsoft Skype, Mozilla Firefox, Mozilla Thunderbird, Notepad++, Spotify, TeamViewer 14, VideoLAN VLC Media Player, WinRAR, dotPDN Paint.NET, pdfforge PDFCreator.

3.2.2 FTP Server

A Windows 10 VM was created and updated to ensure all the latest security updates were installed and Windows Defender was up-to-date. SMB was turned off, and the Internet Information Services, FTP Server, and Web Management Tools were turned on. An FTP site was created and the site folder was shared with a new local user. Credentials of this user were used by the analysis VMs to log into the FTP site. A firewall exception was made for FTP connections. Finally, all miner samples were placed in the FTP folder. A snapshot of the FTP server was made for debugging purposes, resetting the lab, and bringing back samples that were taken during analysis. Refer to Section 6.2.3 for remarks on the FTP server.

3.2.3 Gateway

The gateway was created using pfSense [40], and has two network interfaces. One interface is connected to the internal network of VirtualBox, and the other interface is bridged to the network interface of the host machine. All sample traffic was routed through an OpenVPN tunnel to a VPN provider. This was done to prevent reputation damage for the network that hosted the lab because of the malicious traffic that was generated. The gateway is visualized in Figure 3.2.

Note that the gateway was built to support UDP traffic. For example, Whonix [70] could be used as an alternative to pfSense with out-of-the-box routing through Tor. However, Tor does not support UDP traffic [31]. This could result in loss of traffic. In this experiment, VPN provider Cryptostorm [12] and their pfSense tutorial were used. But any VPN provider with support for OpenVPN connections could be used to set up the gateway.

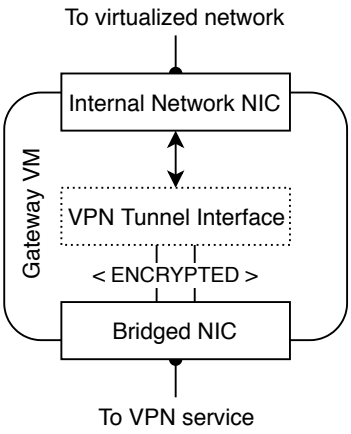

Figure 3.2: All traffic from the data collection lab was routed through a VPN tunnel interface, created by OpenVPN [30]. Routes were set up to ensure no traffic could bypass the interface, and all traffic on the outside of lab was encapsulated in the VPN tunnel. This prevented reputation damage for the network that hosted the lab.

3.3 Preprocessing

Preprocessing was done as part of a pipeline built using Jupyter Notebooks [36] through the Anaconda Distribution [3]. The same holds true for the diversification in Section 3.4 and the modeling in Section 3.5.

In the preprocessing steps, the gathered network traffic is converted into a format that can be used in the model learning step. First, the labeling of the traffic is discussed in Section 3.3.1. It is then split into flows, which is discussed in Section 3.3.2. Next, statistical features calculation is discussed in Section 3.3.3. Finally, the mixing of sample traffic and a reference set is discussed in Section 3.3.4. A visual explanation of Sections 3.3.1- 3.3.3 is shown in Figure 3.3.

The results of the preprocessing can be found in Section 4.4.

3.3.1 Labeling

To handle the labeling of the large volume of traffic, the Snort IDS [10] was used. To instruct Snort on what traffic should be considered malicious, it was given two rulesets. The first is the Snort Registered rules. This ruleset is available on the Snort website for all users who create an account. The second is the Emerging Threats Ruleset [71], an open ruleset that is contributed to, and maintained by, the security community.

For each original PCAP file, Snort generated a new PCAP file that contained only the packets that triggered detection rules. All attributes were kept as the packets were copied by Snort. The epoch timestamps, or the time logged by the network interface when the packet was received, was used during feature generation in Section 3.3.3 to label the data.

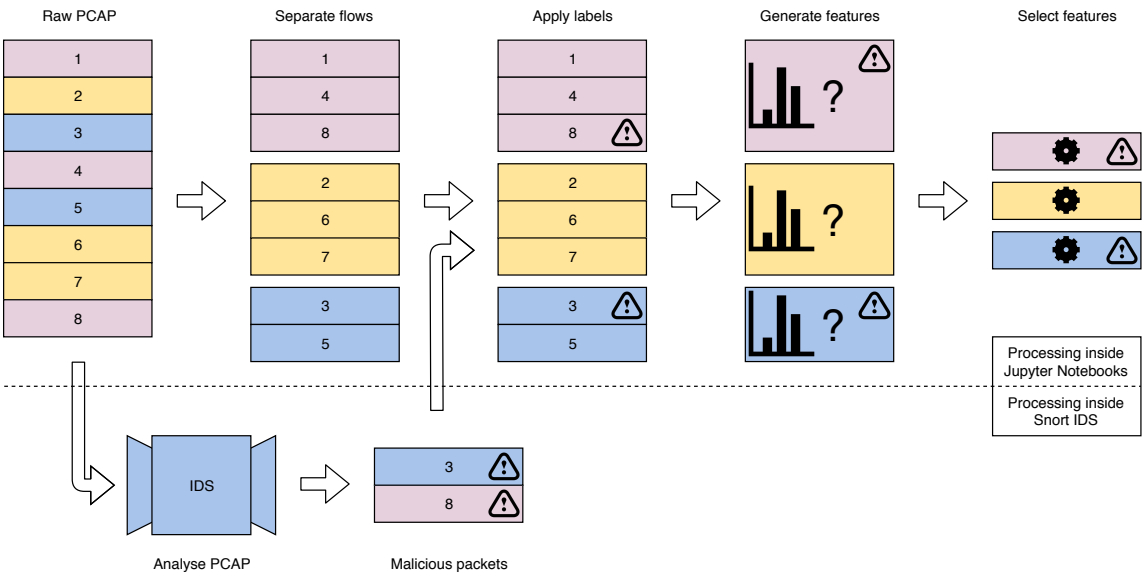

Figure 3.3: To gain insight into the preprocessing, the steps have been visualized. For convenience, all packets are numbered by their original order and colored by flow. Packets flagged by Snort as malicious are matched by number, or timestamp in the actual program. Flagged packets are used to label the entire flow as malicious, which in turn is used as feature in the model learning step.

3.3.2 Flow Separation

The PCAP files, consisting of network packets, were processed using the Python library Scapy [7]. Packets were separated into conversations and flows. Conversations were defined as the stream of packets between two IP addresses. An example of a conversation might be (198.51.100.0, 203.0.113.0). Flows were defined as the stream of packets that shared a tuple of IP addresses, ports and transport-layer protocol. An example of a flow might be (198.51.100.0, 9001, 203.0.113.0, 12345, TCP). Conversations represent two machines communicating with each other. However, it could be the case that there are multiple applications talking to each other on those two machines. Flows represent two applications communicating. Keep in mind that there are many scenarios where these assumptions do not hold. As such, they should be treated as approximations.

Conversations and flows represent levels of looking at connections between two endpoints, which means that there are two notations for the same tuple. For instance, conversation (198.51.100.0, 203.0.113.0) is the same as conversation (203.0.113.0, 198.51.100.0), as both show the same two endpoints communicating. In the same way, flows (198.51.100.0, 9001, 203.0.113.0, 12345, TCP) and (203.0.113.0, 12345, 198.51.100.0, 9001, TCP) are the same. For flows, the port numbers must remain associated to the same IP addresses and the protocol must remain the same. If the ports are switched, it could mean that two completely different pairs of applications are communicating. This is also the case when the protocols do not match in two tuples.

The notation of the tuple was determined by whichever situation was observed first. The first address belonged to the sender of the packet, who initiated the connection, and the second address belonged to the receiver. Responses were matched to the existing tuple and added to the corresponding flows. All packets belonging to a flow were grouped together. Similarly, all flows belonging to a conversation were grouped together.

3.3.3 Feature Generation

To describe the characteristics of the conversations and flows, many different statistical features were calculated. For instance, a flow could be described by the number of bytes received or the duration of the connection. Some statistics could only be computed for conversations, such as the total number of used ports or the average number of packets per flow. The features were based on work by Bekerman et al. [5], but adapted to fit my definitions of flow and conversation, as well as fit the scope of the project. A list of features is available in Appendix B.

Two non-statistical features were included. The detections from Section 3.3.1 were used to label flows, indicating that one or more packets were flagged as malicious. Flows with and without flagged packets were labeled with a one and a zero respectively. Similarly, conversations with or without flagged flows were also labeled with a one or a zero.

3.3.4 Mixing with Reference Data

As the data collection lab from Section 3.2 did not simulate a realistic scenario with human interactions, a reference data set was used. The Intrusion Detection Evaluation Dataset (CICIDS2017) [64] provided data of a simulated small company network over the course of a week. The data set includes one day of activity without simulated attacks, to be used as reference. The PCAP file of that day was preprocessed as described in Sections 3.3.1- 3.3.3.

The labeling step was also performed for the CICIDS2017 data. Simulated activity could have resulted in malicious traffic. For instance, randomly browsing the web could have led to malicious

advertisements being served to the machine, or even cryptominers in the browser. Although the preprocessing was not guaranteed to catch all malicious traffic, it was used to successfully sanitize the data further.

All benign conversations from the CICIDS2017 data set and all malicious conversations of the data collection lab were combined into one data set for further use.

3.4 Diversification

Diversification was done as part of a pipeline built using Jupyter Notebooks [36] through the Anaconda Distribution [3]. The same holds true for the preprocessing in Section 3.3 and the modeling in Section 3.5.

After the preprocessing steps from Section 3.3, there were two decisions that could still have an impact on the modeling steps in Section 3.5. The first decision to be made, was on how to scale the data. The second, was on how to calculate correlation for the Correlation Feature Selection as described in Section 2.5.3. Instead of picking any specific option based on intuition, the decisions were incorporated as a diversification step to explore different combinations of choices.

The results of the diversification can be found in Section 4.5.

3.4.1 Scaling

Before the data was used, several scaling functions from scikit-learn [53] were applied to create variations of the data set. This was done to be able to compare the impact on the final model scores. The following functions, as defined by scikit-learn, were used:

- Unscaled: no scaling was applied.
- Min-Max: each feature was scaled to a range of $[0, 1]$.
- Normalize: each record was transformed to a unit vector.
- Robust: each feature was centered on the mean, and the 25-75 percentile range was scaled to unit variance.
- Scale: each feature was centered on the mean and scaled to unit variance.

Note that the normalization function applies to records and not to features. That means that the number of generated features from Section 3.3.3 has an impact on this scaling method.

3.4.2 Correlation

For each of the scaling functions of Section 3.4.1, the data was stored in Pandas [42] DataFrame objects. These have built-in functionality to calculate correlation using the Pearson correlation coefficient [52], Kendall rank correlation coefficient [33], and Spearman's rank correlation coefficient [67]. This resulted in a total of fifteen correlation matrices, that were used as input for the feature selection in Section 3.4.3.

3.4.3 Feature Selection

Irrespective of the scaling method from Section 3.4.1, the number of generated features from Section 3.3.3 can easily overwhelm machine learning algorithms and result in bad performance and/or

extremely long computation times. Using the correlation matrices from Section 3.4.2, Correlation Feature Selection was performed as described in Section 2.5.3. For each of correlation matrices, a subset of features was selected. These subsets were used in Section 3.5.

3.5 Modeling

Modeling was done as part of a pipeline built using Jupyter Notebooks [36] through the Anaconda Distribution [3]. The same holds true for the preprocessing in Section 3.3 and the diversification in Section 3.4.

The classifiers as described in Section 2.4 were created using the scikit-learn [53] library. Each classifier was trained on scaled data sets from Section 3.4.1, using each of the corresponding subsets of features from Section 3.4.3. Each training session was repeated using K-Fold cross validation, splitting the data set into 10 subsets and using each as a test set for the other 9. This process was then repeated for hyperparameter training, where grid searching was used to test every combination of parameters in predefined ranges. Both K-Fold cross validation and grid search were also taken from scikit-learn. Finally, the results were plotted using the Matplotlib Python library [29]. The plots are shown in Section 4.6, and discussed in Section 5.

For each of the classifiers, these parameters ranges were tested:

- K-Nearest Neighbors: $[1, 50]$ neighbors
- Random Forest: $[1, 100]$ trees, $[1, 10]$ maximum tree depth
- AdaBoost: $[1, 200]$ estimators
- Naive bayes: -
- Support Vector Machine: $[10, 1000]$ maximum iterations
- SVM with Stochastic Gradient Descent: $[10, 1000]$ maximum iterations

No undersampling or oversampling was performed to correct the imbalance between positives and negatives in the data set, as this would intentionally skew results of the classifiers and most often leads to worse results. The results of the modeling can be found in Section 4.6.

Chapter 4

Results

4.1 Sample Selection

Selecting samples was done in several steps, each time further narrowing down the number of samples. The process is described in Section 3.1. The number of samples retained at each step is displayed in Table 4.1.

When downloading the repository, there were many more reports than samples available. In some cases, only the sample was available. But in many cases, only the report was available. All samples that were missing either the report or the sample were excluded. Out of the 331.528 reports, 215.451 were selected because the corresponding samples were provided in the repository.

After extracting the 820.042 antivirus engine labels from these reports, they were explored for terms that could help find all cryptominer samples. The terms *coin* and *mine* were chosen based on the assumption that these could also be used to find terms such as *bitcoin* or *miner*, and initial searches seemed to return relevant antivirus engine labels. The 5.077 found labels were checked for irrelevant terms such as *CoinStealer* and *Imminent*, that do not relate to cryptocurrency miners. For more on the excluded labels, we refer the reader to Appendix A. The 4.819 relevant labels matched 16.928 reports. The corresponding samples were gathered and categorized, the result of which is shown in Table 4.2. Of the 16.928 samples, 6.597 were executable file types.

Processing step	**Sample count**
Reports in repository	331.528
... with sample	215.451
Unique detection labels	820.042
... using 'coin' or 'mine'	5.077
... after manual checks	4.819
Selected samples	16.928
Of executable types	6.597
Tested in sandbox	1.702
... that ran successfully	1.653

Table 4.1: Number of samples after each selection step as described in Section 3.1

File type	Sample count	File type	Sample count
7ZIP	3	Pascal	1
Android	11	RAR	35
C	2414	Text	3942
C++	3219	Win32 DLL	360
CAB	1	Win32 EXE	1702
DOS EXE	22	Windows Installer	2
ELF	42	XML	10
GZIP	7	ZIP	90
HTML	4717	unknown	350

Total sample count: 16.928
Duplicate samples: 29

Table 4.2: The sample count per file type in the set of selected samples. Note that some samples consist of source code that has not been compiled.

4.2 Data Collection

This section covers two separate results from the data collection phase described in Section 3.2. First, the result of the hardening process as described in Section 3.2.1 is discussed in Section 4.2.1. Second, a brief overview of the collected data is given in Section 4.3.

4.2.1 Sandbox Detection Mitigation & Verification

As described in Section 3.2.1, the analysis VMs were hardened using VBoxHardenedLoader [75] to make sandbox detection more difficult for the miner samples. This process was verified using Al-Khaser [18], of which the results have been recorded in Table 4.3. In the final setup, only the script was used for stability reasons. Note that this did mitigate several detection methods, but was still far from the score of a plain Windows 7 installation on a laptop. This meant that there was a possibility that miner samples were able to detect that they were running in an isolated environment, which potentially influenced the way in which the samples operated.

	Mitigation	**Detection points**
Baseline	VM, no mitigation	45
Testing	Only driver	39
	Only script	31
Unstable setup	Driver & script, base VM	25
	Driver & script, linked clone	29
Final setup	Only script, base VM	31
	Only script, linked clone	35
Reference	Windows 7 installed on laptop	9

Table 4.3: Number of sandbox detection triggers by Al-Khaser [18] after each step of detection mitigation as described in Section 3.2.1 using VBoxHardenedLoader [75]. The driver used by VBoxHardenedLoader was too unstable for the final data collection setup. For reference, Windows 7 installed on a laptop without any virtualization still produced 9 sandbox detection triggers.

4.3 General Overview of Collected Data

Due to time constraints, only the Windows 32-bit executable samples of Section 4.1 could be tested, limiting the experiment to 1.702 samples. Some samples crashed the testing environment repeatedly, making it impossible to complete the data collection session. In the end, network traffic of 1.653 samples was collected using the methodology explained in Section 3.2. These generated a combined total of 871 MB of network traffic.

4.4 Preprocessing

For a detailed description of the preprocessing process, we refer the reader to Section 3.3.

Snort labeled 1,44 MB of traffic as malicious, resulting in 3.222 flagged flows. These were spread across 148 conversations that spanned a total of 16.820 flows. Of the flows, 569 used TCP and 16.251 used UDP. For conversations with TCP flows, the average number of TCP flows was 14,23 with a standard deviation of 31,67. For conversations with UDP flows, the average number of UDP flows was 118,62 with a standard deviation of 75,41. Yet the average number of source ports in a conversation was 112,97 and the average number of destination ports was 2,41. This means the analysis VMs opened a large number of ports, which connected to a small number of ports on the server corresponding to each conversation. 107.468 packets were sent by initiating parties, while 50.767 packets were sent by responding parties.

The flagged conversations of the sandbox and the unflagged conversations of the CICIDS2017 data set resulted in 32,91 MB and 456,39 MB of features respectively. The combined set consisted of 198.453 flows across 22.617 conversations. With 3.222 out of 198.453 flows having been flagged as malicious, there were 60.6 times more benign flows than malicious ones.

4.5 Diversification

The data was scaled and correlation matrices were calculated following the functions from Section 3.4. For each of the combinations of scaling and correlation, Correlation Feature Selection was performed. The selected features for each combination are shown in this list:

- **Unscaled & Pearson**: flow_bytes_A_var, flow_bytes_min
- **Unscaled & Kendall**: cw_udp_packets_A_thirdQ, cw_udp_packets_thirdQ, cw_src_ports_flows_ratio
- **Unscaled & Spearman**: cw_udp_packets_A_thirdQ, cw_src_ports_flows_ratio
- **Min-Max & Pearson**: flow_bytes_A_var, flow_bytes_min
- **Min-Max & Kendall**: cw_udp_packets_A_thirdQ, cw_udp_packets_thirdQ, cw_src_ports_flows_ratio
- **Min-Max & Spearman**: cw_udp_packets_A_thirdQ, cw_src_ports_flows_ratio
- **Normalize & Pearson**: cw_udp_packets_A_stdev, cw_udp_packets_A_sum
- **Normalize & Kendall**: cw_flow_bytes_stdev, cw_flow_duration_thirdQ, cw_flow_packets_A_var, cw_udp_packets_stdev, cw_udp_flows, cw_udp_bytes_A_var
- **Normalize & Spearman**: cw_flow_bytes_stdev, cw_udp_packets_stdev
- **Robust & Pearson**: flow_bytes_A_var, flow_bytes_min
- **Robust & Kendall**: cw_udp_packets_A_thirdQ, cw_udp_packets_thirdQ, cw_src_ports_flows_ratio

- **Robust & Spearman**: cw_udp_packets_A_thirdQ, cw_src_ports_flows_ratio
- **Scale & Pearson**: flow_bytes_A_var, flow_bytes_min
- **Scale & Kendall**: cw_udp_packets_A_thirdQ, cw_udp_packets_thirdQ, cw_src_ports_flows_ratio
- **Scale & Spearman**: cw_udp_packets_A_thirdQ, cw_src_ports_flows_ratio

To clarify the meaning of these features, they were listed below with a short explanation. The full list of features is available in Appendix B.

- **cw_flow_bytes_stdev**: The standard deviation of all bytes per flow in the conversation.
- **cw_flow_duration_thirdQ**: The 75th percentile, or third quartile, of the durations per flow in the conversation.
- **cw_flow_packets_A_var**: The variance of the packets sent by the initiator of each flow in the conversation.
- **cw_src_ports_flows_ratio**: The number of unique source ports in the conversation divided by the number of flows in the conversation.
- **cw_udp_bytes_A_var**: The variance of the bytes sent by the initiator per UDP flow in the conversation.
- **cw_udp_flows**: The number of UDP flows in the conversation.
- **cw_udp_packets_A_stdev**: The standard deviation of the number of packets sent by the initiator per flow in the conversation.
- **cw_udp_packets_A_sum**: The total number of UDP packets sent by the initiator in the conversation.
- **cw_udp_packets_A_thirdQ**: The 75th percentile, or third quartile, of the number of packets sent by the initiator per flow in the conversation.
- **cw_udp_packets_stdev**: The standard deviation of the number of UDP packets sent per flow in the conversation.
- **cw_udp_packets_thirdQ**: The 75th percentile, or third quartile, of the number of UDP packets sent per flow in the conversation.
- **flow_bytes_A_var**: The variance in bytes per packet sent by the initiator in the flow.
- **flow_bytes_min**: The minimum number of bytes in a single packet of the flow.

4.6 Modeling

All six classifiers were trained as described in Section 3.5, with K-Fold cross validation and hyperparameter tuning for the specified parameter ranges. The results for each classifier were plotted as follows: K-Nearest Neighbors in Figure 4.1, Random Forest in Figure 4.2, AdaBoost in Figure 4.3, Naive Bayes in Figure 4.4, Support Vector Machine in Figure 4.5, and SVM using Stochastic Gradient Descent in Figure 4.6.

For each classifier, the best performing combinations of scaling and correlating were:

- **K-Nearest Neighbors**: Scale & Pearson
- **Random Forest**: Robust & Pearson
- **AdaBoost**: Robust & Pearson
- **Naive Bayes**: Normalize & Pearson
- **Support Vector Machine**: Unscaled & Kendall
- **SVM using SGD**: Scale & Spearman

Confusion matrices for the best performing combinations were constructed by letting each classifier make predictions for all records in the data set. The matrices can be found in Table 4.4. Using the confusion matrices, the performance metrics of Section 2.6 have been calculated for the convenience of the reader. The metrics can be found in Table 4.5. For convenience, the number of errors per classifier are listed below:

- **K-Nearest Neighbors**: 6582 errors
- **Random Forest**: 2807 errors
- **AdaBoost**: 2839 errors
- **Naive Bayes**: 13069 errors
- **Support Vector Machine**: 15209 errors
- **SVM using SGD**: 20729 errors

The classifiers were compared in a pairwise manner to compute how many errors were made by one out of the two classifiers, and how many errors were made by both classifiers. The results are shown in Table 4.6 and Table 4.7 respectively. Out of the 3.222 malicious flows, sixteen flows were never correctly classified by any of the six selected best performing classifiers.

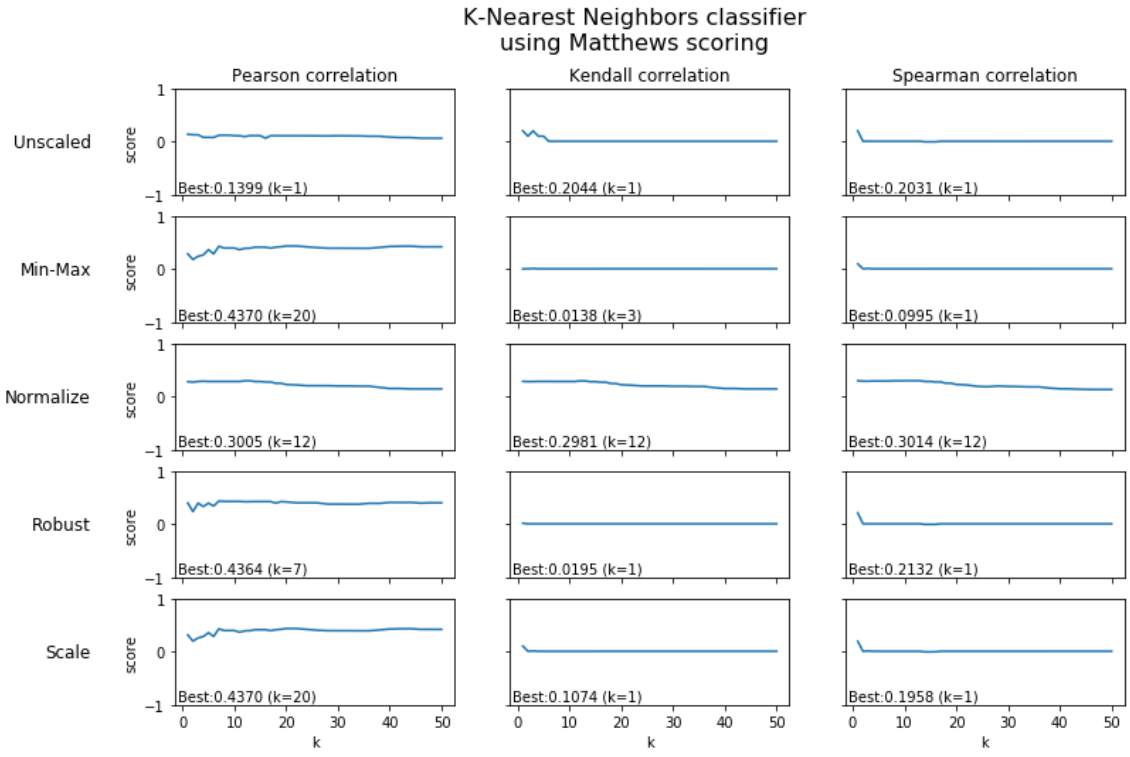

Figure 4.1: Scores for the K-Nearest Neighbors classifier. The Scale & Pearson set performed best (first column, fifth row).

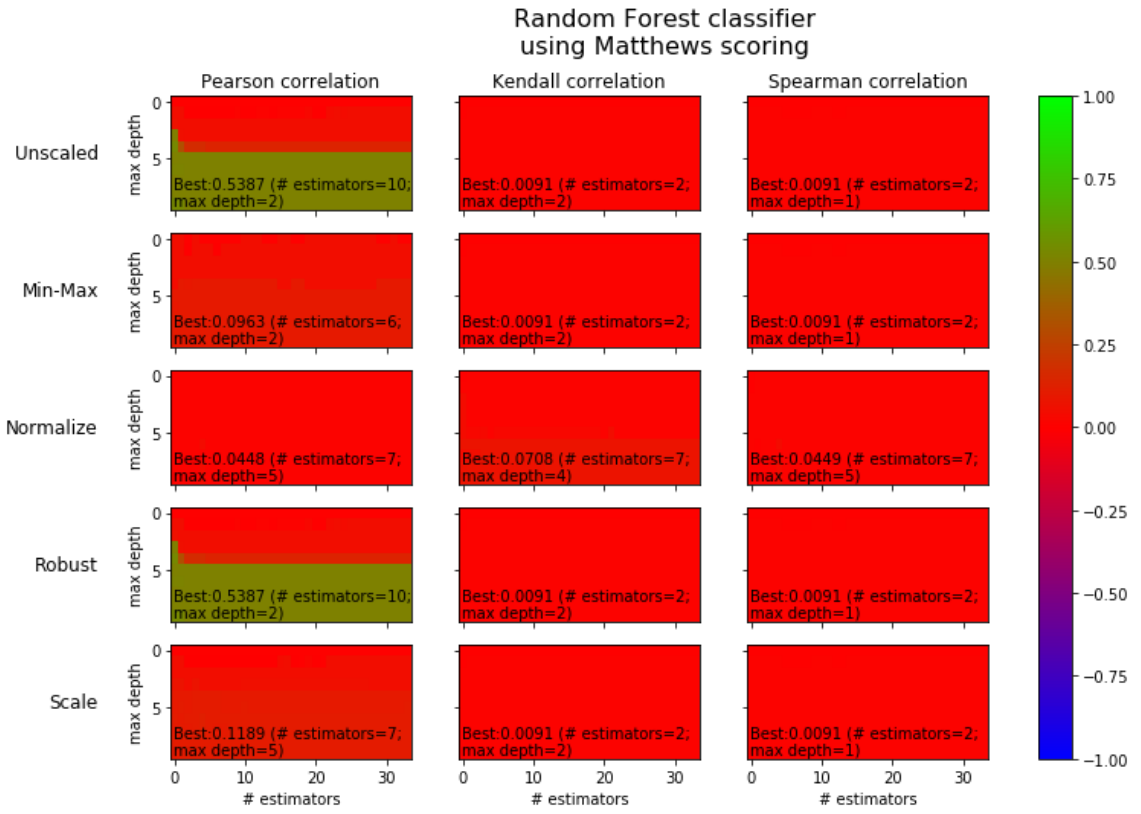

Figure 4.2: Scores for the Random Forest classifier. The Robust & Pearson set performed best (first column, fourth row).

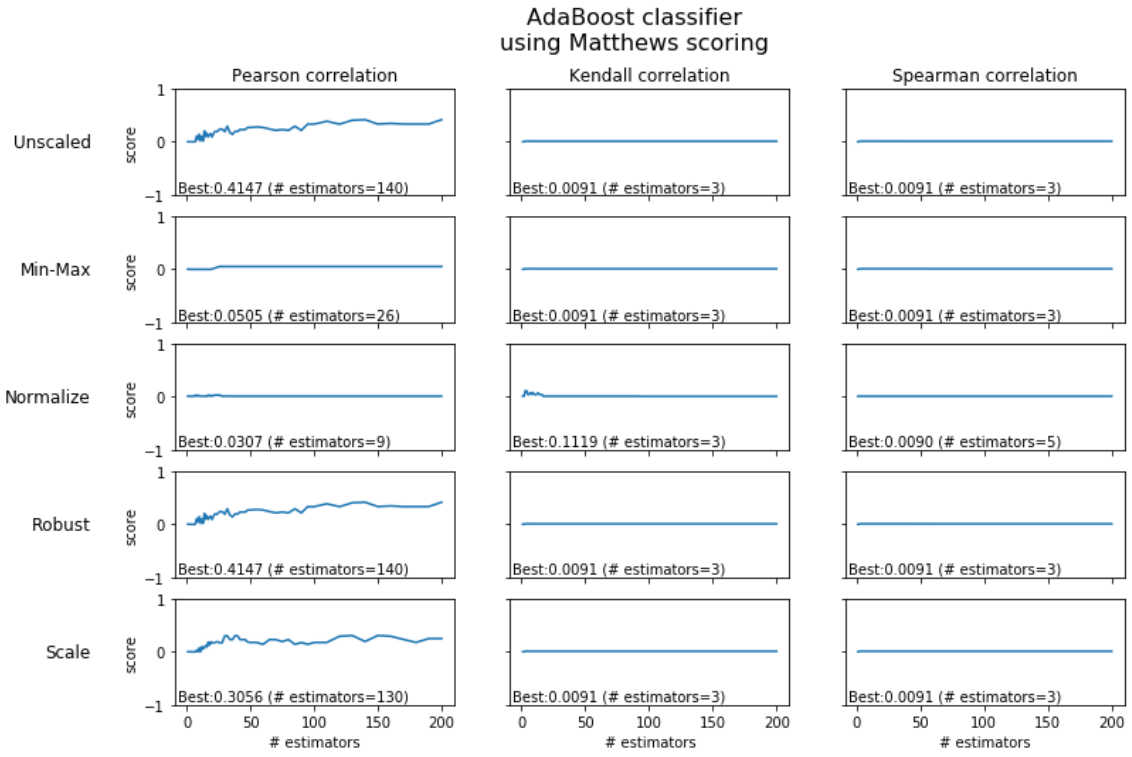

Figure 4.3: Scores for the AdaBoost classifier. The Robust & Pearson set performed best (first column, fourth row.

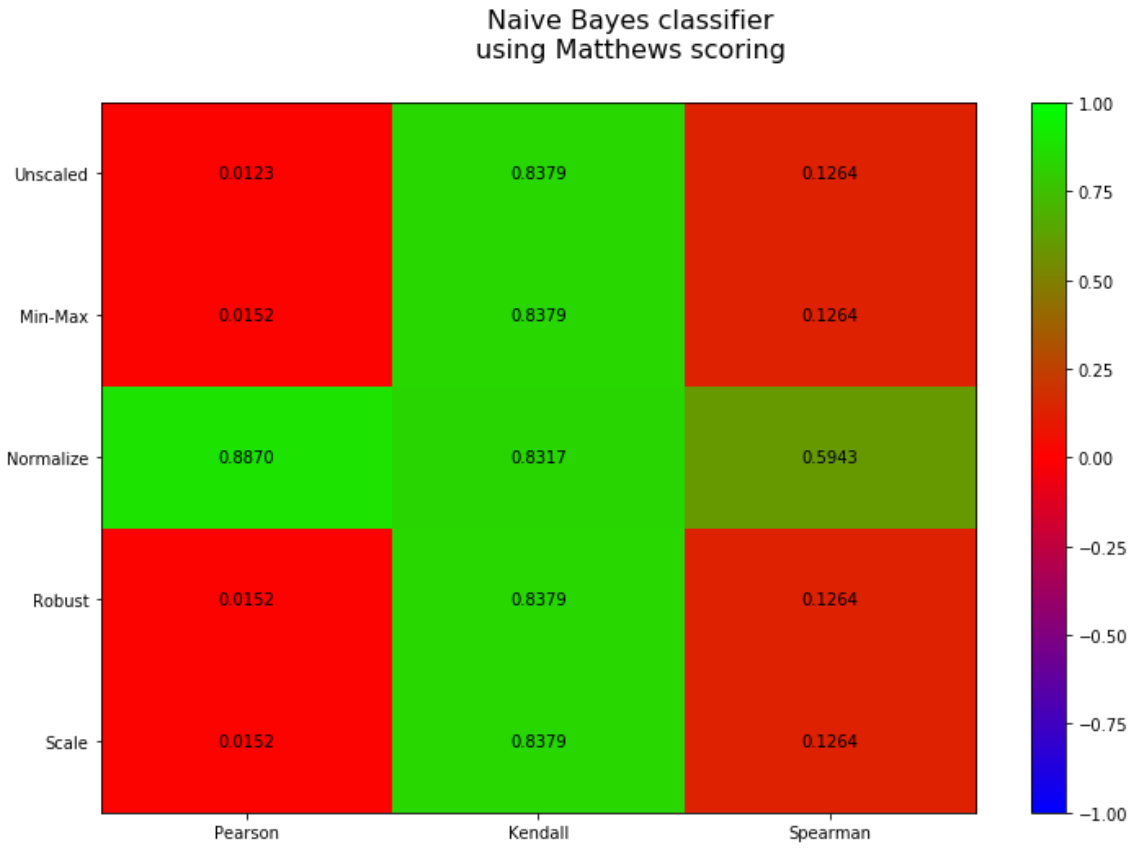

Figure 4.4: Scores for the Naive Bayes classifier. The Normalize & Pearson set performed best (first column, third row).

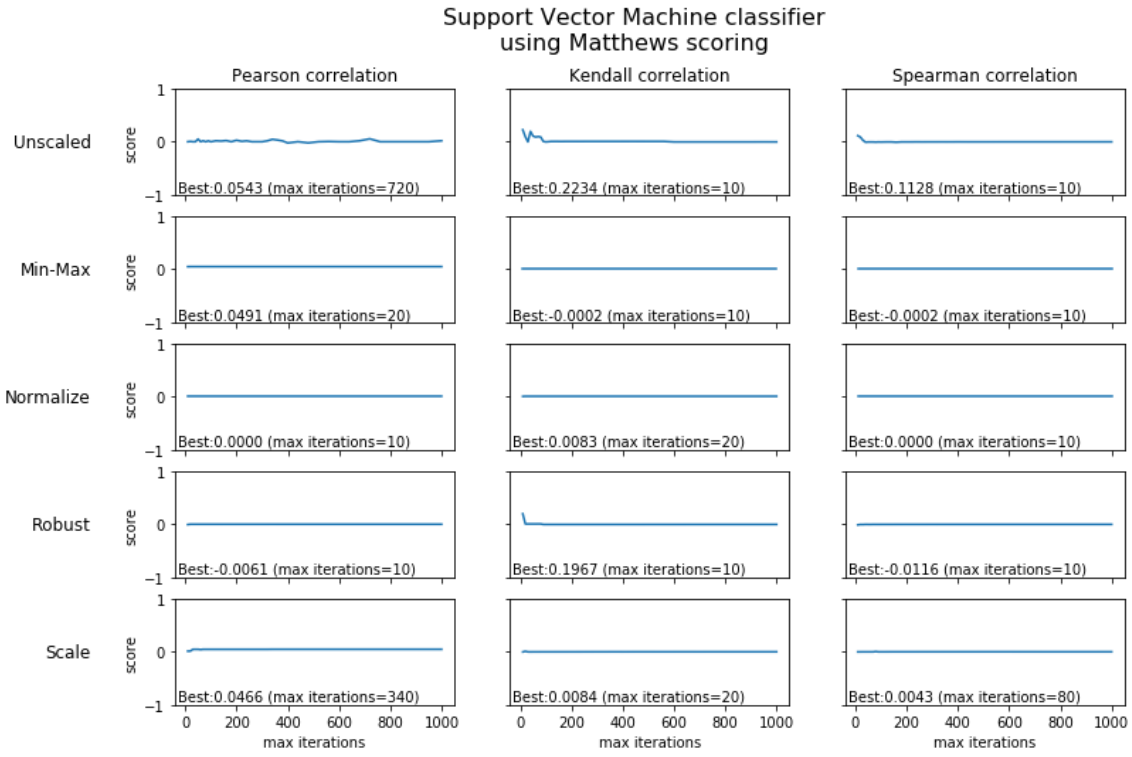

Figure 4.5: Scores for the Support Vector Machine classifier. The Unscaled & Kendall set performed best (second column, first row).

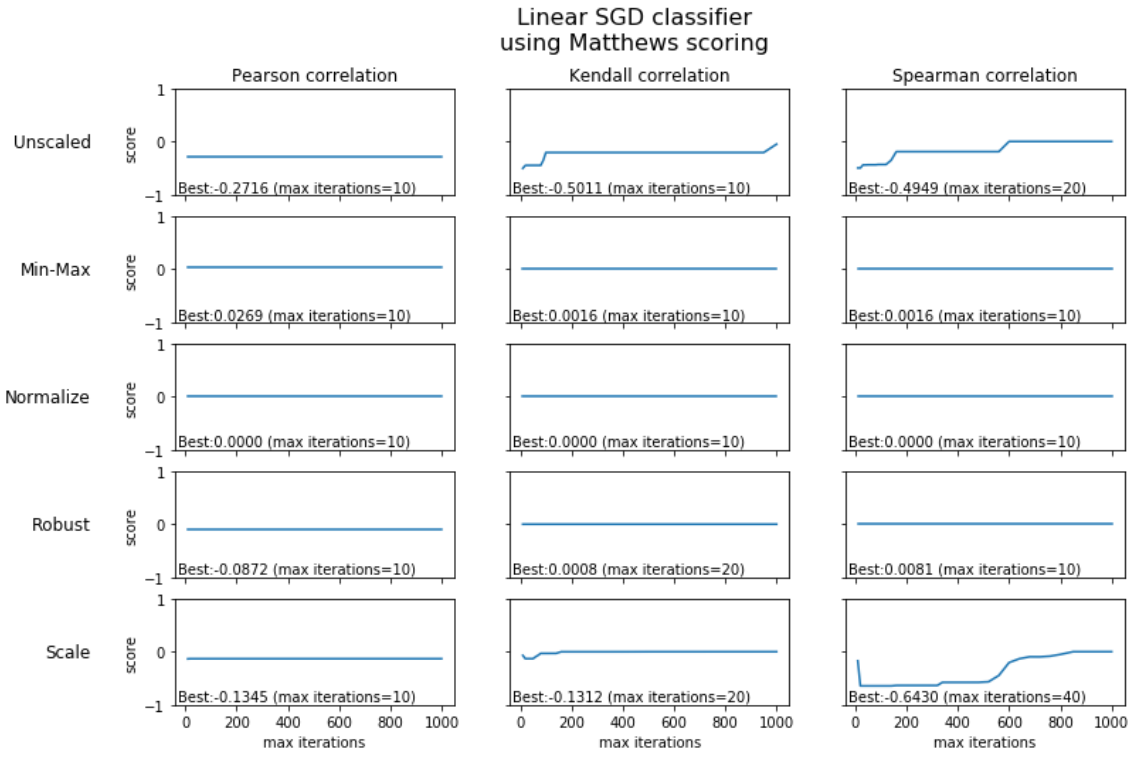

Figure 4.6: Scores for the SVM classifier using Stochastic Gradient Descent. The Scale & Spearman set performed best (third column, fifth row). Note the negative score for this set, which means the classifier performs better when inverting its predictions.

K-Nearest Neighbors

	TP	TN
PP	1972	5332
PN	1250	189899

Random Forest

	TP	TN
PP	1674	1259
PN	1548	193972

AdaBoost

	TP	TN
PP	1642	1259
PN	1580	193972

Naive Bayes

	TP	TN
PP	3161	13008
PN	61	182223

Support Vector Machine

	TP	TN
PP	3161	15148
PN	61	180083

SVM using SGD

	TP	TN
PP	39	174541
PN	3183	20690

Table 4.4: Confusion matrices of the trained classifiers for the best set of scaling and correlation. The terms True Positive (TP), True Negative (TN), Predicted Positive (PP) and Predicted Negative (PN) have been shortened. Note that the SVM using SGD performs better when inverting its predictions, essentially swapping the two rows.

	Accuracy	F1	Precision	Recall	Jaccard	Matthews	ROC AUC
KNN	0.9668	0.3747	0.2700	0.6120	0.9668	0.3925	0.8604
Random Forest	0.9859	0.5439	0.5707	0.5196	0.9859	0.5374	0.9796
AdaBoost	0.9857	0.5363	0.5660	0.5096	0.9857	0.5298	0.9784
Naive Bayes	0.9341	0.3260	0.1955	0.9811	0.9341	0.4225	0.9645
SVM	0.9234	0.2936	0.1726	0.9811	0.9234	0.3946	-
SVM using SGD	0.8955	0.2350	0.1333	0.9879	0.8955	0.3426	-

Table 4.5: Performance metrics of Section 2.6 calculated for each of the best performing classifiers. Note that scores of the SVM using SGD use the inverted predictions as mentioned in Table 4.4.

	KNN	Random Forest	AdaBoost	Naive Bayes	SVM	SVM+SGD
KNN	-	321	336	12282	14421	19941
Random Forest	4096	-	32	13053	15192	20712
AdaBoost	4079	0	-	13038	15177	20712
Naive Bayes	5795	2791	2808	-	2141	7682
SVM	5794	2790	2807	1	-	7261
SVM+SGD	5794	2790	2822	22	1741	-

Table 4.6: Comparison of errors between pairs of classifiers. Each value represents the number of records that the classifier of that row classified correctly, but the classifier of that column classified incorrectly.

	KNN	Random Forest	AdaBoost	Naive Bayes	SVM	SVM+SGD
KNN	-	2486	2503	787	788	788
Random Forest	2486	-	2807	16	17	17
AdaBoost	2503	2807	-	31	32	17
Naive Bayes	787	16	31	-	13068	13047
SVM	788	17	32	13068	-	13468
SVM+SGD	788	17	17	13047	13468	-

Table 4.7: Comparison of errors between pairs of classifiers. Each value represents the number of records that both classifiers predicted incorrectly.

Chapter 5

Discussion

Although the results in Section 4.6 show that some classifiers do indeed correctly classify many samples, it is important to take another look at Table 4.4. All classifiers generated over 1.000 false positives, with half even generating over 10.000 false positives. In a business case, false positives result in wasted effort trying to investigate whether an alert is a true positive. If we look at the Random Forest and AdaBoost classifiers, we see that about half of all alerts in this scenario would have been false positives.

Yet at the same time, false negatives could result in damages because infections were not spotted. This should draw the reader's attention to the confusion matrices of the Naive Bayes and Support Vector Machine classifiers. These classifiers found almost all true positives, albeit at the cost of a large number of false positives. If the cost of investigating false positives is relatively low, the Naive Bayes classifier could provide a solution that is fast to train and leaves little false negatives.

However, if the cost of investigating false positives is relatively high, another solution would be necessary. In this case, we can include Table 4.7 to search for an answer. The Naive Bayes classifier and the Random Forest classifier rarely make the same mistakes, which suggests that they could be used to complement each other. If we first take all positive predictions of the Naive Bayes classifier to find almost all true positives, and then apply the Random Forest classifier to those samples to reduce the number of false positives, we get confusion matrix in Table 5.1 and the performance metrics in Table 5.2.

Naive Bayes before Random Forest

	True Positive	True Negative
Predicted Positive	1624	5
Predicted Negative	1598	195226

Table 5.1: Confusion table after using Naive Bayes to select potential positives, and using Random Forest for the final decision.

Although there are ways to minimize the number of false positives or the number of false negatives, there are two important factors to keep in mind. First, this data set only approximates a real life scenario for companies looking to protect their networks. The CICIDS2017 data set from Section 3.3.4 only contains a single day of traffic, simulating a small company network. In reality, we could assume a network to be clean at some point in time, and monitor the network

	Accuracy	F1	Precision	Recall	Jaccard	Matthews
Combined	0.991923	0.669553	0.996931	0.504035	0.991923	0.705953

Table 5.2: Performance metrics after using Naive Bayes to select potential positives, and using Random Forest for the final decision.

from that moment forward. If we monitor the network for 100 days without any cryptominers on the network, the false positive rate in Table 5.1 would still generate 500 alerts. In addition, if the company network would be larger than the simulated one, we would expect more than five alerts to be generated per day. In such a small company as was simulated, one can easily imagine the number of alerts being far too high to be taken seriously on a daily basis.

The second factor to keep in mind, concerns the imbalance in the data set. We have added the results of 1.653 malicious samples, whether they successfully executed or not, to a single day worth of traffic of a small company. Once again, the assumption that a network is relatively clean for extended periods of time reinforces the principle that true company networks should generate far more benign traffic than malicious traffic. In reality, we assume networks to generate more varied traffic, which would only make training the classifiers more difficult. To approach the true balance of networks in our data set, we would either have to add great amounts of benign traffic or greatly reduce the number of miner samples. It is therefore highly likely that the current performance is far from the performance seen outside a laboratory environment.

Naturally, it is also necessary to note that there are ways to improve upon the current framework. There are many more features that could be computed in Section 3.3.3, some of which might greatly improve the performance of the classifiers. We tried to refrain from using the typical malware signatures that require pattern matching indicators, as malware could easily be altered to avoid detection once more. However, one could imagine less static features, such as the top-level domains in DNS requests, that might differ enough in malware traffic compared to normal company traffic. Or perhaps the length of the domain name would be a good feature. Additionally, combining multiple classifiers such as suggested earlier in this section, could prove to be useful as well. It could even be beneficial to use one classifier as a filter, like the Naive Bayes classifier, and use the output to train another classifier.

Chapter 6

Observations

This section documents observations made during the research, that did not fit anywhere else in the thesis. Nevertheless, they could provide insight and information for readers interested in replicating parts of this research.

6.1 Initial Exploration

Initially, a small malware lab was built and samples were randomly picked from reports that mentioned terms such as bitcoin or miner. This was done to explore the miner samples and gain general insight into the network traffic produced by these samples. Within VirtualBox, a REMnux [82] VM was created to serve as a simulated internet gateway. The internal network of VirtualBox was used to all samples were contained within a virtualized environment without connections to the host or the internet. VirtualBox was also configured to write all network traffic to a local PCAP file. Building such a small and contained network provided an ideal learning environment to become familiar with the malware and understand its behavior.

The small lab was not yet configured for automatic data collection. In order to create an initial data set, the list of selected names from Section 3.1 was used to find the sample with the highest detection rate per name, i.e. the sample that was detected by most antivirus engines. This sample set was filtered for unique entries and submitted to a publicly available Cuckoo Sandbox [24] by the Estonian CERT [9]. After the analysis was completed, the PCAP files were downloaded to test the initial versions of the preprocessing (Section 3.3), diversification (Section 3.4), and modeling (Section 3.5) stages with. Having a reasonably sized data set was useful in trying to debug the Jupyter Notebook and quickly finding packet types that were not yet being handled.

6.2 Data Collection

During the data collection phase described in Section 3.2, there were a few remarks to be made on the use of VBoxHardenedLoader (Section 6.2.1), the collection of PCAP files on the host system (Section 6.2.2), and the workings of the collection lab during the execution of the samples (Section 6.2.3).

6.2.1 VBoxHardenedLoader

The driver that was provided with VBoxHardenedLoader did mitigate some VM detection methods, as described in Section 4.2.1. However, the host machine would crash with a Windows blue screen. Whether the problem was related to VBoxHardenedLoader, to VirtualBox, or to Windows, is unknown. For stability reasons, the use of the driver was abandoned and only the provided scripts to change the VM configurations were used.

6.2.2 PCAP Collection

The collection of network traffic was done using built-in functions of VirtualBox. As all hardware for the VMs was emulated by VirtualBox, it was possible to let all traffic be logged to local PCAP files. This functionality was only available through the commandline-interface VBoxManage: `modifyvm [--nictrace<1-N> on|off] [--nictracefile<1-N> <filename>]`. As such, there was the added advantage that this method of logging network traffic did not require scripts on the analysis VMs or secure storage inside the virtualized network.

6.2.3 Data Collection Lab During Execution

Whilst running the data collection lab, the number of samples taken from the FTP server and the number of PCAP files collected were not running in lockstep as anticipated. Upon investigation, we noticed that some malware samples restarted the VM, which led to the automation script accidentally collecting another sample from the FTP server. At the same time, an unknown file was visible on the FTP server for a few seconds. This one incident revealed mistakes in our assumptions. First, we assumed that VMs would not restart. This was fixed by making the automation script remove itself at the same time as it started the miner sample. The second assumption, was that a fully updated Windows 10 machine would be safe as the FTP server in the hostile network. Fortunately, Windows Defender caught the file before it could be executed, and identified it as the Bluteal trojan, which abused the SMB protocol to try and infect the FTP server. This was fixed by disabling SMB on the FTP server, which we assume was turned on by default.

Having all infrastructure in the virtualized environment allowed us to take snapshots and quickly debug and restart the lab. Since this research had to rely on given labels as described in Section 3.1, there was no guarantee that the samples would indeed purely be cryptominers. This incident showed that it was well worth being prepared for samples to be unpredictable.

Chapter 7

Future Work

The current framework can be improved in four ways. First, the number of samples can be expanded by testing a wider variety of samples. As mentioned in Section 4.3, only the Windows 32-bit executable samples were tested due to time constraints. But as shown in Table 4.1, there were five times more executable files that could have been executed. Even then, that would only be 6.597 samples out of the 16.928 that were found in the academic repository. It would be interesting to collect the data from the other samples too. For instance, there are 4.717 HTML samples left unexplored, as well as 5.633 C and C++ samples that have yet to be compiled, and even 3.942 samples that were only labeled as being text.

The second improvement would be to try and create more realistic data sets. Section 5 already mentioned that the current data set features a day worth of network activity of a simulated small company, versus 1.653 malicious file samples. More benign traffic could provide a more diverse baseline, as well as decrease the significance of the malicious samples that are currently over-represented in the data set. This would likely make model learning significantly harder, but would also more realistically represent the performance of the classifiers in a real-world scenario.

The third improvement lies in the features that are being generated. As the features are selected using Correlation Feature Selection (Section 2.5.3), the most important features will be used to train the classifiers. Finding new features that correlate well to the samples increases the performance of the classifiers. However, unused features that have been implemented could still be useful in the future. If miner samples shift their behavior to avoid detection, previously unused features could become more important and get selected. And as mentioned in Section 5, features that rely less heavily on statistics while not focusing on static signatures, could still be reused when retraining the classifiers to account for changes in sample behavior.

The fourth and final improvement could be made by investigating the possibilities to combine several classifiers. One approach would be to create a voting scheme where multiple classifiers have to agree on the decision, as only sixteen flows were never correctly identified (Section 4.6). Perhaps, a staged approach such as suggested in Section 5 would result in a better performance, where classifiers are used to filter or split the data set before handing it off to the next classifier.

Chapter 8

Related Work

For a gentle but thorough introduction to the intricacies of Bitcoin, including security and privacy aspects, we refer the reader to Tschorsch & Scheuermann [72]. An in-depth investigation of illicit mining practices was performed by Pastrana & Suarez-Tangil [50].

MineGuard [69] uses hardware performance signatures associated with proof-of-work computations, to detect cryptominers on CPUs and GPUs with minimal overhead. Solanas, Hernandez-Castro & Dutta [66] used CPU, disk, and network utilization metrics to detect malicious behavior on VMs in a cloud environment. Barbhuiya et al. [4] used anomaly detection with similar features to detect the moment a VM started the illicit behavior.

Both Musch et al. [45] and Saad, Khormali & Mohaisen [60] show the wide-spread use of cryptominers in websites, and conclude that blacklists provide insufficient protection. Kharraz et al. [34] demonstrated that CPU utilization can be used in addition to other features, but generates a large amount of false-positives on its own. SEISMIC [78] provided a semantic signature-matching method by counting CPU instructions during the execution of WebAssembly modules. MineSweeper [37], which builds upon this principle, adds the detection of cryptographic primitives and excessive cache operations during execution.

BotHunter [22], BotSniffer [23] and BotMiner [21] are three similar methods for botnet detection. BotHunter correlates network connections between machines inside and outside a network to five infection phases. BotSniffer uses the principle that infected machines show strong similarities in communication with C&C servers. It detects potentially infected machines through clustering traffic characteristics, which means no prior knowledge of the malware is required. BotMiner takes this one step further by applying layered clustering to connections and network activity, making it capable of detecting both centralized C&C structures as well as P2P botnets. It also reduces the data footprint by 90% before clustering. In contrast, Morales et al. [44] identified unusual network behavior used by malware, with which it could be detected and classified. In 2011, Rossow et al. [59] monitored traffic of malware families for an hour (contrary to the usual two to ten minutes) and showed that it mainly generated DNS and HTTP traffic, which was further dissected. Bekerman et al. [5] proposed a malware agnostic approach by looking at network traffic from three levels of aggregation and generating 972 metrics, upon which part of this research was based. Their system was able to detect new malware samples up to four weeks before community-made detection rules appeared.

Chapter 9

Conclusion

As shown in the results in Section 4 and discussed in Section 5, it is possible to do cryptomining detection using only statistical features. The large feature set generated in Section 3.3.3 can be reduced using Correlation Feature Selection in Section 3.4.3. Searching through all combinations of scaling and correlating led to different optimal mutations for the various classifiers as shown in Section 4.6. Although the individual classifiers might not be usable in real-life scenarios, as shown in Table 4.4, combinations of classifiers could work towards usable systems as shown in Section 5.

Our framework does not need to be applied to all individual machines in a network, but can be used to monitor all machines at once. Like MineGuard [69], SEISMIC [78], and MineSweeper [37], we use behavior inherent to cryptominers that would be difficult to mask. However, we applied this concept to network level behavior instead of machine level behavior. BotHunter, BotSniffer and BotMiner need at least two machines in the network, where our framework is agnostic to the number of machines. We based our statistical approach on Bekerman et al. [5], but avoided features that could introduce static signatures. We showed that these additional features are not necessary for reaching recall scores of 98.11% or precision scores of 99.69%. Finally, we tested a variety of scaling and correlation methods to preprocess the data. Not only did we show the score graphs of the hyperparameter tuning, but also demonstrated the effects of the preprocessing methods. As far as the authors are aware, the effects of data preprocessing methods have never been discussed, or shown, in related works.

Further improvements could be made to this work, as mentioned in Section 7. Not all available samples could be tested, which leaves the question how well the framework performs on other types of cryptominer samples. In addition, the CICIDS2017 data set that was used as reference material to train the classifiers with, might not be sufficient to accurately simulate a real company environment. Furthermore, there are many options to add more features, which could potentially give better results. Finally, using combinations of classifiers as described in Section 5 shows potential improvement gains that are worth investigating.

Bibliography

[1] Ahmed Almubarak, A.: The effects of heat on electronic components. International Journal of Engineering Research and Applications **07**, 52–57 (06 2017). https://doi.org/10.9790/9622-0705055257

[2] Altman, N.S.: An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician **46**(3), 175–185 (1992). https://doi.org/10.1080/00031305.1992.10475879, `https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879`

[3] Anaconda, I.: Anaconda distribution, `https://www.anaconda.com/distribution/`, accessed on 24-03-2019

[4] Barbhuiya, S., Papazachos, Z.C., Kilpatrick, P., Nikolopoulos, D.S.: RADS: real-time anomaly detection system for cloud data centres. CoRR **abs/1811.04481** (2018), `http://arxiv.org/abs/1811.04481`

[5] Bekerman, D., Shapira, B., Rokach, L., Bar, A.: Unknown malware detection using network traffic classification. In: 2015 IEEE Conference on Communications and Network Security (CNS). pp. 134–142 (Sep 2015). https://doi.org/10.1109/CNS.2015.7346821

[6] Bellman, R.: Dynamic programming. Science **153**(3731), 34–37 (1966)

[7] Biondi, P.: Scapy, `https://scapy.net/`, accessed on 25-03-2019

[8] Bulavas, V., Kazantsev, A.: A mining multitool (2018), `https://securelist.com/a-mining-multitool/86950/`, accessed on 25-02-2019

[9] CERT-EE: Cuckoo Sandbox (2018), `https://cuckoo.cert.ee/`, accessed on 25-03-2019

[10] Cisco Systems, I.: Snort, `https://www.snort.org/`, accessed on 24-03-2019

[11] Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20**(3), 273–297 (Sep 1995). https://doi.org/10.1007/BF00994018, `https://doi.org/10.1007/BF00994018`

[12] cryptostorm: cryptostorm (2019), `https://cryptostorm.is/pfsense`, accessed on 31-05-2019

[13] Danaher, P.J., Mullarkey, G.W., Essegaier, S.: Factors affecting web site visit duration: A cross-domain analysis. Journal of Marketing Research **43**(2), 182–194 (2006). https://doi.org/10.1509/jmkr.43.2.182, `https://doi.org/10.1509/jmkr.43.2.182`

[14] Dashjr, L.: getblocktemplate - fundamentals (2012), https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki

[15] Dashjr, L.: Getwork - bitcoin wiki (2013), https://en.bitcoin.it/wiki/Getwork

[16] Desmond, C.: Bitcoins: Hacker cash or the next global currency. Pub. Int. L. Rep. **19**, 30 (2013)

[17] Duffield, E., Diaz, D.: Dash: A payments-focused cryptocurrency (2018), https://github.com/dashpay/dash/wiki/Whitepaper

[18] Faouzi, Ayoub and Ogilvie, Duncan and Lavrijsen, Matthijs and Sutherland, Graham and hfiref0x: Al-khaser, https://github.com/LordNoteworthy/al-khaser, accessed on 15-02-2019

[19] Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences **55**(1), 119 – 139 (1997). https://doi.org/https://doi.org/10.1006/jcss.1997.1504, http://www.sciencedirect.com/science/article/pii/S002200009791504X

[20] F.R.S., K.P.: Liii. on lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **2**(11), 559–572 (1901). https://doi.org/10.1080/14786440109462720, https://doi.org/10.1080/14786440109462720

[21] Gu, G., Perdisci, R., Zhang, J., Lee, W.: Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In: Proceedings of the 17th Conference on Security Symposium. pp. 139–154. SS'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1496711.1496721

[22] Gu, G., Porras, P., Yegneswaran, V., Fong, M.: Bothunter: Detecting malware infection through ids-driven dialog correlation. In: 16th USENIX Security Symposium (USENIX Security 07). USENIX Association, Boston, MA (2007), https://www.usenix.org/conference/16th-usenix-security-symposium/bothunter-detecting-malware-infection-through-ids-driven

[23] Gu, G., Zhang, J., Lee, W.: Botsniffer: Detecting botnet command and control channels in network traffic. In: NDSS (2008)

[24] Guarnieri, Claudio and Tanasi, Alessandro and Bremer, Jurriaan and Schloesser, Mark and Houtman, Koen and vanZutphen, Ricardo and deGraaff, Ben: Cuckoo sandbox, https://cuckoosandbox.org/, accessed on 25-03-2019

[25] Guarn, N.: Gaussianscatterpca.svg (2018), https://commons.wikimedia.org/w/index.php?title=File:GaussianScatterPCA.svg&oldid=324210291, [Online; accessed 4-May-2019]

[26] Hall, M.A.: Correlation-based feature selection for machine learning. Tech. rep., University of Waikato Hamilton, New Zealand (1999)

[27] Ho, T.K.: Random decision forests. In: Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1. pp. 278–. ICDAR '95, IEEE Computer Society, Washington, DC, USA (1995), http://dl.acm.org/citation.cfm?id=844379.844681

[28] Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification. Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep. (2019), https://github.com/zcash/zips/raw/master/protocol/protocol.pdf

[29] Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing In Science and Engineering **9**(3), 90–95 (2007). https://doi.org/10.1109/MCSE.2007.55

[30] Inc, O.: Openvpn, https://openvpn.net/, accessed on 31-05-2019

[31] Inc., T.T.P.: Tor Project: Abuse FAQ (2019), https://www.torproject.org/docs/faq-abuse.html.en, accessed on 12-06-2019

[32] John, G.H., Langley, P.: Estimating continuous distributions in bayesian classifiers. In: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence. pp. 338–345. UAI'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995), http://dl.acm.org/citation.cfm?id=2074158.2074196

[33] KENDALL, M.G.: A new measure of rank correlation. Biometrika **30**(1-2), 81–93 (06 1938). https://doi.org/10.1093/biomet/30.1-2.81, https://doi.org/10.1093/biomet/30.1-2.81

[34] Kharraz, A., Ma, Z., Murley, P., Lever, C., Mason, J., Miller, A., Borisov, N., Antonakakis, M., Bailey, M.: Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In: The World Wide Web Conference. pp. 840–852. WWW '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3308558.3313665, http://doi.acm.org/10.1145/3308558.3313665

[35] Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the gordian knot: A look under the hood of ransomware attacks. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–24. Springer International Publishing, Cham (2015)

[36] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B.E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J.B., Grout, J., Corlay, S.: Project jupyter, https://jupyter.org/, accessed on 24-03-2019

[37] Konoth, R.K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., Vigna, G.: Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1714–1730. CCS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243858, http://doi.acm.org/10.1145/3243734.3243858

[38] Kullback, S., Leibler, R.A.: On information and sufficiency. Ann. Math. Statist. **22**(1), 79–86 (03 1951). https://doi.org/10.1214/aoms/1177729694, https://doi.org/10.1214/aoms/1177729694

[39] Lab, K.: KSN Report: Ransomware and malicious cryptominers 2016-2018 (2018), https://securelist.com/ransomware-and-malicious-crypto-miners-in-2016-2018/86238/, accessed on 25-02-2019

[40] LLC, R.C.: pfsense, https://www.pfsense.org/, accessed on 31-05-2019

[41] Maron, M.E.: Automatic indexing: An experimental inquiry. J. ACM **8**(3), 404–417 (Jul 1961). https://doi.org/10.1145/321075.321084, http://doi.acm.org/10.1145/321075.321084

[42] McKinney, W., et al.: Data structures for statistical computing in python. In: Proceedings of the 9th Python in Science Conference. vol. 445, pp. 51–56. Austin, TX (2010), https://pandas.pydata.org/

[43] Meshkov, A.: Cryptocurrency mining affects over 500 million people. And they have no idea it is happening. (2017), https://adguard.com/en/blog/crypto-mining-fever/, accessed on 25-02-2019

[44] Morales, J.A., Al-Bataineh, A., Xu, S., Sandhu, R.: Analyzing and exploiting network behaviors of malware. In: Jajodia, S., Zhou, J. (eds.) Security and Privacy in Communication Networks. pp. 20–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[45] Musch, M., Wressnegger, C., Johns, M., Rieck, K.: Web-based cryptojacking in the wild. CoRR **abs/1808.09474** (2018), http://arxiv.org/abs/1808.09474

[46] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Bitcoin.org (2008)

[47] Narang, P., Hota, C., Venkatakrishnan, V.: Peershark: flow-clustering and conversation-generation for malicious peer-to-peer traffic identification. EURASIP Journal on Information Security **2014**(1), 15 (Oct 2014). https://doi.org/10.1186/s13635-014-0015-3, https://doi.org/10.1186/s13635-014-0015-3

[48] Oracle: Virtualbox, https://www.virtualbox.org/, accessed on 15-02-2019

[49] Palatinus, M., Capek, J., Moravec, P.: Stratum protocol (2015), https://slushpool.com/help/stratum-protocol/

[50] Pastrana, S., Suarez-Tangil, G.: A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. CoRR **abs/1901.00846** (2019), http://arxiv.org/abs/1901.00846

[51] Pavlov, I.: 7-zip, https://www.7-zip.org/, accessed on 18-03-2019

[52] Pearson, K., Galton, F.: Vii. note on regression and inheritance in the case of two parents. Proceedings of the Royal Society of London **58**(347-352), 240–242 (1895). https://doi.org/10.1098/rspl.1895.0041, https://royalsocietypublishing.org/doi/abs/10.1098/rspl.1895.0041

[53] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

[54] Piacentini, Mauricio and Ravanini, Raquel and Miltner, Jens and Morgan, Pete and Peinthor, Ren and Kleusberg, Martin and Clift, Justin and Haller, John T.: Db browser for sqlite, https://sqlitebrowser.org/, accessed on 18-03-2019

[55] Plohmann, D., Gerhards-Padilla, E.: Case study of the miner botnet. In: 2012 4th International Conference on Cyber Conflict (CYCON 2012). pp. 1–16 (June 2012)

[56] Postel, J.: User datagram protocol. STD 6, RFC Editor (August 1980), http://www.rfc-editor.org/rfc/rfc768.txt

[57] Postel, J.: Transmission control protocol. STD 7, RFC Editor (September 1981), http://www.rfc-editor.org/rfc/rfc793.txt

[58] Powers, D.: Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. Journal of Machine Learning Technologies **2**, 37–63 (01 2011)

[59] Rossow, C., Dietrich, C.J., Bos, H., Cavallaro, L., van Steen, M., Freiling, F.C., Pohlmann, N.: Sandnet: Network traffic analysis of malicious software. In: Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security. pp. 78–88. BADGERS '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1978672.1978682, http://doi.acm.org/10.1145/1978672.1978682

[60] Saad, M., Khormali, A., Mohaisen, A.: End-to-end analysis of in-browser cryptojacking. CoRR **abs/1809.02152** (2018), http://arxiv.org/abs/1809.02152

[61] Secure by Design Inc: Ninite, https://ninite.com/, accessed on 15-02-2019

[62] Segura, J.: The state of malicious cryptomining (2018), https://blog.malwarebytes.com/cybercrime/2018/02/state-malicious-cryptomining/, accessed on 25-02-2019

[63] Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal **27**(3), 379–423 (July 1948). https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[64] Sharafaldin., I., Lashkari., A.H., Ghorbani., A.A.: Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: Proceedings of the 4th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,. pp. 108–116. INSTICC, SciTePress (2018). https://doi.org/10.5220/0006639801080116, https://www.unb.ca/cic/datasets/ids-2017.html

[65] Sinegubko, D.: Hacked Websites Mine Cryptocurrencies (2017), https://blog.sucuri.net/2017/09/hacked-websites-mine-crypocurrencies.html, accessed on 25-02-2019

[66] Solanas, M., Hernandez-Castro, J., Dutta, D.: Detecting fraudulent activity in a cloud using privacy-friendly data aggregates. CoRR **abs/1411.6721** (2014), http://arxiv.org/abs/1411.6721

[67] Spearman, C.: The proof and measurement of association between two things. The American Journal of Psychology **15**(1), 72–101 (1904), http://www.jstor.org/stable/1412159

[68] SQLite Development Team: Sqlite, https://system.data.sqlite.org/, accessed on 18-03-2019

[69] Tahir, R., Huzaifa, M., Das, A., Ahmad, M., Gunter, C., Zaffar, F., Caesar, M., Borisov, N.: Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) Research in Attacks, Intrusions, and Defenses. pp. 287–310. Springer International Publishing, Cham (2017)

[70] Team, W.: Whonix, `https://www.whonix.org/`, accessed on 31-05-2019

[71] Threats, E.: Emerging threats ruleset, `https://rules.emergingthreats.net/`, accessed on 12-06-2019

[72] Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: A technical survey on decentralized digital currencies. IEEE Communications Surveys Tutorials **18**(3), 2084–2123 (thirdquarter 2016). https://doi.org/10.1109/COMST.2016.2535718

[73] Uslu, Hseyin and Dashjr, Luke: Poolservers, `https://en.bitcoin.it/wiki/Poolservers`, accessed on 22-02-2019

[74] Van Saberhagen, N.: Cryptonote v 2.0 (2013), `https://cryptonote.org/whitepaper.pdf`

[75] VBoxHardenedLoader Project: Vboxhardenedloader, `https://github.com/hfiref0x/VBoxHardenedLoader/`, accessed on 15-02-2019

[76] VirusTotal: Virustotal: Analyze suspicious files and urls to detect types of malware, automatically share them with the security community., `https://www.virustotal.com/`, accessed on 18-02-2019

[77] de Vries, A.: Bitcoin's growing energy problem. Joule **2**(5), 801 – 805 (2018). https://doi.org/https://doi.org/10.1016/j.joule.2018.04.016, `http://www.sciencedirect.com/science/article/pii/S2542435118301776`

[78] Wang, W., Ferrell, B., Xu, X., Hamlen, K.W., Hao, S.: Seismic: Secure in-lined script monitors for interrupting cryptojacks. In: Lopez, J., Zhou, J., Soriano, M. (eds.) Computer Security. pp. 122–142. Springer International Publishing, Cham (2018)

[79] Werner, T.: The Miner Botnet: Bitcoin Mining Goes Peer-To-Peer (2011), `https://securelist.com/the-miner-botnet-bitcoin-mining-goes-peer-to-peer-33/30863/`, accessed on 15-02-2019

[80] Wiki, L.: Comparison between litecoin and bitcoin — litecoin wiki, (2018), `https://litecoin.info/index.php?title=Comparison_between_Litecoin_and_Bitcoin&oldid=18`, [Online; accessed 2-May-2019]

[81] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2019), `https://ethereum.github.io/yellowpaper/paper.pdf`

[82] Zeltser, Lenny and Westcott, David: Remnux, `https://remnux.org/`, accessed on 15-02-2019

Appendix A

Excluded Malware Detections

This list contains all manually removed names from Section 3.1. The reports in the VirusTotal repository list the detection names (if detected) of up to 69 unique antivirus applications, most using different naming conventions. For that reason, any detections using the following strings were removed from the samples:

- BitCoinStealer
- CoinClipStealer
- CoinSteal
- CoinStealer
- Coinficon
- Coinge
- Coins
- Iminent
- Imminent
- Minecraftpe
- Minecru
- ProcessHijack.lrX@amiNEwoO

Note that a malware sample could still have been detected by another antivirus application with a name that includes “coin” or “mine” that is not on this blacklist. In that case, the samples would still be selected.

Appendix B

Implemented Statistical Features

For all features marked with *, we calculated the average, 25th percentile, maximum, median, minimum, standard deviation, sum, 75th percentile, and variance.

1. **Flow**: Was the flow flagged? (Used only as label)
2. **Flow**: Number of ACKs
3. **Flow**: ACKs by party A
4. **Flow**: ACKs by party B
5. **Flow**: Was the flow bidirectional?
6. **Flow**: *Bytes per packet
7. **Flow**: *Bytes per packet by party A
8. **Flow**: *Bytes per packet by party B
9. **Flow**: Ratio of total bytes A/B
10. **Flow**: *DNS response addresses
11. **Flow**: *DNS response additional records
12. **Flow**: *DNS response answer records
13. **Flow**: *DNS response authoritative records
14. **Flow**: *DNS response TTL
15. **Flow**: *DNS response errors
16. **Flow**: Duration
17. **Flow**: Number of packets
18. **Flow**: Packets by party A
19. **Flow**: Packets by party B
20. **Flow**: Ratio of packets A/B
21. **Flow**: Packets with PUSH flag
22. **Flow**: Packets by A with PUSH flag
23. **Flow**: Packets by B with PUSH flag
24. **Flow**: Packets with RESET flag
25. **Flow**: Packets by A with RESET flag
26. **Flow**: Packets by B with RESET flag
27. **Flow**: *TTL of packets
28. **Flow**: *TTL of packets by A
29. **Flow**: *TTL of packets by B
30. **Flow**: Packets with URGENT flag
31. **Flow**: Packets by A with URGENT flag
32. **Flow**: Packets by B with URGENT flag

33. **Conversation**: Was the conversation flagged? (Not used during training)
34. **Conversation**: Number of flows
35. **Conversation**: Number of destination ports
36. **Conversation**: Average number of destination ports per flow
37. **Conversation**: Number of TCP destination ports
38. **Conversation**: Number of UDP destination ports
39. **Conversation**: Ratio of TCP/UDP destination ports
40. **Conversation**: Conversation duration
41. **Conversation**: *Total bytes per flow
42. **Conversation**: *Total bytes by A per flow
43. **Conversation**: *Total bytes by B per flow
44. **Conversation**: *Flow duration
45. **Conversation**: *Total packets per flow
46. **Conversation**: *Total packets by A per flow
47. **Conversation**: *Total packets by B per flow
48. **Conversation**: *Total PUSH packets per flow
49. **Conversation**: *Total PUSH packets by A per flow
50. **Conversation**: *Total PUSH packets by B per flow
51. **Conversation**: *Total RESET packets per flow
52. **Conversation**: *Total RESET packets by A per flow
53. **Conversation**: *Total RESET packets by B per flow
54. **Conversation**: *Total URGENT packets per flow
55. **Conversation**: *Total URGENT packets by A per flow
56. **Conversation**: *Total URGENT packets by B per flow
57. **Conversation**: Number of source ports
58. **Conversation**: Average number of source ports per flow
59. **Conversation**: Number of TCP source ports
60. **Conversation**: Number of UDP source ports
61. **Conversation**: Ratio of TCP/UDP source ports
62. **Conversation**: *Total TCP bytes per flow
63. **Conversation**: *Total TCP bytes by A per flow
64. **Conversation**: *Total TCP bytes by B per flow
65. **Conversation**: *TCP flow duration
66. **Conversation**: *Total TCP packets per flow
67. **Conversation**: *Total TCP packets by A per flow
68. **Conversation**: *Total TCP packets by B per flow
69. **Conversation**: *Total UDP bytes per flow
70. **Conversation**: *Total UDP bytes by A per flow
71. **Conversation**: *Total UDP bytes by B per flow
72. **Conversation**: *UDP flow duration
73. **Conversation**: *Total UDP packets per flow
74. **Conversation**: *Total UDP packets by A per flow
75. **Conversation**: *Total UDP packets by B per flow
76. **Conversation**: Number of TCP flows
77. **Conversation**: Number of UDP flows
78. **Conversation**: Ratio of TCP/UDP flows