Abstract

Today we are seeing over 350,000 [1] malicious and so called potentially unwanted applications every day. The amount is so overwhelming that analysts are not able to process and analyse all these samples. In addition, we have several successfully attacks the later years by for instance the 1. March 2019, when the Norwegian aluminium and renewable energy company Norsk Hydro was attacked with the ransomware "LockerGoga" [4], the 1. September 2020 when the Norwegian Parliament Stortinget announced that they have had a data breach [3], and the 22. October 2020 when the European technology consulting company Sopera Steria had had a data breach, including their Norwegian department in Stavanger [2]. It is therefore necessary to either be able speed up the existing methods or develop new once, since it does not matter if you have been hit once, you can still be hit twice. Like the Norwegian Parliament Stortinget who was hit again in the beginning of March, due to the Microsoft Exchanged vulnerability CVE-2021-26857 [2], leading to a data breach. They were not the only once this time, there were several companies in Norway like the public transport company AtB, Andøy municipal in Norway, among others all over the world [3]. It is therefore necessary to find mitigation techniques that are able to help the analysts by processing the malicious files, in order for them to be able to focus on the malicious samples that poses the biggest treat, like the once described above. One solution to this problem is to let computers do the job, by utilizing machine learning. In this master thesis will therefore investigate the approach called STAtic Malware-as-Image Network Analysis (STAMINA). This approach converts malicious and benign files into grayscale images, and then uses a machine learning algorithm that is trained to identify objects like cats, dogs, houses, cars, etc. and learns this algorithm to detect the benign and malicious images.

Sammendrag

Vi ser daglig over 350,000 [1] skadevarer eller såkalte potensielt uønskede applikasjoner. Omfanget er så stort at det ikke er mulig for analytikere å prosessere og analysere alle disse filene. I tillegg så har vi flere vellykkede angrepet det siste åre med for eksempel den norske aluminiums og fornybare energi bedriften Norsk Hydro, som ble angrepet 1. mars 2019 med løspengeviruset "LockerGoga" [4], det norske parlamentet Stortinget som meddelte at de hadde blitt utsatt for ett dataangrep 1. september 2020, og det europeiske teknologiselskapet Sopra Steria som hadde ett datainnbrudd, inkludert deres norske avdeling i Stavanger den 22. oktober 2020. Det er derfor nødvendig å enten øke hastigheten på eksisterende metoder eller å utvikle nye, siden det hjelper ikke om du har vært utsatt for angrep tidligere, da det fremdeles kan skje igjen. Ett eksempel her er det norske parlamentet Stortinget, som igjen var angrepet i begynnelsen av mars, på grunn av en sårbarhet i Microsoft Exchanged CVE-2021-26857, som medførte ett datainnbrudd. De var ikke de eneste som ble angrepet denne gangen, det var flere selskaper i Norge som kollektivselskapet AtB og Andøy kommune, i tillegg til andre over hele verden [3]. Det er derfor nødvendig å finne teknikker som kan hjelpe analytikerne med å prosessere skadevare filer, for at de skal kunne fokusere på de filene som utgjør den største trusselen, som de beskrevet i eksemplene over. En løsning på dette problemet kan være å la datamaskiner ta seg av jobben, ved å ta i bruk maskin læring. I denne masteroppgaven vil vi derfor se nærmere på metoden som heter STAtic Malware-as-Image Network Analysis (STAMINA). Denne metoden gjør om skadevare og vanlige filer til gråskala bilder, for deretter å bruke en maskin lærings algoritme som er trent til å gjenkjenne objekter som katter, hunder, hus, biler osv. Deretter trenes denne algoritmen til å detektere vanlige filer og skadevare som bilder.

Acknowledgements

I would like to thank my supervisor Geir Olav Dyrkolbotn and my co-supervisor Trygve Brox at NorthernLifeLock, for the guidance during the writing of this master thesis. I would also like to thank my fellow student Alexander Daniel Forfot for our discussions in this field, a good friend and teammate! Thanks to Åsmund Kamphaug as well, for giving me a broader understanding of machine learning. Finally, I would also like to thank my friends and family for all the support I have received.

Table of Contents

List of Figures

List of Tables

1 Introduction

This chapter introduces the topics covered in this thesis. Then presents a description of the problem, the motivation for conducting this research project, the research question, and what the thesis seeks to contribute with.

1.1 Topics covered

The 1. March 2019, the Norwegian aluminium and renewable energy company Norsk Hydro was attacked with the ransomware "LockerGoga" [4], the 1. September 2020 the Norwegian Parliament Stortinget announced that they have had a data breach [3], and the 22. October 2020 the European technology consulting company Sopera Steria had had a data breach, including their Norwegian department in Stavanger [2]. These are just some of the headlines in 2019-2020 and shows that the numbers of attacks are increasing rapidly. On the contrary, this phenomenon is not new. More than 350,000 are daily observed of malicious and so called potentially unwanted applications [1]. Another example from last year that shows how fast the development of malicious application can go, is the malware "GoSearch22", who was discovered by VirusTotal [4] in December [5], targeting Apples new Silicon platform who was released November 10. This platform is also sharing the same architecture as the Apple iPhone and iPads, meaning that it would not only be able to infect Macs with their new Silicon, but could potentially reach a broader audience. The start of the news year 2021 is not different, by looking at the beginning of March, the Microsoft Exchanged vulnerability CVE-2021-26857 [2] leading to a data breach at the Norwegian Parliament Stortinget again, the public transport company AtB, Andøy municipal in Norway, among others all over the world [3]. This shows that if you have been attacked once, you still can be attack twice, and that the number of new samples is far above the capacity of malware analysts to handle by their own. It is therefore necessary to find mitigation techniques that are able to help the analysts by processing the malicious files, in order for them to be able to focus on the malicious samples that poses the biggest treat, like the once described above. One solution to this problem is to let computers do the job, by utilizing machine learning. In this master thesis will therefore investigate the approach called STAtic Malware-as-Image Network Analysis (STAMINA). This approach converts malicious and benign files into grayscale images, and then uses a machine learning algorithm that is trained to identify objects like cats, dogs, houses, cars, etc. and learns this algorithm to detect the benign and malicious images.

1.2 Keywords

Malware, malware classification, static analysis, image analysis, machine learning, deep transfer learning

1.3 Problem description

In reverse engineering and malware analyses, one of the problems that we are facing today, is the overwhelming amount of new malware samples discovered every day. This number is according to AV- TEST.org, over 350,000, consisting of new malwares and so called potentially unwanted applications [1]. When we take this into account, and the fact that malware often is obfuscated by packing or encryption, in either one or several rounds, there is just not enough resources, and time to analyse them all. Therefore, it is necessary to either increase the performance of existing methods or invent new ones. Because the analyst needs to be able to focus his or her energy on the new malware samples that really matters. The ones that pose the biggest threat, which could be samples that has never been seen before, or existing ones that has gain new and more dangerous features, like the ability to infect other computers by taking the advantages of a new an unknown vulnerability, also known as a zero-day.

1.4 Justification, motivation and benefits

This research is motivated by a paper called *"STAMINA: Scalable Deep Learning Approach for Malware Classification"* written by Intel Labs and Microsoft Threat Protection Team, Li Chen, Ravi Sahita and Jugal Parikh, Marc Marino [6]. The concept of this paper is to convert benign and malicious samples into images and then apply machine learning with deep transfer learning, to be able to identify the benign and malicious samples in their dataset.

However, the paper does not go into depth and clearly explain how they are measuring the performance of their machine learning algorithm. They claim that this approach also works with malware that are obfuscated by for instance packing or encryption, but they are not providing any results or explanation of how they have drawn this conclusion. For the machine learning algorithm, to be able to detect packed and encrypted samples, they might have used an approach where both some of the benign and malicious samples are packed with the same packers, and some of the benign and malicious samples are encrypted with the same encryption algorithm. In order to make the machine learning algorithm aware that benign files or software also might be packed and or encrypted. Otherwise, it is not possible to be sure that the actual result here is that the machine learning algorithm detected a packed or encrypted malware. It could in fact actually be that the machine learning algorithm only detected that specific packer type and or version, or encryption algorithm and or version. Hence it is not actually detecting if a sample is benign or malicious. It would also be beneficial if they had provided their code samples, dataset, and test environment, in order to be able to repeat and validate their result.

In this thesis we will therefore try to conduct the same experiment as they did, as closely as possible, but rather be focusing on if it is possible to detect packed and encrypted malware samples, by using the suggested approach above. We will also be providing the necessary theory to explain how the research is conducted and why, code, ML, and resized image samples. Along with the result of the experiment, in order for others to be able to redo, extend and or validate the founding's in this thesis.

1.5 Research questions

The primary goal of this thesis is to evaluate the validity and the reliability of the claims in [6] that STAMINA is well suited to detect packers. We want to investigate if we are able to recreate STAMINA and recreate similar results (reliability) and is STAMINA detecting packed malware or the packer, regardless of benign and malicious content (validity). In order to do this, we will look into the following questions:

1) How can STAMINA be implemented and used to detect packed malware?
2) What is the performance of the approach?
3) How does the performance compare with other methods, such as entropy analysis and packer signature detection?
4) What is being detected by STAMINA?

In order to limit the scoop of this master thesis, we will only be looking at the first implementation of STAMINA [6] by Chen [7] and not the approach involving the File size gate. In short, this approach is implemented in order to remove files that are skewed due to their file size distribution when they are resized, and therefore uses file size gate, to sorted out files by their file size, described in more details in [6]. The implementation by Chen [7] is here shown in his figure below.

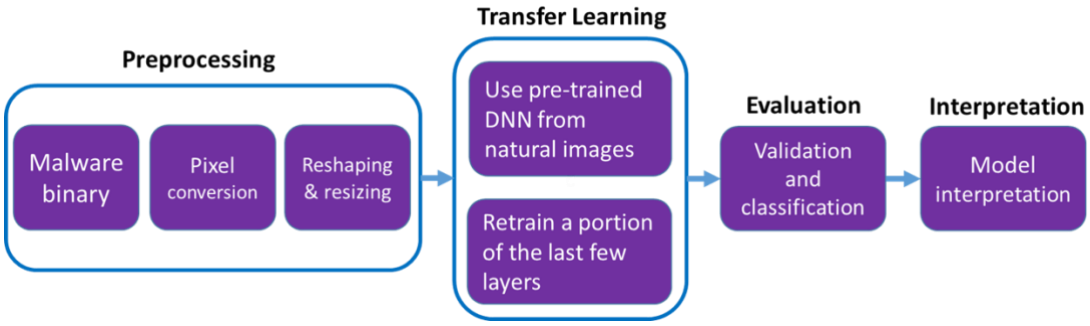


Figure 1.1: Shows STAMINA implementation by Chen [7]

1.6 Planned contributions

Other researchers [6] have looked into how malware can be converted to images by reading the malware sample byte by byte, converting every byte into a value between 0 and 255. This one-dimensional array of pixels would then be divided into a two-dimensional matrix, based on the file size and an empirical validated table. They convert both malicious and benign images, and then resize them by using the algorithm nearest neighbour or bilinear interpolation. The resizing is recommended to be either 224x224 or 299x299 *(height x width)* depending on the input shape of the pre trained deep neural network. Then they have applied transfer learning to a pre trained deep neural network of natural images, to take the advantages of transfer learning, saving time by only retraining the last few layers of the network, to be able to classify malicious and benign images. However, I have not been able to find any research that presents this whole process in code, for others to be able to revalidate their findings, investigate other approaches in this domain, or to build further on. Earlier research also points out that this method is able to detect packing and encryption, without showing to any results. The goal of this master thesis will therefore be to repeat, recreate and apply the approach suggested by Chen et al. in [6]. Our focus will be on the claim by [6] that their method can detect packed and encrypted malware. The research will also document this process, provide both code samples and resized benign and malicious images, for other researchers to be able to repeat and verify the results found in this thesis. This will ensure that it is easy for future scientists to check the validity and reliability of our contribution and to build upon it to potentially improve a much-needed detection capability. This should go without saying, but as the article by Chen et al. in [6] shows, this is not always the case.

1.7 Thesis outline

This section describes the outline of the thesis and gives a short description of each chapter.

- Chapter 2: This chapter presents the necessary background theory in order to understand why we later on are applying different kinds of methods and teqnices, as well as to understand the terminology used in this thesis along with the existing litterature and related work.
- Chapter 3: This chapter presents the methodology used to answer this master thesis and the research questions presented in the introduction. We will here divide the this chapter into preprocessing, create and train ML models and lastly ecaluate the trained ML models and results. In the preprocessing section we will be looking at creation of a database, header analysis, sample image conversion and reshaping, packer and encryption signature detection, and packing our own samples. In the Create and train ML models we will explain how we created and trained the machine learning models. Lastly in the evaluate the trained ML models and results, we will discuss how we are able to evaluate the performance of the machine learning models and other results.
- Chapter 4: Experiments. Presents the hardware and software setup used to conduct this experiment, along with how the experiment is conducted and the obtained result.
- Chapter 5: Discussion. Discusses the results obtained during the experiment.
- Chapter 6: Conclusion. Draws a conclusion based on the findings and future works.

2 Background theory and related work

This chapter presents the necessary background theory in order to understand why we later on are applying different kinds of methods and teqnices, as well as to understand the terminology used in this thesis along with the existing litterature and related work.

2.1 Malware

Malware is defined by Oxford Languages as: *"Software that is specifically designed to disrupt, damage, or gain unauthorized access to a computer system."* [8]

2.1.1 Types

There exist several types of malware [9] such as backdoors, viruses, worms, ransomware, and others. Our categorization here is based on [9], who categorize malware based on how they are behaving / what kinds of actions they are performing on the computer they are installed on. A malware sample may have more than one behavior (e.g., backdoor and botnet) and therefore belong to more than one category. Some common categories and their behaviour are:

- Adware: Shows unwanted advertisements usually based on browser history
- Backdoor: Allows the attacker access with little or no authentication to a computer
- Botnet: Devices managed by a command-and-control server (backdoors installed)
- Downloader: Allows the attacker to download and install additional malicious code
- Virus: Copying itself after a user interaction to other computers, infecting them
- Worm: Copying itself without user interactions to other computers, infecting them
- Spyware: Steals information from a given computer
    - Keylogger: Reads every stroke from the keyboard
    - Password hash grabbers: Steal's password hashes from disk, RAM, etc.
    - Sniffer: Monitors and collects internet traffic
- Trojan Horse: Legitimate software combined with malicious code (mislead users)
- Ransomware: Encrypts the users files and demand money to release them
- Rootkit: Designed to conceals itself, allows remote access, usually consist of other malware, like backdoor

The type of malware selected by an attacker, usually depends on what their goals and motivations are. Motivation can e.g., be economic / financial gain, opportunistic or targeted. The malware functionality needed will depend on the stage of the attack. Hutchins et al. describes in [10] an attack as a seven-stage process: Reconnaissance, Weaponization, Delivery, Exploitation, Installation, Command and Control (C2), and Actions on Objectives. This model is called the Cyber Kill Chain and Hutchins et al. descriptions are found in the figure below.

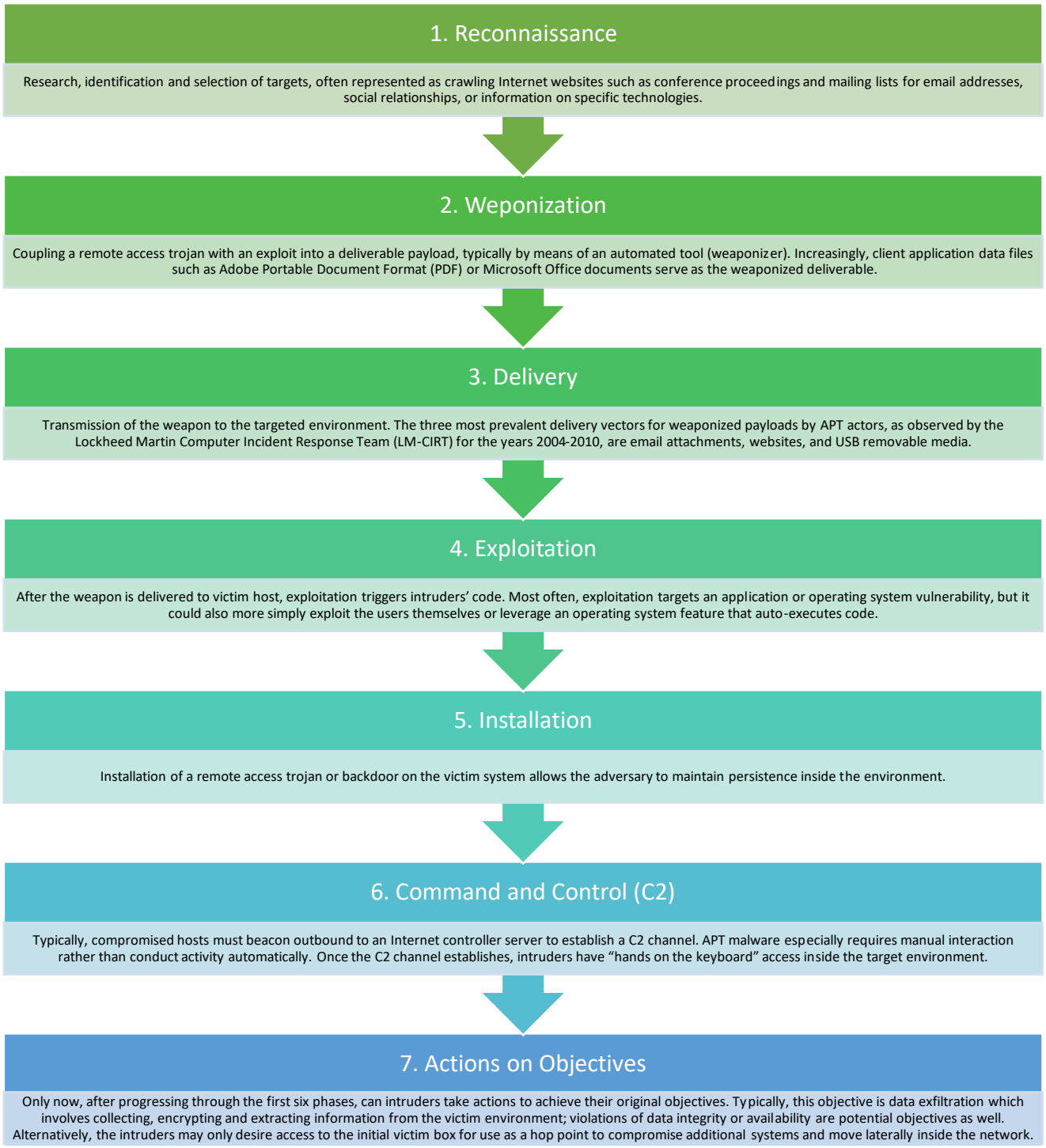


Figure 2.1: Visualization of the Cyber Kill Chain by Hutchins et al. [10] and their description of each of the steps

Malware can be written in any programming and scripting language. They can target anything from mobile phones, computers, cameras, washing machines, to components in an industrial control system, controlling things like the pumps in a wastewater treatment facility for instance. Malware therefore has to be written in a format that the targeted device will be able to understand, called a file format, in order for it to be able to

interpret and perform the desired action. A file format has a given structure, called a file structure. There are exists several types of file formats from proprietary formats where the file structure is not publicly known, to file formats like Executable and Linkable Format (ELF) [11] and Portable Executable (PE) where the file structure is known [12]. In this paper we will only be focusing on the Portable Executable (PE) file format who we will be describing next.

Portable Executable (PE file)

The Portable Executable File format (PE format) were introduced by Microsoft as a part of the original Win32 specifications, but comes from the operating system [13] called VAX/VMS introduced in the late 1970s [14] and is derived from the Common Object File Format (COFF) [12]. The reason for this was due to that the original Windows NT team came from the company Digital Equipment Corporation, who was the developer behind the VAX/VMS operating system. [14] It was therefore easy for these developer to use the existing code that they were used to program in [12]. The intent behind the term Portable Executable was to have a common file format for all versions of Windows and supported CPUs. [12] Today this format is still used for Windows executables in 32 and 64-bit Windows operating systems and can the most common formats EXE (also known as .exe) files, .NET executable and DLLs [15]. The only difference between an EXE file and a DLL, is a single bit, indicating if it should be treated as a DLL or an EXE file. (DLLs can also have the .OCX and .CPL extension instead of .DLL) [12]. The data structures that are on disk are the same data structure when a PE file is loaded into the memory [12]. To give some background information on the file structure itself. The structure can be seen in figure xx. The file starts with a small MS-DOS executable, printing that Windows is required, if the program were executed on another machine than Windows [12]. The MS-DOS header is the first bytes of the PE file and is also called IMAGE_DOS_HEADER. Here the e_magic and e_lfanew values can be found. The first value e_magic needs to be 0x5A4D in hexadecimal values or MZ in the ASCII character encoding [16] and indicates that the file is an EXE file. It is also referred to as IMAGE_DOS_SIGNATURE and EXE magic bytes. The second value e_lfanew is a pointer to where the file offset of the PE header is found in the file itself, and we will come back to these too values several times in this paper and give a little more detailed descrition of them in the section 3.2.1 Header analysis, extracting .exe magic bytes, PE signature and CPU architecture in chapter 3 Methodology. [12]. An whole overview of the file format can be found at the Windows Developer Documentation [17].

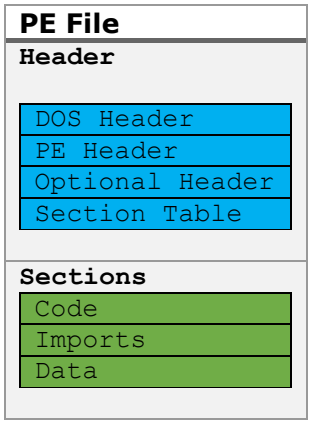


Table 2.1: Shows a simplified version of a PE file structure [15]

In this paper, we will not be moving further down in the PE File structure than the header and to the optional header. The method we are using here consists of looking at the values found in the header, called a PE header analysis. These values can then be extracted and contains information about the given file as described above, such as if the file is a EXE file, but also what processor architecture it is running on, along with packers cryptos and compiler signatures. These signatures can either be located in the header itself, or pointers from the header to other section in the file that holds this information. We will not here ourself detect packers cryptos and compiler signatures by writing code that looks for these signatures in the PE File, but the tool PEiD [18] will perform such operations for us. The PEiD [18] tool will be further described under the section 2.1.3.2 and Packers, cryptos and compiler signatures

2.1.2 Obfuscation

A technique that can be used by malware writers to avoid detection and or make analysis of a given malware sample more complicated, is called obfuscation. There are several ways to obfuscate files, but in this paper, we will only be looking at the two techniques called packing and encryption. The main purpose behind these two techniques are not in themselves malicious but they are commonly used by malware writers [9]. The two next sections will therefore be obfuscation techniques, where we are describing packing and encryption, and the next section obfuscation detection, where we will describe the two detection techniques, called file entropy and packers, cryptos and compiler signature detection.

2.1.2.1 Obfuscation techniques

Packing
According to Sikorski [9] packed programs are when a malicious program is compressed by the malware writer, and therefore cannot be analyzed. Running a packed program will result in a decompressing of the packed file, by a small wrapper program, and then an execution of the unpacked file [9]. A visualization of an original executable and a packed executable can be seen in fig 2.2, and as we can see, Strings are hidden when a program is packed, and there for a search for Strings in a packed program should result in few Strings compared to the not packed program. If we compare legitimate programs and malicious programs, a compress malicious program usually contains few strings, as oppose to legitimate programs, who usually contains many strings [9]. Packed files also often either include the functions LoadLibrary and GetProcAddress (if Strings are searched for in the executable), who are used to be able to load and gain access to additional functions. We will not go into details about this in this paper and more information can be found in [9].

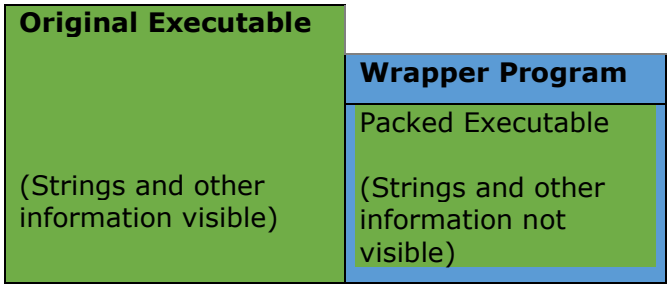


Figure 2.2: Shows the difference between an unpacked executable to the left and a packed executable to the right. The figure is an illustration from Sikorski [9]

Encryption
In a non-technical term, encryption can be described as a way of scrambling data, in order for only authorized parties to interpret the information [19]. The goal of the encryption process is to alter the readable data, so it appears random, even though encryption proceeds in a logical and predictable way. Described in a technical term, it is the process of converting plaintext into ciphertext (incomprehensible text) [19], and requires a cryptographic key, who can be described as a set of mathematical values, that both participants of the encrypted message has agreed on [19]. Like a physical key it unlocks (decrypt) and locks (encrypts) the data for someone with the right key. Data can be encrypted while it is in transit, described as being transmitted to somewhere else or at rest, described as when it is stored. In order for a third party to not guess the key and break the ciphertext by for instance guessing all possible values called brute force, a complex enough key should be used [19]. The two most common encryption types today are symmetric and asymmetric encryption (also known as public key encryption). Asymmetric encryption uses two keys, one for encryption and a different key for decryption. Opposed to symmetric encryption, who only uses one (same secret) key that all communicating participants is using for both decryption and encryption [19].

The reason for using encryption can for instance be due to privacy, security, data integrity authentication and regulations as described by [19]:

Privacy: Ensuring that either the rightful data owner or intended recipient is the only one that can read stored data or in transit. For instance, to prevent anyone from eavesdropping (listen) in like Internet service providers, ad networks and attacker.

Security: Ensures that data on disk is still secure if a hard drive or computer is lost or when communicating parties exchange sensitive data. Prevents data breaches (that someone steels the data) on disk or in transit.

Data integrity: Ensures that the data that the recipient receives har not been altered or tampered with on the way to him or her.

Authentication: For instance, TLS certificate for ensuring that a given user is connected to the real website and to establish a secure communication between his or her device to the website server.

Regulations: GDPR, HIPAA, PCI-DSS etc. are examples of compliance standards and regulatory, required by for instance government regulations and industry against companies to keep data encrypted, if they handle user data.

2.1.2.2 Obfuscation detection

There are several approaches being discussed in existing literature [20], [21], [22], [23] on how obfuscation of packed and encrypted malware can be detected. But here we are only looking at two of these methods. The first method is by calculating the file entropy and the second method is by utilizing a packer detection tool. In the two next sections, we will therefore describe file entropy and packers, cryptos and compiler signature detection, along with the approaches being discussed in the existing literature.

File entropy

In malware analysis, one measurement that is used to indicate whether a malware sample is packed or encrypted, is the entropy of the file. A file that is encrypted or packed, will have a higher entropy, than a malware sample that is not packed or encrypted. The entropy will be in the range [0-8] here, since we are basing our calculation on 8 bits, where 0 will be indicating that the file sample is not packed or encrypted, and the closer the value is to 8, the higher the probability would be that the file is encrypted or packed. The file entropy can be calculated by using Shannon´s entropy:

Formula for calculating Shannon´s entropy
The formula for Shannon´s entropy: $\boldsymbol{H(X) = -\sum_{i=1}^{n} P(x_i)\ log_b\ P(x_i) = -\sum_{i=1}^{n} P(x_i)\ log_2\ P(x_i)}$ b = the base, is here two (since a bit is either 1 or 0)

Table 2.2: Shows the formula for calculating Shannon´s entropy

File Entropy is based upon the hypothesis that the entropy of a packed file is different than the entropy of an unpacked file. This hypothesis may be stronger or weaker depending on what is included in calculating the entropy. There are several papers such as [20], [21] and [24], discussing how file entropy can be calculated. In paper [20] they used a blockwise entropy score of byte features of executables, to see if they were able to sort out the packed files. Their proposed method takes malware and benign files that are not packed, packs them with the same packers and then calculates their blockwise entropy score. Their conclusion was that their proposed method was capable of identifying packing.

The method used in [20] only focuses on one round of packing also called single layer, but malware can be packed in several rounds, for instance either re-packed with the same algorithm or multilayer packed with two algorithms. In paper [21], they used entropy analysis, to detect multi-layer packing. Their approach was to use symbolic aggregate approximation on the entropy of the executables. They also claim that this method is applicable for this kind of tasks. Paper [23] has a four step verification approach for packing and encryption detection, that we will described in a whole under the section packers, cryptos and compiler signature detection, since it fits better there. But we would here like to mention one of these verification processes that only are calculating the entropy of the entry point section, by using Shannon's entropy formula, and coming back to way later on.

In paper [20], [21] and [23] they all have in common that they are using Shannon's entropy as the ground formula for calculating the entropy, but what they are calculating the entropy of differs.

In this paper we will for simplicity stick to the original Shannon's entropy, where the

input file will be read in as binary, and we are creating an array in the range [0-255]. Every time we see a given binary value, the array at that position will be increased by one. After that we are using Shannon's formula to calculate the entropy of the array. But we will keep in mind the findings in [23], when it comes to benign unpacked files and false positives and benign packed files and false negatives described in the next section.

In order to determine where we should set the threshold for when a file is packed and encrypted, we are using the paper [24] and the their given table 2.3:

	Average entropy
Plain text	4.347
Native executables	5.099
Packed executables	6.801
Encrypted executables	7.175

Table 2.3: Shows the average entropy scores that paper [24] has concluded with

Packers, cryptos and compiler signature detection

In section 2.1.2.1 Obfuscation and under packing, we described how the content of an executable file is being hidden by the packer program, revealing a small wrapper program. This means that we instead can try to analyse the packer program, in order to determine what packer we suspect that the given file has been packed with. Like a PE file, where we can find information about what file type it is .EXE, DLL, .NET etc., what compiler version that has been used to create the file, when the file was created, the processor architecture, etc., a packer or encryption program also applies a similar approach, outside the packed or encrypted part. The reason for this is in order for the packer or encryption program to be able to detect what encryption or packing algorithm and version, that has been used, called a packer or encryption signature. This signature can be compared to a compiler signature in executable files, and it is therefore possible to extract, in order to trying to detect packers or cryptos. When this process of detecting such a signature is performed by a program, the program is called a packer identifier.

UPX 2.93 Hex signature from the packer identifier PEiD [18]

```
[UPX 2.93 (LZMA)]
signature = 60 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? 57 89 E5 8D 9C 24 ?? ??
?? ?? 31 C0 50 39 DC 75 FB 46 46 53 68 ?? ?? ?? ?? 57 83 C3 04 53 68 ??
?? ?? ?? 56 83 C3 04 53 50 C7 03 03 00 02 00 90 90 90 90 90
ep_only = true
```

Table 2.4: Shows an example of a packer signature from the packer UPX version 2.93, used by the packer identifier PEiD [18] [25]

In the example above, we have the signature name, the signature and a variable called ep_only, who here is true. Where the ep means entry point, or in other words, if the signature is found in the entry point of PE file, or if we have to look thru the entire file in order to find the signature pattern. According to the Windows Developer Documentation and under the PE Format, the address to the entry point, as Microsoft calls AddressOfEntryPoint, can be found by looking in the Optional Header Standard Fields (Image Only) [17].

In the literature [22] by Mi-Jung et al., they discuss an approach consists of a four-step verification, in order to determine if the given sample is malicious and packed. The first verification is to look at det entry point section of the file for a packer signature (like we are describing above), then they are looking for the WRITE attribute [23] in the entry point section, the third verification is an entropy calculation of the entry point section

only by using Shannon's entropy formula (as described in the File Entropy section), and the last one is a conclusion based on their findings. A file is labelled packed, if any of the mentioned verification tests above yields true [23]. The assumptions are that if there is no entry point section, the file is packed due to their findings showing that some packers scramble this section intentionally or hides it. The decision to only calculate an entropy score for the entry point section is according to [23] due to that the entropy calculation is to width (using the whole file for the entropy calculation) and that they on a benign file would get a false positive when the entropy value is low, and a false negative when the benign file is packed and have a hight entropy score. [23] Therefore this entropy calculation method and the WRITE attribute are according to [23] essential in reducing false positives, because a packed file needs permission to WRITE to be able to perform an unpacking, before the file is executed.

2.2 Malware analysis

The proccess of analysing a malware sample, to understand how the malicious software is able to cause a disruption, damage or gained unauthoized access, is called malware analysis. The goal of the analysis is to find out what functionallity the sample has, or in other words what malicious action the software is capable of performing, on or to a given computer or computers, network and networks, what the potetial impact is, if such an event should occure, who is the sender and programmer behind this malicous code (if possible). The analysing proccess can be divided into static and dynamic analysis [9].

Static analysis examins the malicious sample without running it and can further be divided into basic static analysis and advanced static analysis [9]. In basic static analysis we are examining a given executable file, without looking at the actual program instructions. Instead we are trying to determine the programs functionality by using antivirus tools like VirusTotal [4] where the file either can be uploaded, or we can calculate a file hash (as described below) for the given sample and search by the hash [9]. Another method in basic static analysis is to search for Strings in the malicious sample, who can give valubale information like messages that the program prints, URLs, IP addresses, Windows functions, etc. [9]. Information that we can extract from the PE file header or header information for other file types are also a part og basic static analysis, along with file entropy calculation of the whole file or parts of the file. Advanced static analysis, is on the other hand looking at the actual program instructions, by using a dissasembler program [26] in order to find out what capabilites that the program has [9].

Dynamic analysis on the other hand, consists of actually running the malicious samle, trying to detect what changes it makes to the computer it is running on, by utilizing logging and communication capture tools. Dynamic analysis is also divided into basic and advanced. The goal of basic dynamic analysis is according to [9] to produce an effective signature, remove the infection or both, by observing how the malicious samples is behaving when been executed. In advanced dynamic analysis we are taking it a step further by using a debugger [27] to examine the internal state of a malicious sample during executing, with the goal of extracting more detailed information [9]. We will in this paper only introduce the term dynamic analysis in order for the reader to understand that an analysis process can be divided into static and dynamic, and that a dynamic analysis is more time-consuming than a basic static analysis, hence an analyst performing dynamic analysis will be able to process a lower number of samples. More information about both dynamic and static analysis can be found in the book [9]. Next, we will describe the method call hashing can.

Hashing is a method that is commonly used to uniquely identify malicious samples. The unique hash that identifies that particular malicious sample is produced by running the malware through a hashing program [9]. There exists several hash function algorithm that can be used to create a hash, and in this paper we have implemented both The

Message-Digest Algorithm 5 (MD5) and the Secure Hash Algorithm 256 (SHA-256). The reason for this is due to that the datasets that we are using, who contains the benign and malicious files, have named them with either a MD5 or SHA-256 hash. Hashes can also be used to see if someone else already has analysed and identified that particular file sample [9], and we will here use the service VirusTotal [4], to retrive such information. In this thesis we will also using hashing and hashes to uniquely identify a particular sample during our experiment. For example when we are packing samples in chapter 3. Methodology, the sample that is beeing packed will keep the file hash it had before it was packed as its filename. This is done in order to be able to find the unpacked filename, so that we are able to compare them against each other later on. We wil also continue the naming convention with a hash as the filename for images that we are generating during the experiment, for other researchers to be able to verify our findings by beeing able to calculate the same hashes.

Terminal command md5 'file_name':	md5 'NTNU_Master_2018-12-17_EN.dotx'
MD5 file hash:	130bf9f28dd52637656e7e0558419ac2

Figure 2.3: Showing how the command md5 'NTNU_Master_2018-12-17_EN.dotx' in a Mac Terminal gives us a file hash for the master thesis template used in this paper.

2.3 Artificial Inteligence (AI), Machine Learning (ML) and Deep Learning (DL)

Any human-like intelligence exhibited by a robot, computer or any other machine, is in computer science defined as artificial intelligence (AI). In popular usage, it is defined as learning from experience and examples, making decisions, understanding and responding to language, recognizing objects, solving problems, and combining these and other capabilities. In other words, the ability for a machine or computer to mimic the capabilities of the human mind. AI is today a part of our everyday lives, due to its ability to process large amount of data in a more efficient and accurate way than a human can. They are built into speech recognizing virtual assistants like Amazons Alexa [28], who we can ask for directions, to play the next song on our playlist etc., completing words and sentences as we for instance is composing an e-mail by using Gmail [29], or in self driving cars to detect objects (cars, humans, houses etc.), road markings and signs, etc. to be able to autonomously drive a car like in Tesla´s Autopilot [30]. Artificial intelligence can be thought of as an umbrella, as seen in figure 2.4 [31]. Next, we will explain Machine Learning (ML) under section 2.3.1 and Deep Learning under section 2.3.2.

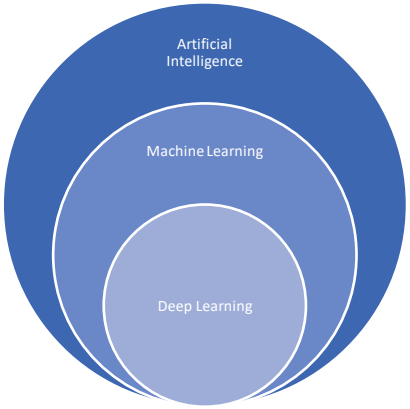


Figure 2.4: Stacked Venn diagram showing the relation between AI, ML and DL [31]

2.3.1 Machine Learning (ML)

Machine learning is divided into unsupervised and supervised learning. Unsupervised learning is when the algorithm is given a huge amount of data that is not labelled and given the task to label it. The machine learning algorithm then tries to understand the meaning behind the given data by classifying it based on clusters or patterns that it is detecting [32]. Clusters are data that the unsupervised learning algorithm has segmented into groups of examples, while patterns are when they are group by features. [32]. In that sense the labels are defined by the unlabelled data. Supervised learning on the other hand, is when the algorithm is given data that is labelled, hence the algorithm gets a certain understanding of how the data is classified [32]. The machine learning tries to find patterns in the data, that can be applied to an analytics process. The meaning behind the data are their labels [32].

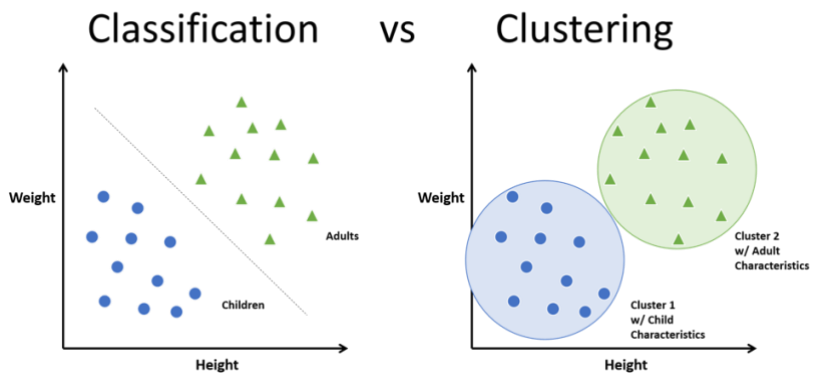


Figure 2.5: Stacked Shows how data is clustered into two clusters, one for children and one for adults [33].

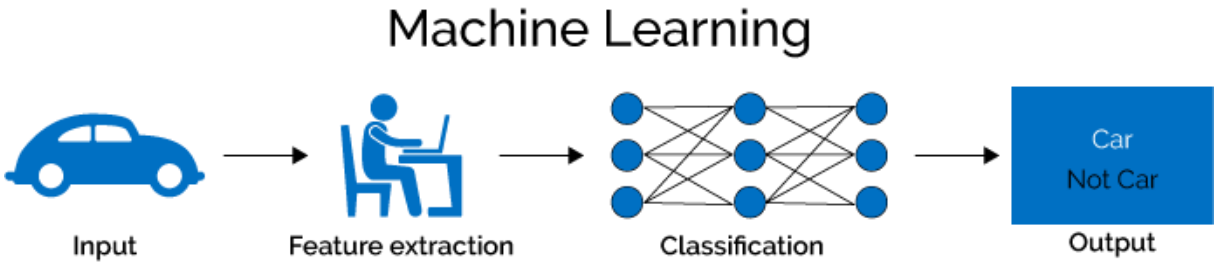


Figure 2.6: Shows how Machine Learning and supervised learning works [34]

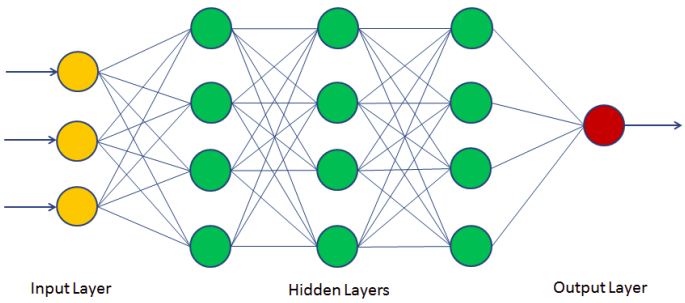


Figure 2.7: Shows a basic neural network [35]

Classification and validation

In order to be able to say something about how well the approach STAtic Malware-as-Image Network Analysis is performing by its own, and compared to entropy analysis and packer signature detection, we need to introduce some measurement techniques. These techniques will further help us to interpret, describe and illustrate the performance. We will further divide this intro classification and validation.

Classification
Samples are in this paper classified into benign and malicious, packed benign, not packed benign, malicious packed and malicious not packed. When we later on are using a machine learning algorithm to classify these results, we need to introduce the following terms:

True Positive (TP): A sample is 1, and predicted to be 1

False Positive (FP): A sample is 0, but predicted to be 1 *(also called a Type 1 Error)*

False Negative (FN): A sample is 1, but predicted to be 0 *(also called a Type 2 Error)*

True Negative (TN): A sample is 0, and predicted as 0

Accuracy: How often the predicted value are equals the correct value *(correct count / total)* in percentage [36]

Binary accuracy: Same as Accuracy, but is used for binary labels (in this paper benign or malicious), resulting in that the predicted value is the probability of the prediction being equal to 1. 1 is assigned if the probability is above the threshold, else it is 0. Are the predicted value equal the correct value, it is considered accurate [36]. For instance in the neural network library Keras, the threshold is default 0.5 [36].

Binary cross entropy: Can be either 0 or 1, compares each of the predicted probabilities to the actual class [37]

Precision: Looking at the positive identifications, and what proportion that was actually labelled correct. Here a model with no false positives will result in a precision 1.0 [38]

Recall: Looking at the actual positives that was identified correctly and in what proportion. Here a model with no false positives will result in a recall 1.0 [38]

Sigmoid: Activation function (also known as logistic function) used to calculate the output of the neural network, returns a value between 0 and 1. Meaning that we based on the output would sort the output into class 1 or class 0. In our case, benign will be 0 and malicious will be 1, hence if the returning value from the Sigmoid function is below 0.5, we sort into class 0, benign. Otherwise, we sort it into class 1, malicious [39]. According to Karakaya [40] the Sigmoid is mostly used in binary classification, due to that it is equivalent to a Softmax function [40] with two elements, there the second element is assumed to be zero.

Adam: Is the optimiser used for the convolutional neural network [41].

Confusion matrix
The above measurements for TP, FP, FN, TN can be combined into a confusion matrix, in order to better visualize the result:

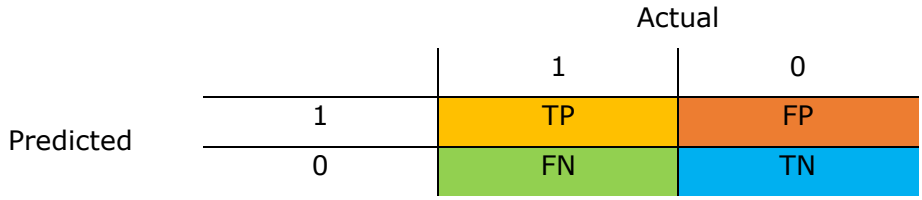


Figure 2.8: Confusion matrix for TP, FP, FN and TN

In our case the 1s and 0s in the matrix above would then be replaced by malicious and benign, packed and not packed, resulting in:

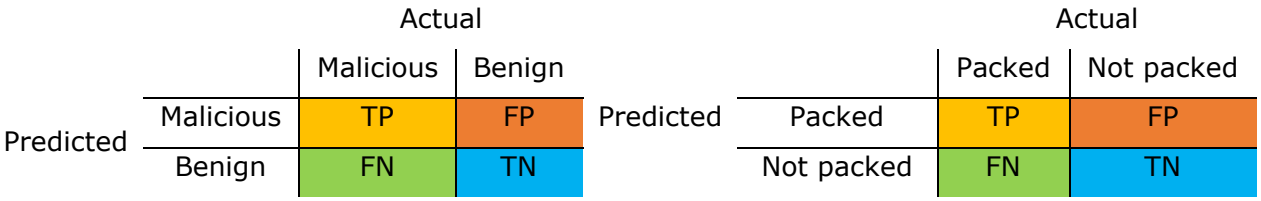


Figure 2.9: Confusion matrix benign and malicious, packed and not packed for TP, FP, FN and TN

Validation
In machine learning, in order to be able to say something about how well the algorithm performing / working, we need to introduce the following terms:

Overfitting
When a machine learning model is modelling the training data to well. [42] The model here learns the random fluctuations or noise in the training data as concepts by the model. What the model then has picked up as concepts does then not apply to new data, hence therefore impacting the model's ability to generalize negatively [42].

Underfitting
In contrast to overfitting, underfitting is when the model neither generalize to new data, nor model the training data [42]. The model will have poor performance on the training data and not result in a suitable model. According to [42] it is easy to detect with a good performance metric.

K-fold cross validation
To estimate how well machine learning models are performing in order to be able to compare and select a model, a common statistical method used in the machine learning field, is cross-validation. [43] The reason for this is that it is relatively easy to implement, understand and is a method that have a lower bias when it comes to skill estimation in general according to Brownlee [43] due to its more strict estimation, compared to for instance a train / test split like 70 % for training and 30 % for validation [43]. One of these cross-validation methods is called k-fold cross-validation and we will now further describe how this method is estimating the skill of machine learning models, by first presenting the model, then how we select the k value for a given dataset and lastly describe a variant of k-fold cross validation called stratified k-fold cross validation.

The k-fold cross-validation method is used to estimate the skill of a machine learning model on unseen data. Or in other words a prediction of how well the machine learning

model is expected to be performing in general, when the model is presented data that it has not seen before [43]. A k-fold divides the entire dataset into smaller groups where x groups are used to training and x groups are used for validation as seen in fig. 2.10. The number of groups that a k-Fold is divided into is based on a parameter called k. This k value can also be used instead of the k when the method is describe like a 10-fold cross-validation for a k with the value 10, or a 5-fold cross validation for a k with the value of 5, etc [43]. The k-fold method can be divide into the sub steps: Randomly shuffle the dataset, splitting the dataset into k groups, (each unique group: holding a group as validation data, use the rest as training data, fit the model on training data, evaluate on validation set, keep its evaluation scores, repeat the process for the next ML model), and lastly estimate the skill of the model by summarizing the models evaluation scores [43]. A visualization of the process can be seen in fig. 2.10. below.

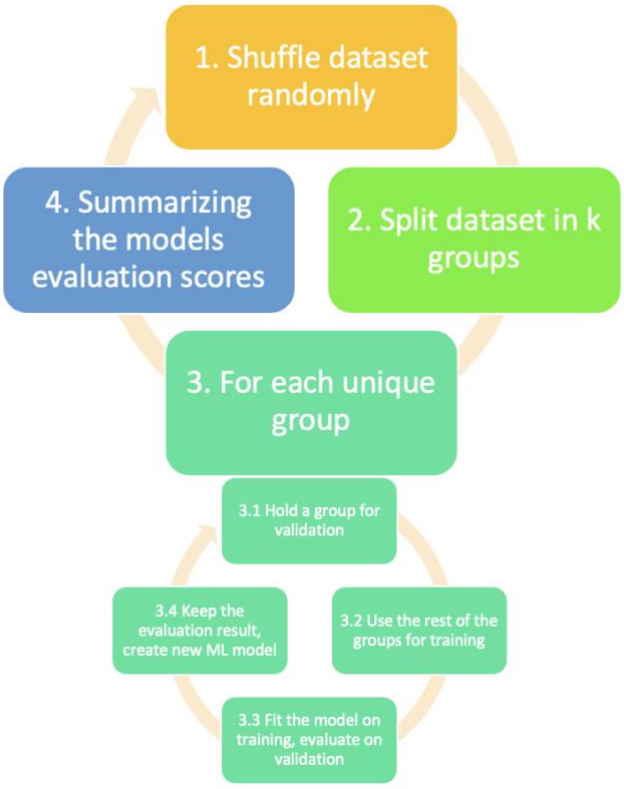


Figure 2.10: Show the processes within a k-fold and (also applicable for a stratified k-fold) described by Brownlee [43]

The selection of a k value is important, in order to not give a falsely impression of how the model is performing. According to [43], a such score can for instance be a high bias (the performance of the model is overestimated) or a high variance (the data used to fit the model varies very much). In order to mitigate this, Brownlee [43] suggests three tactics: 1. The chosen k value is large enough to be statistically representative of the broader dataset. 2. Setting k = 10, resulting in a model performance estimate with modest variance and low bias. 3. Setting the k = n, where n is the number off samples in the dataset, letting every test sample be in the validation dataset, called leave-one-out-cross-validation [43]. He also describes the common range for k as [5-10], where a larger k will result in differences between the validation and training sets gets smaller, resulting in a lower bias for each increase [43].

Dataset (100%)	
Train (90%)	Val (10%)

Val	Train	Train	Train	Train	Train	Train	Train	Train	Train
Train	Val	Train	Train	Train	Train	Train	Train	Train	Train
Train	Train	Val	Train	Train	Train	Train	Train	Train	Train
Train	Train	Train	Val	Train	Train	Train	Train	Train	Train
Train	Train	Train	Train	Val	Train	Train	Train	Train	Train
Train	Train	Train	Train	Train	Val	Train	Train	Train	Train
Train	Train	Train	Train	Train	Train	Val	Train	Train	Train
Train	Train	Train	Train	Train	Train	Train	Val	Train	Train
Train	Train	Train	Train	Train	Train	Train	Train	Val	Train
Train	Train	Train	Train	Train	Train	Train	Train	Train	Val

Table 2.5: Shows how a k-fold cross validation split for k 10 is done, each row represents an iteration k, from [1-k], and each cell either belongs to the training or validation set.

Stratified k-fold follows the same splitting technique as an ordinary k-fold, seen in table 2.5, but it also seeks to preserve the percentage of samples for each classes *(in our case benign and malicious)* in the given dataset [44]. Table 2.6 shows a simplified visualization of a stratified k-fold, with a dataset of 20 samples, where 8 are benign and 12 malicious. Since there are more malicious than benign samples in our example here, the stratified k-fold will only manage to keep the percentage of samples for benign and malicious in 8 out of 10 folds, hence 2 out of 10 folds are not kept. The visualization does not show how the distribution within each fold are shuffled accordingly to an ordinary k-fold.

8 benign 12 malicious	Dataset (100%)									
	Benign	Benign	Benign	Benign	Benign	Benign	Benign	Benign	Malicious	Malicious
	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious

8 of 10 folds	Train (90%)									Val (10%)
	Benign	Benign	Benign	Benign	Benign	Benign	Benign	Malicious	Malicious	Benign
	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious

2 of 10 folds	Train (90%)									Val (10%)
	Benign	Benign	Benign	Benign	Benign	Benign	Benign	Benign	Malicious	Malicious
	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious	Malicious

Table 2.6: Shows how a stratified k-fold with k = 10, seeks to preserving the percentage of samples for each class (benign and malicious). Here 8 of 10 are preserved, and 2 of 10 where not possible. For simplicity, the samples in this visualization are not shuffled.

2.3.2 Deep Learning (DL)

Deep learning is a subset of machine learning, who without human intervention, with increasingly great accuracy, teaches itself to perform a specific task. These models are based on neural networks with several hidden layers, who each forwards their result / calculations to the next layer, called forward propagation. It is also possible to identify errors in these calculations, assigns them weighs, and then sends them back to previous layers, called backpropagation, who is used for either training or refining the model. These models are therefore called deep neural networks (DNN)and can work with both labelled and unlabelled data, unsupervised or supervised learning [31].

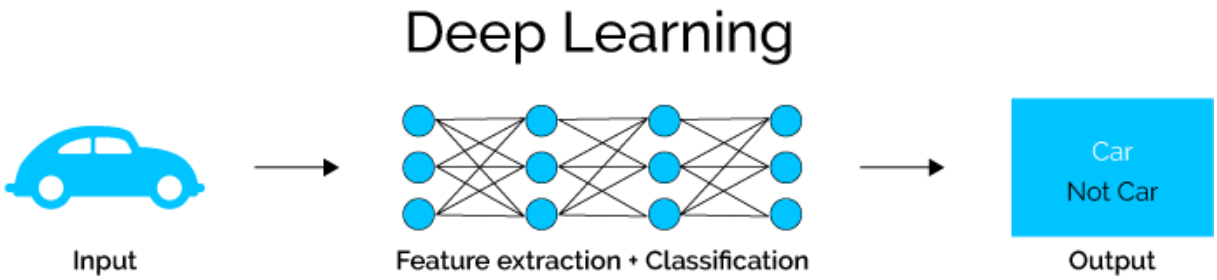


Fig 2.7: Shows how Deep Learning works [34]

Convolutional Neural Network (CNN)

Conventional Neural Networks are inspired by how the animal visual cortex is organized and the individual neurons are organized so that they respond to overlapping regions that are tilting the visual field [45]. It is a type of feed-forward neural network that are composed of three types of layers, fully connected, convolutional and polling [45].

Deep Transfer Learning (DTL)

Deep transfer learning is an approach that has been applied in several fields these days, due to the fact that training a new deep neural network from scratch, takes an awful lot of time and resources. The concept here is to borrow knowledge used in another domain and apply it to the new domain, resulting in reduced time and effort spent on training the neural network, and the benefit of yielding a high classification performance. This approach can also be beneficial, when the new domain has a relatively small dataset due to for instance lack of data in that particular field (called a limited dataset, due to its size), compared to the borrowed domain. The analogy often used to describe Transfer Learning, is that one person can transfer his or her knowledge to another person.

Data augmentation

One approach that can be used when we have a limited dataset (few sample images of the specific task that we are trying to solve), and therefore also might be very applicable in transfer learning, is called data augmentation. The goal of this concept is to create a more diverse dataset, to reduce overfitting. The common approaches here is to apply horizontal flipping, cropping or padding to the sample images in the dataset. Resulting in more samples, without actually collecting new data. But this is not always applicable, due the approaches described above. For instance, it can be applicable to pictures of houses, because if we flip a picture of a house horizontally, it is still an image of a house, if we crop the images, it would still be a part of the house etc. A photographer might have

intentionally or unintentionally created such images. But if either of these approaches was used on benign and malicious samples converted to images, this would not be applicable at all, since these samples would not be represented in this way in the real world.

Fig 2.8: Shows two images of the same image, where data augmentation and horizontal flipping is applied. Here we can see that the method is applicable, because the image still shows buildings, and can be represented this way by a photographer, resulting in two samples, instead of one.

2.4 STAtic Malware-as-Image Network Analysis (STAMINA)

STAtic Malware-as-Image Network Analysis (STAMINA) is the name Intel Labs and Microsoft Threat Protection Intelligence Team, gave their approach of static malware classification. They use deep transfer learning from computer vision, to find a deep learning technique with high accuracy and low false positives, compared to time-consuming manual feature engineering [6]. The concept here is to convert malware samples into images and take the advantages of transfer learning and a pretrained deep neural network used to classifying natural images or objects. This is done by retraining the last layers on "images" converted from malicious and benign files. The entire process can be divided into pre-processing, transfer learning and evaluation [6], as seen in figure 2.9 and explained next.

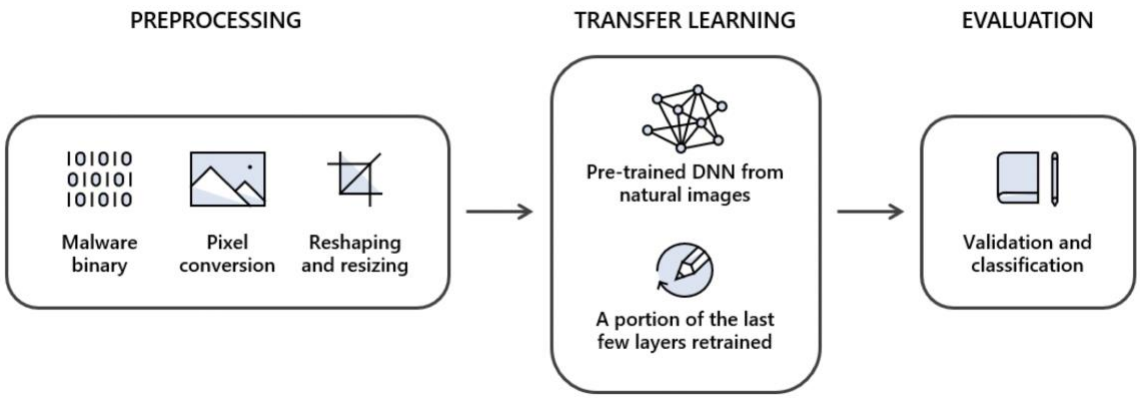


Fig 2.9: Shows the first three steps of the STAMINA method, Figure 1. From paper [6]

2.4.1 Preprocessing

The purpose of the pre-processing step is to convert malicious and benign files into images of the specific format needed to train the Deep Neural Network in the step 2.4.2. The pre-processing step is divided into pixel conversion and reshaping and resizing.

2.4.1.1 Pixel conversion

One way of converting a binary sample, into an image is by reading the malware sample byte by byte. The reason for doing so, is due to the fact that a byte is represented as 8 bits, resulting in 2^8 possible combinations or 256 choices, which is the same as a grayscale image, which only has one channel see figure 2.12, consisting of grey scale pixel values in the range [0-255]. Each and every byte will then directly become a shade of grey. The outcome of the pixel conversion process, will be a one-dimensional array of decimal values, as seen in figure 2.10, which we will reshape and resize in the next pre-processing step [6].

	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits
Binary	01001101	01011010	10010000	00000000	00000011	00000000	00000000
Decimal	77	90	144	0	3	0	0

Fig 2.10: Shows how the first 56 bytes of a given binary sample is converted to a one-dimensional array (here marked in grey), reading 8 bits, converting them to decimals

2.4.1.2 Reshaping and resizing

From the pixel conversion step above, the outcome is a one-dimensional array of decimal values, that each and every one is representing a grey scale pixel value. In order to create an image, we would need to reshape this long array of decimal values into a two-dimensional array, since an image has a width and a height e.g., two dimensions. We therefore start by determine the image width and height by the empirically validated table shown below, suggested by paper [6]. The reason for using this table is to be able to keep the relation between the image width and height linearly scaled. The pixel file size is the actual file size and is here calculated by getting the length of the one-dimensional array. Once the width is determined by the table, the height is calculated by dividing the one-dimensional array by the width, and if the result is a decimal number, it is rounded up, and the extra pixels are padded as zeroes. The result from this step, would now be a sample converted into an image [6].

(Pixel) File Size	Image Width
Between 0 to 10	32
Between 10 and 30	64
Between 30 and 60	128
Between 60 and 100	256
Between 100 and 200	384
Between 200 and 1000	512
Between 1000 and 1500	1024
Greater than 1500	2048

Table 2.11: Shows the relationship between pixel file size and image width, where the image with is the number of bytes (now pixels) at each row, and the [6]

		Row (width)		
		0	**1**	**2**
Column (height)	**0**	255	98	32
	1	0	70	56
	2	4	43	7

Fig 2.12: Grayscale with one channel [46]

The last pre-processing step is to resize the image according to the Deep Neural Networks input requirement, in order for the neural network to be able to interpret the images it is given. The input requirement is referred to as the input shape and written in the format (224, 224, 3), where the two first values are representing the image height and width and the third value is referred to as the number of colour channels that the deep neural network model is supporting. The third value in the input shape is set to 3 when the model is requiring three colour channels, meaning that it is requiring a colour image, who has the three channels: red, green, and blue (RGB). If the model is requiring a grayscale images, the third value in the input shape will be set to 1, since a greyscale image only has one channel as mentioned earlier. It is here important to understand that the input shape is the deep neural network models requirement, we cannot therefore send a grayscale image directly to a model that is requiring a colour image as an input, however a grayscale image can be converted into a colour image by repeating the one-color channel three times [7], see figure 2.13. In that way, we are not limited to selecting a model that is only supporting grayscale images. The reason for mentioning this here is, due to the fact that paper [6] is using the model Inception-v1, who has the input shape (224, 224, 3) [47], and if we are following the references in [6] we see that Chen [7] is converting the grayscale image into a colour image in his paper from 2018, before passing it to the deep neural network, by following the one-color channel three times replication described above.

	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits
Binary	01001101	01011010	10010000	00000000	00000011	00000000	00000000
Red	77	90	144	0	3	0	0
Green	77	90	144	0	3	0	0
Blue	77	90	144	0	3	0	0

Fig 2.13: Shows how the first 56 bytes of a given binary sample can be converted to three one-dimensional arrays, one for red, one for green and one for blue (RGB)

		Row (width)		
		0	**1**	**2**
Column (height)	**0**	255	98	32
	1	0	70	56
	2	4	43	7

		Row (width)		
		0	**1**	**2**
Column (height)	**0**	255	98	32
	1	0	70	56
	2	4	43	7

		Row (width)		
		0	**1**	**2**
Column (height)	**0**	255	98	32
	1	0	70	56
	2	4	43	7

Fig 2.14: Shows how a red, green and blue (RGB) looks like with three channels (0, 1, 2) [46]

Paper [6] recommends that the image is resized to either 224 by 224 or 299 by 299 (height x width). This does not only limit the scoop of models to choose from, but also means that we are losing some of the information in the original image, due to the resizing. They are in [6] further suggesting that the resizing algorithm used should either be nearest neighbour [48] or bilinear interpolation [49]. We will not here be describing how the nearest neighbour [48] and bilinear interpolation [49] algorithms are working, but rather show an image to give an impression of how the quality of an image is after compression by either of them, as seen in fig 2.15. According to [6] either of these resizing algorithms will not impact classification results or pattern matching techniques to for instance detect malware from the same family, due to the extraction of deep-represented features by the deep neural network.

Fig 2.15: Shows how a car image (to the left) is impacted by being resized by nearest neighbour (in the middle) and bilinear interpolation (to the right). Here the compressed images are magnified 930%. Tabora [50]

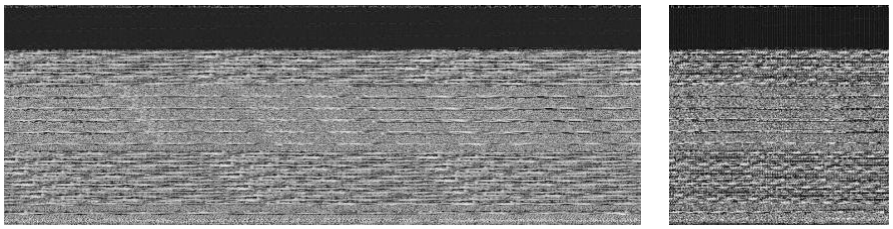


Fig 2.16: Shows the sample with sample_id = 8074 in dataset 1, original file to the left and scaled file to the right

2.4.2 Transfer learning

Transfer learning is in our case when a pre-trained convolutional neural network is used as the base model, in order to take the advantages of that models previous learning and apply them to the new domain we are trying to solve. In our case benign and malicious classification. Transfer learning can be divided into pre-trained deep neural network and retraining the deep neural network, explained next.

2.4.2.1 Pre-trained Deep Neural Network

The purpose of pre-training is to avoid needing to train the whole deep neural network from scratch, by taking the advantages of that previous model's learnings. There are several ways to apply transfer learning such as feature extraction and fine-tunning. We will explain the two methods relevant for this paper, feature extraction and fine-tuning, in next.

2.4.2.1.1 Feature extraction

The approach feature extractor consists of keeping the whole pre-trained network, except for the last fully connected layer, who is replaced with a new layer. The new layer is usually a dense layer, meaning that this layer will receive the output from all of the previous layers in the pre-trained network. The dense layer can for instance be given the sigmoid function [39], if the result is one of two desired outcome, also called binary classification, or the softmax function [51], if there are more than two desired outcomes, also called multi-classification [6].

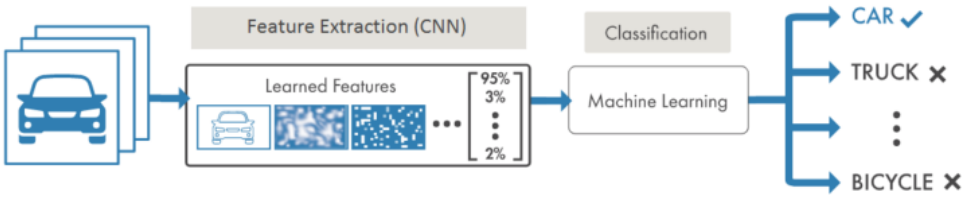


Fig 2.17: Shows a feature extraction model, with 4 output layers [52]

2.4.2.1.2 Fine-tunning

Fine-tuning is when you keep the pre-trained convolutional neural network and add a new classification on top of the already existing network, as shown in figure 2.18, where they have added a new classification layer to detect if the image is of a car or a truck. The newly added layers and some of the layer behind the newly added layers are then fine-tuned, meaning that we freeze all the other layers except for them, in order to not train the whole model, but only some parts. To not overfit the model, this is done with a low learning rate. The idea behind not training all the layers of the network, is that if the model has been trained on enough images in our case, the last layers should be generalized enough to be able to be apply to the new problem we are trying to solve.

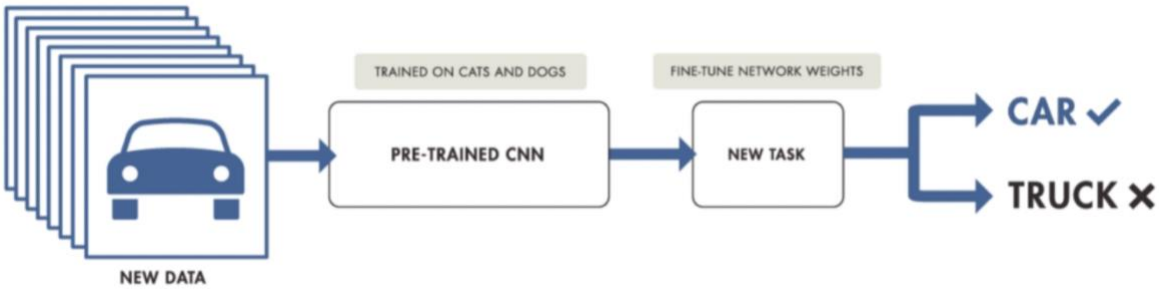
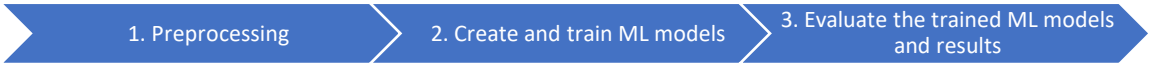


Fig 2.18: Shows a pre-trained CNN with fine-tuning with two output nodes [53]

3 Methodology

This chapter presents the methodology used to answer this master thesis and the research questions presented in the introduction. We will here divide the this chapter into preprocessing, create and train ML models and lastly ecaluate the trained ML models and results. In the preprocessing section we will be looking at creation of a database, header analysis, sample image conversion and reshaping, packer and encryption signature detection, and packing our own samples. In the Create and train ML models we will explain how we created and trained the machine learning models. Lastly in the evaluate the trained ML models and results, we will discuss how we are able to evaluate the performance of the machine learning models and other results.



3.1 Dataset

In this thesis we are using a small dataset referred to a as Dataset 1. This dataset consisting of 10204 malicious files (originaly 1082 samples, and the malware family's agent, hupigon, obfuscator, onlinegames, renos, small, vb, vbinject, vundo, zlob. Where each families is 1000 samples, except for vuno who is 823) and 4388 benign files. There are 2176 benign packed samples, where we packed 1860 with UPX 3.96, and 4886 samples where 430 where packed by us with UPX 3.96. The benign files were gathered by Sergii Banin [54] in September 2019, and is a collection of software that is free and portable downloaded from Portabelapps.com. The method used to download the software is a grab-it-all approach, meaning that he was downloading all the samples he managed to get his hands on from their website. The software downloaded is Windows applications that is possible to run from a USB-stick, without installation. In Banins [54] dataset we removed duplicates and files that were not .exe by our suggested approach in the in this section.

3.2 Preprocessing the dataset

The pre-processing of the dataset can be divided into the five main tasks: creation of a database, header analysis, sample image conversion and reshaping, packer and encryption signature detection, and packing our own samples, as seen in figure 3.1 where each main topics also have their own subtasks. The main goal here is to reduce the number of times that we would need to pre-process the entire datasets. Saving us time later on when we are creating the machine learning models and performing our experiment, starts a new one or tries to re produce an existing experiment.

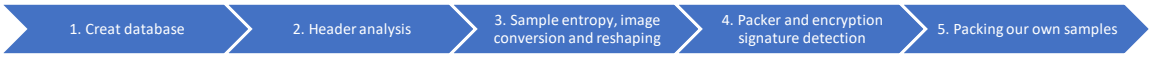


Fig 3.1: Showing the main tasks and subtask ADD SUBTASKS TO THE FIGURE

Creating a database
The first thing that we are creating in the pre-processing stage is a database, in order to be able to save the outcome of the pre-processing stage such as the results of the header analysis, the name of the converted images, the detected packer and encryption signatures, along with the name of the files that we packed ourselves etc. Another benefit by this approach is that we later on can extract information much easier and share the gathered information with others.
An ER-diagram (a drawing of the database structure), showing what tables the database consists of and what datatypes the different fields are like: plaintext, numbers, dates, etc. can be found in appendix 24.

3.2.1 Header analysis, extracting .exe magic bytes, PE signature and CPU architecture

The header analysis is performed to limit the scoop of this thesis to only PE files for Intel x86 and AMD x64 CPU architectures, and to verify what type of file that we are processing. The header analysis is further divided into a three-step verification process, in order to gather as much relevant information as possible about the file, before a decision is made. The first step is to read the .exe magic bytes, then we are looking for a PE signature, and lastly, we are extract the machine type field. As seen in figure 3.1, if the preceding step fails, we will not be able to move further one and extract any of the other values in the later steps. This is due to the assumption that if the .exe magic bytes are not found, the file is not a .exe file and we will therefore not be able to retrieve any of the other values. A drawback by this assumption is that if someone else changes these to any other values than .exe magic bytes, our detection will fail. Nevertheless, in this experiment we have not been able to find any sample that has failed our detection by the earlier described scenario, and therefore here for this experiment concludes that the method holds.

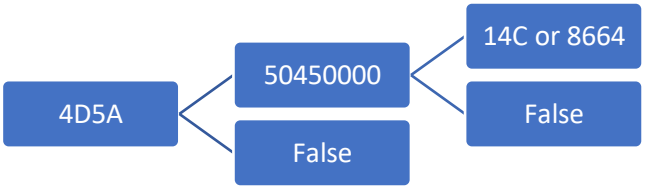


Figure 3.1: Shows a decision tree of how the header analysis is performed and which hex values we are looking for described in the following sections

Step 1. .exe magic bytes: The header analysis starts by reading the first 64 bytes of the given file sample, due to performance. This saves time by not having to read the whole file, when the values that we are interested in the both the first, and the start of second step, are found in this range. We will come back to how we have calculated this value in the second step. Extracting the .exe magic bytes (4D 5A in hex) are done by reading the first 2-bytes from the given file sample, in read binary mode in Python, and for instance converting it to hex values, as seen here:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	**4D**	**5A**														

Table 3.2: Shows how the .exe magic bytes are found in a hex editor.

Step 2. PE signature: According to Microsoft Windows Developer Documentation, the PE signature can be found in the following way:

1. Look at the offset location *0x3C*, here there is a pointer called *e_lfanew*, who has a 4-byte offset to the location of the PE signature. This is the reason for why we are starting by reading 64 bytes instead of 2-bytes or the whole file. There are several ways to show how we get 64 bytes, like counting the cells in the first row, times number of rows (16*4=64), but here we have followed the Microsoft Windows Developer Documentation, and converted the hex value 0x3C to decimal, who is 60 and then added 4 bytes, giving us 64. The values are here written in little endian, meaning that if we look at the table, we have 0x00D00000 giving us D000000 that we need to start reading from right to extract the *e_lfanew* pointer. So, we read the values on the 63 and 64 positions, the 62 and 63 positions, the 61 and 62 positions and the 60 and 61 positions, giving us 0000D000. We then convert the value to decimal, here 53248 to get the start position and 53248 + 4 = 53252 to get the end position [53248, 53252]. Resulting in that we now can read the binary file until 53252 + 2, since we later will look for the CPU architecture.

2. At the offset location, the signature PE\0\0 in ASCII or 50450000 in hex would be found [17].

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	**00**	**D0**	**00**	**00**
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
00000080	09	C9	D5	DF	4D	A8	BB	8C	4D	A8	BB	8C	4D	A8	BB	8C
00000090	CE	A0	E6	8C	4E	A8	BB	8C	4D	A8	BA	8C	49	A8	BB	8C
000000A0	48	A4	DB	8C	4C	A8	BB	8C	48	A4	E1	8C	4C	A8	BB	8C
000000B0	52	69	63	68	4D	A8	BB	8C	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	**50**	**45**	**00**	**00**	4C	01	02	00	BE	08	97	48	00	00	00	00

Table 3.3: Shows how the e_lfanew pointer is found at location 0x3C *(here marked in dark blue),* and that the offset points to the PE signature 50 45 00 00 *(here in light green),* in a hex editor.

Step 3. CPU architecture: According to Microsoft Windows Developer Documentation, the CPU architecture is found in the COFF File Header (Object and Image), starting right after the PE signature [17]:

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. For more information, see Machine Types.

Table 3.4: Shows the table COFF File Header (Object and Image) [17]

The table states here that there is no offset, and that the two preceding bytes is the field *Machine*. Or in other words, that the two bytes immediately after the PE signature, is the *Machine Types*. Jumping further down the Windows Developer Documentation to the Machine Types table, we can see the value for both x64 and x86 in hex [17]:

Constant	Value	Description
IMAGE_FILE_MACHINE_AMD64	0x8664	x64
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386 or later processors and compatible processors

Table 3.5: Shows the table Machine Types from [17]

Using the same example as before we then had added 2 in decimal in order to be able to retrieve the Machine Type. We then just look for the values 4C01 (x86) or 6486 (x64) at the end.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	D0	00	00
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
00000080	09	C9	D5	DF	4D	A8	BB	8C	4D	A8	BB	8C	4D	A8	BB	8C
00000090	CE	A0	E6	8C	4E	A8	BB	8C	4D	A8	BA	8C	49	A8	BB	8C
000000A0	48	A4	DB	8C	4C	A8	BB	8C	48	A4	E1	8C	4C	A8	BB	8C
000000B0	52	69	63	68	4D	A8	BB	8C	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	54	45	00	00	**4C**	**01**	02	00	BE	08	97	48	00	00	00	00

Table 3.6: Shows how the CPU architecture *(here x86)* is found in a hex editor

3.2.2 Sample entropy, image conversion and reshaping

The first thing we are starting with after we have done the header analysis is to create a file hash of the given file with the hashing algorithm MD5, in order to have a unique key representing that sample. We are also when the pre-processing data is inserted into the database creating a unique primary key (unique integer value) that we are using to referring to the given sample within the database structures itself. This is done because it is much clearer to follow an integer value across different kinds of tables in the database, in opposite to a long hash value. Another benefit by this approach is that we are saving storage space, due to a shorter length of the integer value. After the hash is created, we are calculating the file entropy of the given file sample and stores it in the database, in order to be able to use it for a comparison later on when we are evaluating the STAMINA [6] approach against entropy and packer and encryption signature detection.

Then we are as mention earlier, following the suggested approach by STAMINA [6] and how they are pre-processing their dataset, by first reading the given benign or malicious sample in binary, read each and every byte value that directly becomes a pixel value, before this array of pixel (byte) values is reshape and resized according to [6] in order to be a grayscale image, describe in detail under section 2.4.1.2. During this process, we will first be creating the grayscale image for the given benign or malicious sample. Then we are making a copy of the newly created grayscale image, and resizing it according to the Inception V3 [55] models requirement, who is an input shape of (299, 299, 3), meaning as we described earlier in section 2.4.1.2 an image with the height and width of 299, and a colour image. The method used here to resize the grayscale images is bilinear [56] referring to Bilinear interpolation [49], who is the default algorithm that the framework TensorFlow [57] is using. Both the full dimension grayscale image and the resize version is being hashed, in order to create a unique filename to each of the images, and these names are saved in the database. The whole process described in this section is first applied to all the benign samples, and then all the malicious samples.

3.2.3 Packer and encryption signatur detection

In order to be able to detect if a given sample is packed or encrypted, NortonLifeLock [58] have extracted reports on all the samples in dataset 1 and dataset 2 in the data format JSON [59] from VirusTotal [4]. The extraction was done by hashing the given

benign or malicious sample by using the MD5 hash algorithm, and sending the hash as a JSON [59] request to the API VirusTotal [4] provides, getting a JSON [59] result in return. More information about the JSON format and how this process is done in details can be found here [60]. Then the JSON files was looped thru for each sample, looking for if the compiler, packer and crypto signature detecting tool called PEiD [18] was referenced in any of these reports. The whole process described in this section is as in the previous section first applied to all the benign samples, and then all the malicious samples.

3.2.4 Packing our own samples

Since we here are investigating if we are able to detect packed and encrypted samples by using the method STAtic Malware-as-Image Network Analysis [6], we here packed some benign and some malicious samples in both dataset 1 and dataset 2. The chosen packer was UPX [61] version 3.96 because it is easy to install and due to the fact that many malware authors are using this algorithm to pack their malicious files with, as seen in both dataset 1 and dataset 2. UPX [61] was installed by first installing the Package Manager tool (a tool that lets you install programs directly from the console / terminal) Homebrew [62] for macOS. Then we installed the latest version of UPX by writing the command: `brew install upx` [63] in the terminal as seen in fig xx. We tried to extract an convert all the benign samples and 2000 malicious samples that not already where samples we had labelled either packed or encrypted by the reports from VirusTotal [63] in the previous section.

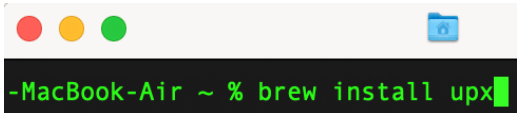


Fig 3.7: Shows how brew is installing UPX

3.3 Create and train machine learning (ML) models

When we are creating the machine learning (ML) models we are trying to follow paper [6] as closely as possible, and their recommended approach for transfer learning called fine-tunning, as describe in more detail under section 2.4.2.1.2. According to paper [7], all the layers before the last polling layers should be frozen, before we are retraining the transfer learning model, referred to as the base_model in the code samples in the appendix. Since we are applying transfer learning to the Inception V3 model, we are freezing all the layers before layer 299 marked here in fig 3.8 in green and the last frozen layer marked in purple, by setting the layers before this layer to not trainable by this code snippet: `layer.trainable = False`.

```
batch_normalization_92 (BatchNo (None, 8, 8, 384)    1152        conv2d_92[0][0]
__________________________________________________________________________________________________
conv2d_93 (Conv2D)              (None, 8, 8, 192)    393216      average_pooling2d_8[0][0]
__________________________________________________________________________________________________
batch_normalization_85 (BatchNo (None, 8, 8, 320)    960         conv2d_85[0][0]
__________________________________________________________________________________________________
activation_87 (Activation)      (None, 8, 8, 384)    0           batch_normalization_87[0][0]
__________________________________________________________________________________________________
activation_88 (Activation)      (None, 8, 8, 384)    0           batch_normalization_88[0][0]
__________________________________________________________________________________________________
activation_91 (Activation)      (None, 8, 8, 384)    0           batch_normalization_91[0][0]
__________________________________________________________________________________________________
activation_92 (Activation)      (None, 8, 8, 384)    0           batch_normalization_92[0][0]
__________________________________________________________________________________________________
batch_normalization_93 (BatchNo (None, 8, 8, 192)    576         conv2d_93[0][0]
__________________________________________________________________________________________________
activation_85 (Activation)      (None, 8, 8, 320)    0           batch_normalization_85[0][0]
__________________________________________________________________________________________________
mixed9_1 (Concatenate)          (None, 8, 8, 768)    0           activation_87[0][0]
```

```
                                                                 activation_88[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 8, 8, 768)    0           activation_91[0][0]
                                                                 activation_92[0][0]
__________________________________________________________________________________________________
activation_93 (Activation)      (None, 8, 8, 192)    0           batch_normalization_93[0][0]
__________________________________________________________________________________________________
mixed10 (Concatenate)           (None, 8, 8, 2048)   0           activation_85[0][0]
                                                                 mixed9_1[0][0]
                                                                 concatenate_1[0][0]
                                                                 activation_93[0][0]
==================================================================================================
Total params: 21,802,784
Trainable params: 395, 777
Non-trainable params: 21,409,056
```

Fig 3.8: Showing the last layers in the Inception V3 model, where layer 299 is marked in green as trainable and layer 298 in purple as the last frozen layer.

The model Inception V3 [55] model is then added to a sequential model, with an average polling layer and a dens layer with one output node. The chosen optimiser is Adam.

```
Layers frozen until: conv2d_93
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 8, 8, 2048)        21802784
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 2049
=================================================================
Total params: 21,804,833
Trainable params: 395,777
Non-trainable params: 21,409,056
_________________________________________________________________
```

Fig 3.9: Showing the Convolutional Neural Network

The next step is to divide the dataset into training and validation sets by first applying a stratified k-fold split with shuffling. Meaning that each class samples is also shuffled before the dataset is divided into smaller pieces called batches, by setting the `shuffle=True` in the stratified k-fold [44]. We will come back to the reason for the shuffling and why we are using a stratified k-fold split under section 2.3, but for now, the reason is to divide the dataset into training and validation sets. The machine learning algorithm is not able to process all the samples it is been giving if we have a large dataset, hence it is divided into x smaller pieces, that it is being served one after the other, until the last one. The creation of these smaller pieces, batches is the next process after the k-fold split. When we are creating these batches, we are also applying a random shuffling. Each and every image is here converted according to Chen [7] from grayscale images to RGB by replicating the one colour channel three times. Here in this paper, we are letting the TensorFlow [57] framework take care of this by using the method `tf.io.image.decode_png(image, channels=3)` [64], where the image is the grayscale image, and the three channels represents RGB as described earlier under section 2.4.1.2.

Then we are creating a machine learning model for each k in the k-fold and starts to train the first model by iterating thru the training and validation set with a low learning rate, the number of epochs and early stopping according to table 3.10, in order to not overfit the model. This is done for all k (number of) models.

Datasets	K-fold splits	Learning rate	Number of epochs	Early stopping
Dataset 1	5	0.01	1	1

Table 3.10: Shows the k-fold splits, learning rate, number of epochs and early stopping for dataset1

3.4 Evaluate the machine learning (ML) models and results

In this section we will be describing how we can measure the performance of machine learning (ML) algorithms and the predictions from these models.

We have here chosen to use a stratified k-fold split, in order to be able to evaluate how well the machine learning models are performing by measuring them against each other. The reason for not using an ordinary k-fold, is due to the ratio in our dataset between benign and malicious samples, where we have a lot more samples of malicious files, compared to benign files, as described in section 3.1 Dataset. Therefore, in order ensure that each of the k folds are both trained and validated on benign and malicious samples, we have selected a stratified k-fold split, who seeks to preserve the ratio of benign and malicious samples in each fold as closely as possible, as described under section 2.3 and visualized in the table 3.11. We are creating a machine learning model for each k in the k-fold split, as seen in the visualization in the table below for an ordinary k-fold, (due to a simpler drawing), but also applies for a stratified k-fold. This means that the result of the performance for the whole dataset can only be seen by looking at the validation sets for all models. Here 1-10.

	Dataset (100%)	
ML	Train (90%)	Val (10%)

ML 1	Val	Train	Train	Train	Train	Train	Train	Train	Train	Train
ML 2	Train	Val	Train	Train	Train	Train	Train	Train	Train	Train
ML 3	Train	Train	Val	Train	Train	Train	Train	Train	Train	Train
ML 4	Train	Train	Train	Val	Train	Train	Train	Train	Train	Train
ML 5	Train	Train	Train	Train	Val	Train	Train	Train	Train	Train
ML 6	Train	Train	Train	Train	Train	Val	Train	Train	Train	Train
ML 7	Train	Train	Train	Train	Train	Train	Val	Train	Train	Train
ML 8	Train	Train	Train	Train	Train	Train	Train	Val	Train	Train
ML 9	Train	Train	Train	Train	Train	Train	Train	Train	Val	Train
ML 10	Train	Train	Train	Train	Train	Train	Train	Train	Train	Val

Table 3.11: Shows how an ML model is related to the number of k in k-fold cross validation split for k 10, each row represents an iteration k, from [1-k], and each cell either belongs to the training or validation set.

We will then get how well each machine learning algorithm is performing by the loss and binary accuracy, and the average.

Evaluating the results

The formulas used to calculate the true positive, true negative, false positive and false negative rate. The accuracy, precision and recall. To able to say something about how accurate the machine learning model is, its precision and performance.

Calculating the True Positive Rate
$$TPR = P(A\|I) = \frac{TP}{TP + FN}$$

Table 3.12: Shows the formula for calculating the true positive rate

Calculating the True Negative Rate
$$TNR = P(\neg A\|\neg I) = \frac{TN}{TN + FP}$$

Table 3.13: Shows the formula for calculating the true negative rate

Calculating the False Positive Rate
$$FPR = P(A\|\neg I) = \frac{FP}{FP + TN}$$

Table 3.14: Shows the formula for calculating the false positive rate

Calculating the False Negative Rate
$$FNR = P(\neg A\|I) = \frac{FN}{FN + TP}$$

Table 3.15: Shows the formula for calculating the false negative rate

Calculating the Accuracy
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Table 3.16: Shows the formula for calculating the accuracy

Calculating the Precision / Positive Prediction Value
$$Precision = \frac{TP}{TP + FP}$$

Table 3.17: Shows the formula for calculating the precision / positive predictive value

Calculating the Recall
$$Recall = \frac{TP}{TP + FN}$$

Table 3.18: Shows the formula for calculating the recall

4 Experiments and results

This chapter presenst how the experiment was conducted and the results. The chapter is divided into environment and experiment setup. Where the environment describes the hardware and software used. The experiment setup the results from the machine learning models, and the results entropy and packer signature detection.

4.1 Environment

Hardware:

The experiment was performed on a MacBook Air 13-inch Mid 2013 with the following specification:

- Operating system: macOS Big Sur Version 11.4 Beta (20F5046g)
- CPU: 1.3 GHz Dual-Core Intel Core i5
- Memory: 4GB 1600 MHz DDR3
- Graphics: Intel HD Graphics 5000 1536 MB
- SSD: 128 GB
- HDD: 1 TB External Western Digital Elements

Software:

The software installed on the MacBook Air and Windows PC to perform the experiment was:

- brew (Homebre version 3.1.8 git revision 2d4d7b2e8e: last commit 2021-05-22)
- dbdiagram.io: Web-tool for drawing Entity-Relationship diagrams *(ER diagrams),* or in other words visulaizing how the the database structure looks like
- DB Browser for SQLite version 3.12.1: Database tool for viewing data and write SQL queries
- PyCharm 2021.1.1 (Community Edition) Build #PC-211.7142.13: Python IDE
- Visual Studio Code Version 1.55.2 (Universial)
- Python version 3.8.7.
    - Installed Packages see Appendix 23
- UPX version 3.96

4.2 Experiment setup

Dataset 1 was trained with a k = 5, hence we trained 5 models, each model was trained with an epoch of 3, and a learning rate of 0.01. There are total 14624 samples in the dataset where 11680 samples was in the training set and 2944 samples was in the validations set. The last frozen layer was 299 (here conv2d_187). In the ML models there are 21,804,833 parameters in total, we froze 21,409,056 and trained 395,777 parameters.

4.2.1 Machine learning performance

The bar chart below in fig 4.1 shows that each of the 5 machine learning models are performing more or less equal, as we also can see by the numbers in table 4.2 below, that the plotted bar chart is based on. However, we see that the model is performing a little bit better on the training set, than on the validation set and the loss function.

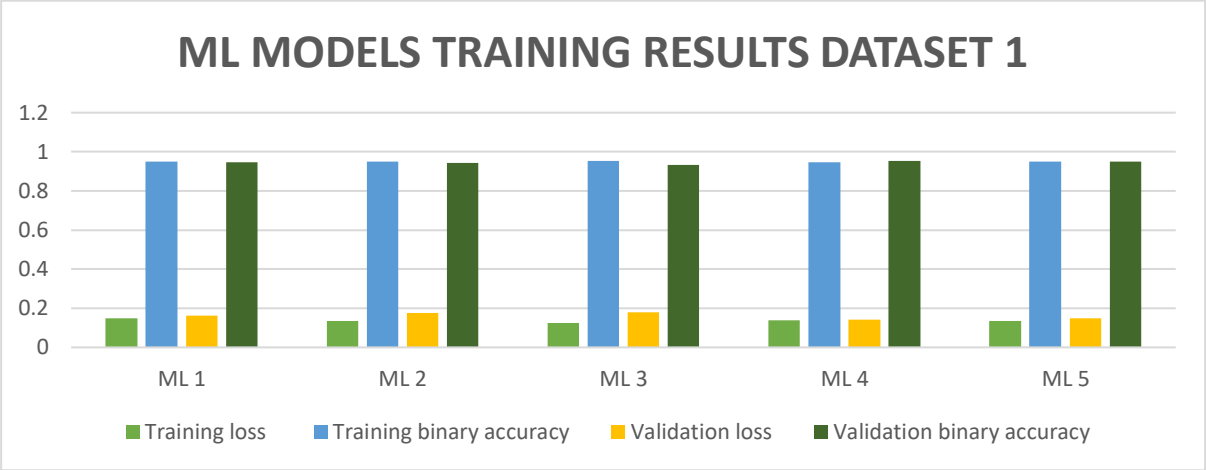


Fig 4.1: Showing the performance of our 5 trained ML models

	Training		Validation	
	Loss	Binary accuracy	Loss	Binary accuracy
ML 1	0.15	0.95	0.16212	0.9458
ML 2	0.1338	0.951	0.1743	0.9444
ML 3	0.1242	0.9543	0.1791	0.934
ML 4	0.1389	0.9478	0.1432	0.9527
ML 5	0.1345	0.95	0.1477	0.9502
Average	0.13628	0.95062	0.161284	0.94542

Table 4.2: Showing the training and validation loss and binary accuracy for the five models

The table below shows the progress for each epoch. The 5 machine learning models were only trained for 3 epochs each. We can here see that there is a significant improvement in loss from the 1 epoch until the 2 epochs, but between the 2 epoch and the 3 epoch, there are not so much improvement.

		Training		Validation	
ML / Epoch		Loss	Binary accuracy	Loss	Binary accuracy
ML 1	1	0.2905	0.8889		
	2	0.1575	0.9428		
	3	0.15	0.95	0.16212	0.9458
ML 2	1	0.3457	0.8671		
	2	0.1773	0.9378		
	3	0.1338	0.951	0.1743	0.9444
ML 3	1	0.3189	0.8757		
	2	0.1578	0.9415		
	3	0.1242	0.9543	0.1791	0.934
ML 4	1	0.3134	0.8806		
	2	0.1634	0.9397		
	3	0.1389	0.9478	0.1432	0.9527
ML 5	1	0.3040	0.8816		
	2	0.1774	0.9356		
	3	0.1345	0.95	0.1477	0.9502

Table 4.3: Showing the progression of each model and epoch (3 per model), and that the validation is only done for the last model

Machine learning benign and malicious classification

As described under the definition of the Sigmoid function in the background theory in chapter 2, we will get a result from our binary classification that is in the range [0-1], hence we here say that a benign file will be 0 and malicious file will be 1. When we then get the result from the Sigmoid function, the threshold will therefore be that if the return value is below 0.5, the file is categorised as benign, otherwise it is categorised as malicious [39].

Benign	Predicted value from the Sigmoid function **< 0.5**
Malicious	Predicted value from the Sigmoid function ≥ **0.5**

Table 4.4: Showing how we predict that a sample is benign or malicious [39]

		Actual	
		Malicious	Benign
Predicted	Malicious	TP	FP
	Benign	FN	TN

		Actual	
		Malicious	Benign
Predicted	Malicious	9861	343
	Benign	428	3960

Fig 4.5: Shows the confusion matrix for benign and malicious samples

By following fig xx and fig xx, by looking at the benign and malicious samples, we therefore get for the benign samples TN = 3960, FN = 428, and for the malicious samples TP = 9861, FP = 343 by the following SQL 1 Appendix 1. By visualizing the confusion matrix in a chart, we get the two following in pie charts. It is important here to remark that these results are calculated by the confidence interval for benign $0 < 0.5$ and malicious $1 \geq = 0.5$. Meaning that the closer the value is to 0, the machine learning model would think that sample is benign and the closer the value is to 1, the machine learning model would think that the sample is malicious. Within these two intervals there will therefore be values that are close to 0.5 in both cases.

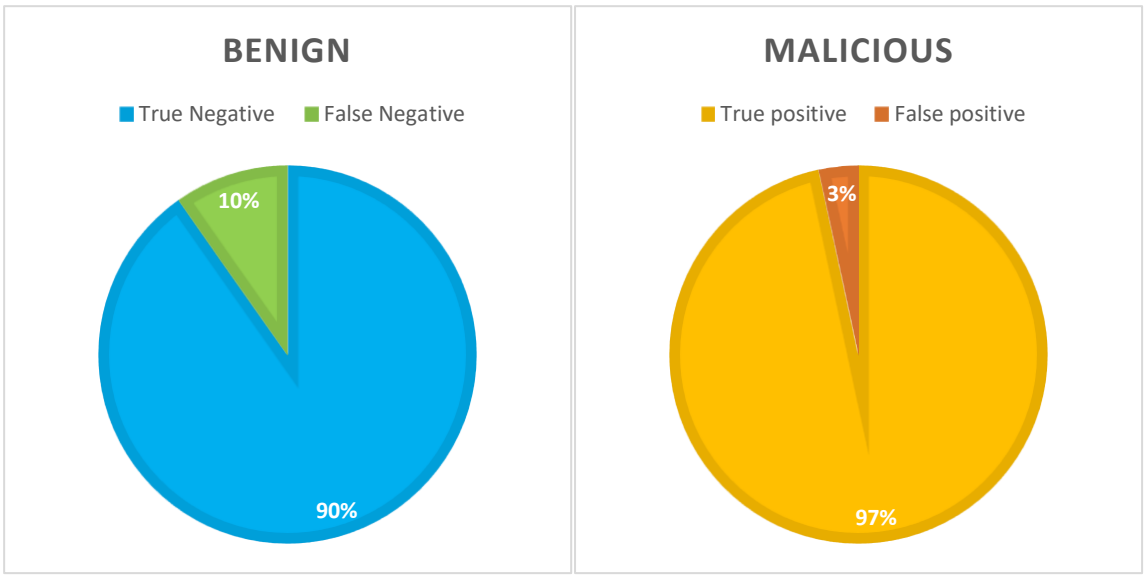


Fig 4.6: Shows the Benign TN and FN, malicious TP and FP

The pie chart 4.6 above to the left shows that there are 90% of the benign files that are correctly classified and 10 % that are wrongly classified malicious. The pie char 4.6 above to the right show that 97% of the malicious files are correctly classified, while 3% er wrongly classified as malicious.

In order to later be able to compare Inception V1 [47] and the results from STAMINA [6] with our Inception V3 [55] model, we calculated the accuracy, false positive rate, precision and recall in table 4.7 below as in paper [6].

Benign and malicious samples			
Accuracy	**False positive rate**	**Precision**	**Recall**
$\frac{9861+3960}{9861+3960+343+428} = \mathbf{0.947}$	$\frac{343}{343+3960} = \mathbf{0.079}$	$\frac{9861}{9861+343} = \mathbf{0.966}$	$\frac{9861}{9861+428} = \mathbf{0.958}$

Table 4.7: Shows how the accuracy, false positive rate, precision and recall is calculate for the machine learning models

In order to better show how the models are performing, we have drawn the chart below in figure 4.8. Here it is possible to see how many samples in either cases that are close

to their respectively boundaries categorization them either benign or malicious, and how they are evolving in both directions. This figure shows the benign samples where the values below 0.5 is the Benign True Negative and continues as the Benign False Negative, from greater than or equals 0.5 to 1. It also shows the malicious samples where the values in the range from greater than or equal 0.5 to 1 is the Malicious True Positive and the values in the range 0 to lower than 0.5 is the Malicious False Positives.

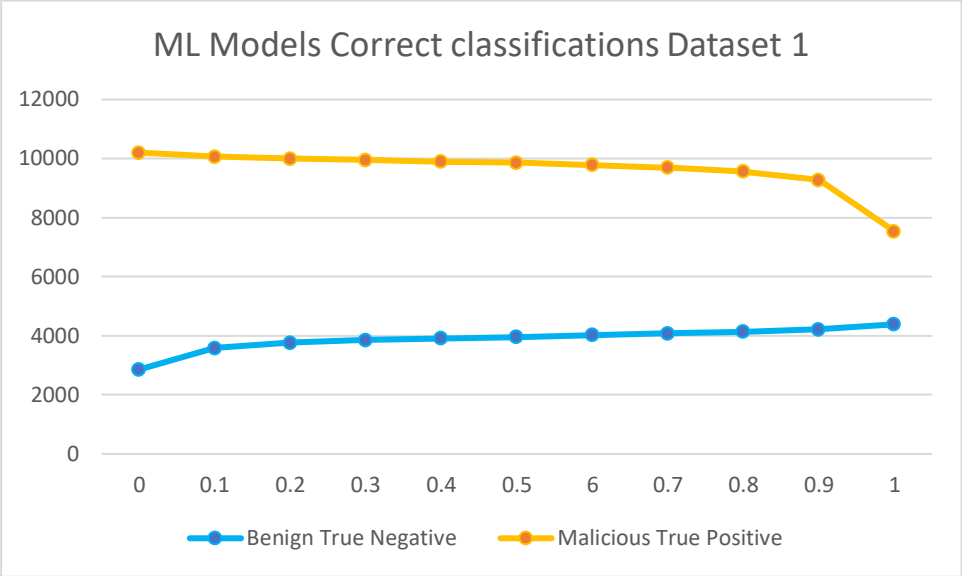


Fig 4.8: Showing Benign True Negative and Malicious True Positive

In fig 4.8 there might be files that are very close to either side of 0.5 and this might not mean that they are wrongly classified, due to the fact that a benign file might have the same code as a malicious file, the only difference is that the malicious file uses the code for malicious actions, were the benign file is not. Like everything else code / programs can also be used for the greater good. But it also actually be the case that they are wrongly classified do too that they are too close to the threshold.

Machine learning packed benign and malicious classification

Looking at the benign packed samples that was labelled benign, and how many of them that was labelled malicious. Then we did the same for the malicious files. Benign: TN = 1890, FN = 286. Malicious: TP = 4649, FP = 237, by the following SQL 2 Appendix 2.

		Actual: Packed	Actual: Not packed
Predicted	Packed	TP	FP
	Not packed	FN	TN

		Actual: Packed	Actual: Not packed
Predicted	Packed	4649	237
	Not packed	286	1890

Fig 4.9: Shows the confusion matric for the packed and not packed benign and malicious samples

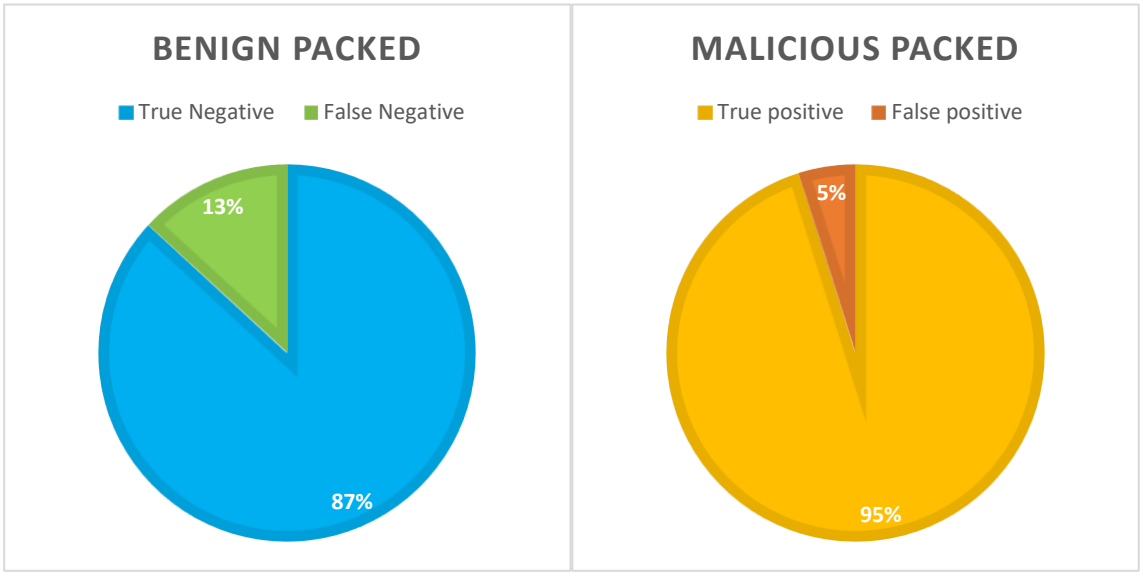


Fig 4.10: Showing the benign packed TN and FN, and the malicious packed TP and FP

Looking closer at the 237 malicious packed false positive (FP) samples, counting the different packers and grouping packers that uses the same algorithm, the packer UPX, is the only one that has an occurrence over 8, giving us 155 samples. For the true positive (TP) samples that are malicious, we get ASProtect 1294 samples, UPX 1085 samples, PECompact 433 samples and ASPack 306 samples. The other once are below 133 in count. In the benign true negative (TN) packed samples, we have UPX 1748 samples, and the rest is below 40 in count. In the false negative (FN) benign samples, we have UPX with 269, the rest is below 7 in count. (Here used SQL 3 Appendix 3 and 4 Appendix 4)

4.2.2 Entropy and packer signature detection

If we are only looking at the file entropy and packer signature detection, by using the table from paper [24]. we get the following confidence interval: not packed [0.0-6.800] and packed or encrypted executable [6.801-8.0]. In order to get all samples that are predicted packed and actually are packed, we will use the confidence interval and the detected packers by VirusTotal [4], but discard any packer names that contains "Microsoft" and ".NET" in TP and TN, in order to remove these compilers (not packed samples). We then get TP = 5570 by following the SQL query 5. seen in the Appendix 5 with the optional arguments 3 and 4. In order to get the samples that are labelled packed, but are not packed, we take alle the samples that has the entropy range [6.801-8.0] and subtracts the samples that are packed. The SQL 5 Appendix 5 with argument 3, gives us all the samples 10872 – 5570 = 5302 FP. The predicted not packed samples that are packed, are found by getting all the samples that are predicted as not packed [0.0-6.800] and then selecting the packed samples. By using the SQL 5 Appendix 5 and argument 4 we get the 963 samples in that range that are packed, hence FN. To get the true negative we take all not packed [0.0-6.800] samples and removes the FN samples, and then the compilers containing "Microsoft" and ".NET". Using the SQL 7 Appendix 7 and the argument 4, we get 3716 samples that are labelled as not packed, subtracting 3716 – 963 gives us 2753. Removing the compilers containing "Microsoft" and ".NET", respectively 362 and 5 = 367, by the SQL 6 Appendix 6 and including argument 5 we get all compiler signatures for "Microsoft" and, making a new query with argument 2 for the

".NET". We therefore get 2753 – 367 = 2386 TN, resulting int the confusion matrix in figure 4.11 below.

		Actual	
		Packed	Not packed
Predicted	Packed	TP	FP
	Not packed	FN	TN

		Actual	
		Packed	Not packed
Predicted	Packed	5570	5302
	Not packed	963	2386

Fig 4.11: Show a confusion matrix for packed and not packed samples in Dataset 1 by using entropy and packer signature detection

The confusion matrix can be visualized in chart two pie charts, one for packed and on for not packed. As we can see, there are 49% of the packed files that were predicted packed but not were packed. When it comes to the not packed samples, there are 29% that are detected as not packed, but was packed.

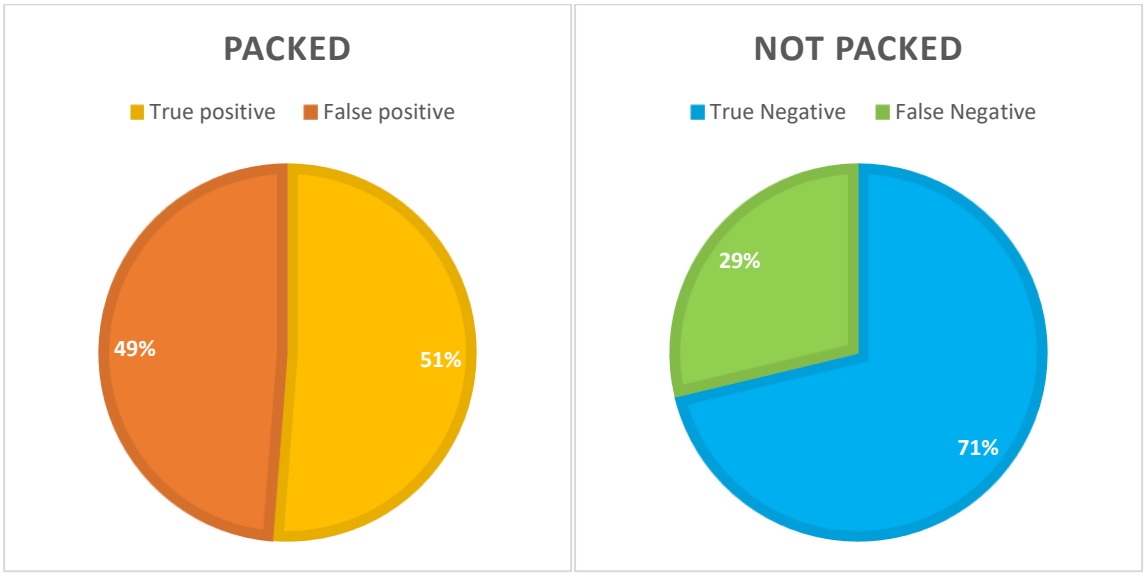


Fig 4.12: Shows the packed TP and FP and not packed TN and FN, when using file entropy and packer signature detection

Calculating the accuracy, precision and recall for the file entropy and packer signature detection in the table 4.13 below

Benign and malicious samples		
Accuracy	**Precision**	**Recall**
$\frac{5570+2386}{5570+2386+5302+963} = \mathbf{0.559}$	$\frac{5570}{5570+5302} = \mathbf{0.511}$	$\frac{5570}{5570+963} = \mathbf{0.852}$

Table 4.13: Shows Fig xx Showing the accuracy, precision and recall for the file entropy and packer signature detection

Looking further into the packed and not packed samples, and the benign and malicious samples in each one, we get the following. For the benign samples, we can calculate the true positive (TP) by using the same approach as above but including the argument 2 and 1 as the file_type_id, hence 5 in SQL Appendix 5, argument 2, 3, and 4, resulting in 1755 TP. For malicious samples we do the same, but set argument 2, file_type_id = 2, giving us 3815 TP. Continuing with the FP for benign and malicious samples, we seek the samples that are labelled packed, but are not packed, we take alle the samples that has the entropy range [6.801-8.0], and benign or malicious and subtracts the samples that are packed. The SQL 5 Appendix 5 with argument 2, file_type_id = 1 and argument 3, gives us for all the benign samples 2190 – 1755 = 435 FP, and for the malicious samples file_type_id = 2, giving 8682 – 3815 = 4867 FP. Then we get the false negative (FN) for bot benign and malicious samples by getting all the samples that are predicted as not packed [0.0-6.800], either benign or malicious and then selecting the packed samples. By using the SQL 5 Appendix 5 and argument 2 as file_type_id = 1 and argument 4, we get 415 FN for benign and by using file_type_id = 2, we get 548 malicious FN. Lastly getting the true negative (TN) for both benign and malicious samples we then take all not packed [0.0-6.800] samples benign and malicious and removes the FN samples, and then the compilers containing "Microsoft" and ".NET". Using the SQL 5 Appendix 5 and the argument 2 as file_type_id = 1 and argument 4, we get 2195 benign samples that are labelled as not packed, subtracting 2195 – 415 = 1780. For malicious we get 1521 malicious samples that are labelled as not packed by file_type_id = 2, giving us 1521 – 548 = 973. Then we need to remove the compilers containing "Microsoft" and ".NET", for benign 67 and 5, for malicious 295 and 0 by SQL 5 Appendix 5 and including argument 1 and argument 2 for benign file_type_id = 1 and malicious file_type_id = 2, we get all compiler signatures for "Microsoft" and, making a new query with argument 5 for the ".NET". Benign TN will then be 1780 – (67 + 5) = 1708, and malicious 973 – 295 = 678 TN. This results in the confusion matrixes below in figure 4.14 for benign and malicious samples.

Benign

		Actual: Packed	Actual: Not packed
Predicted	Packed	1755	435
	Not packed	415	1709

Malicious

		Actual: Packed	Actual: Not packed
Predicted	Packed	3815	4867
	Not packed	548	678

Fig 4.14: Showing a confusion matrix for benign to the left and malicious to the right

Visualizing the confusion matrix in fig. 4.14, we get the pie charts below in fig. 4.14. We can here see that the approach with calculating the entropy according to paper [24] and including a packer signature detection, this works best for benign files and not so good for malicious files for our small dataset.

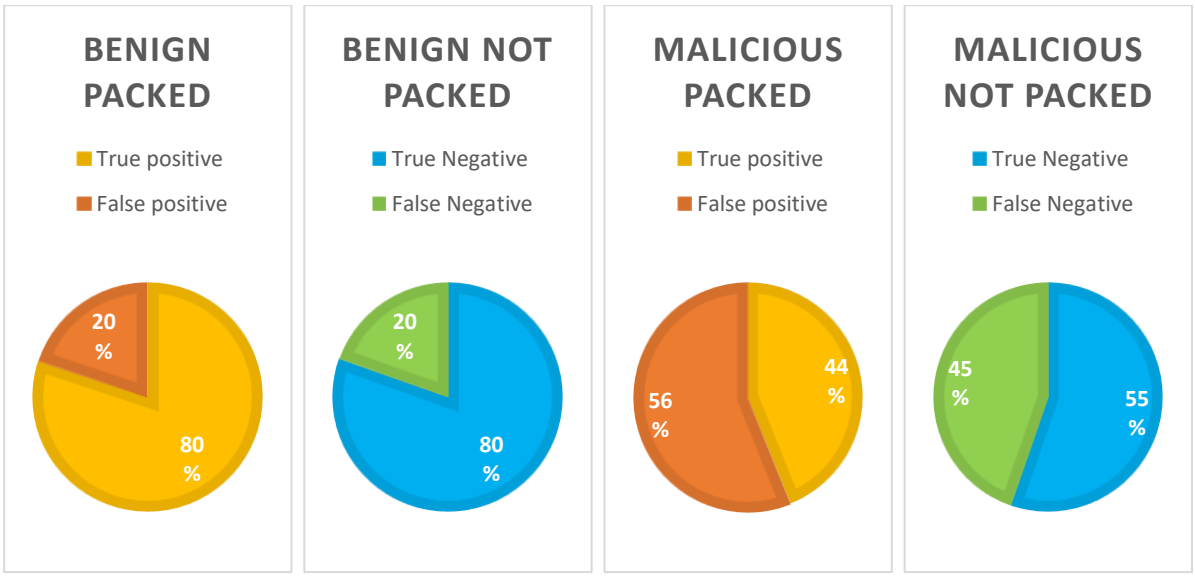


Fig 4.15: Showing benign packed TP and FP, benign not packed TN and FN, malicious packed TP and FP, and malicious not packed TN and FN

5 Discussion

This chapter discusses the results presented in chapter 4, and tries to discover any strengths and wekneses that might have been affecting the experiments we have performed and therfore the results we have gain by our method. We therefore start by discussin the implementation of STAMINA and performance, then STAMINA vs entropy analysis and packer signature detetcion, what STAMINA detects and lastly future works.

5.1 Implementation of STAMINA and performance

STAMINA were here implemented by using the open-source machine learning software library TensorFlow [57], who again uses the deep learning API Keras [65]. In Keras, there are several deep learning models that are available called Keras Application to for instance fine-tuning

, feature extraction or predictions [66]. According to [6] they used Inception V1 as there fine-tuning model, but that model is not available in Keras Applications [66], hence we used the Inception V3, in order to at least be using a model that is based on the same model as Inception V1. Based on Bensaoud [67], who compared which machine learning models that was best suited for malware detection (Windows PE binaries) and binaries converted to grayscale images. His results shows that the VGG16 is the least accurate model with 15.92 % accuracy, oppose to Inception V3, who was in the other end of this scale, at the top with an accuracy of 99.24 %. The second worst model were ResNet50 with 35.10%, then the nine other models (not included Inception V3) range from 77.22%-99.11%. The best models based on his research looks to be convolutional neural networks, with the range 98%-99.24% [67]. Based on Bensaoud [67], we therefore does not think that choosing Inception V3 has had any negative impact on our experiment, oppose to if we have used Inception V1. In order to try to verify this, we calculated the same measures as STAMINA [6], were their results are 99.07% accuracy with a false positive rate 2.58%, the precision 99.09% and a recall at 99.66%. Opposed to our 94.7 % accuracy, 7.9% false positive rate, 96.6 % precision and 95.8% recall, as show in table 4.7 Our performance are not as good as their machine learning model, but considering that our test set consists of 14592, divided into 4388, and 10204, where they have 157837 + 39781 = 197618 benign samples and 495077 + 89529 = 584606 malicious samples, it is not so bad after all. According to the stratified k-fold cross-validation results in fig 4.1, our models are generalizing well, but due to such a small dataset, it is not possible to conclude that this is the case in general. The models are also here only training for 3 epochs in order to not overtrain them, compared to STAMINA where they picked their best model at the 10^{th} epoch, to avoid overfitting [6]. We saw that only 1 epoch gave a dramatic increase in accuracy on our small dataset, but from the 2 epochs to the 3 epochs, there was not much improvement in the accuracy according to table 4.2.

5.2 STAMINA vs entropy analysis and packer signature detection

Comparing STAMINA against entropy analysis and packer signature detection, we see that the entropy analysis and signature detection results for the packed samples TP = 51 % and FP = 49 %. So not very good. Looking at the not packed samples, it works better with a TN = 71% and FN = 29%. By dividing it into benign and malicious samples, we see in figure xx that we get benign packed TP = 80%, FP = 20%, not packed TN = 80% and FN = 20%. For the malicious packed samples, we got TP = 44% and FP = 56%, and not packed TN = 55% and FN = 45%. It looks therefore like this is working better for the benign files, than the malicious files. This can be because there are more packed malicious files than there are packed benign files, hence they are more often associate with a packer. Another possibility is that the threshold values used from paper [24] were to wide. When it comes to STAMINA and we have malicious packed TP = 95% and FP = 5, benign packed TN = 87% and FN = 13%. It therefore looks like STAMINA is performing much better, but it is hard to conclude that, due to not have a representative enough benign set to compare it against, almost only UPX packing. Another consideration here is that the dataset is to small to be able to generalize. The four-step verification approach mention in paper [22] might also be a better compression method, due to their method who is checking the entry point section of the file, then looks for a packer signature, then the WRITE attribute, and lastly performs an entropy analysis of the entry point section only.

5.3 What STAMINA detects

Our machine learning models shows that STAMINA is capable of detecting benign and malicious samples based on fig 4.6. It looks based on the bar chart in figure 4.1 that the models are generalizing well, due to the high accuracy and low loss on each model, and that they are very close to each other. However, the small dataset here makes it difficult to alone draw that conclusion in general. Most likely it has not generalized enough due to too few samples and will therefore have a poor performance in a real-world situation, by presenting the machine learning models to new samples that are far away from the samples in the training and validation set. It might also look like STAMINA is detecting packing, but again here we have to little samples, in this case that are packed in the benign set, in order to draw that conclusion. We can see from the pie chart 4.10 for the benign packed samples that TN = 87% and FN = 13 %, and the pie chart 4.10. for the malicious packed samples, that TP 95% and FP = 5%. Overall, very good result. Taken into consideration that we here only have looked at the benign samples that were correctly labelled benign, and how many that were packed, and then how many benign files that were wrongly labelled malicious and packed. Lastly doing the same for the malicious files, these numbers would just be indicating that STAMINA might detect packers. If we look at the packer UPX we see that TP = 1085 samples and TN = 1748 FN = 269, we might therefore say that there is a small trend, at least when it comes to the UPX packer, but there is also a bias, since the benign samples are mostly packed with UPX. Else we can see that in the TP we have 1294 samples that are from ASProtect, 433 samples PECompact and 306 samples that are from ASPack. The rest of the benign and malicious samples that are packed have too few occurrences.

Another consideration here is that in order to be able to detect packed samples, we here used reports from VirusTotal [4], who again bases their results on a tool called PEiD [18], in order to detect packers, cryptos and compiler signatures. This was a convenient way to gain such information in a short amount of time, due to the easiness of extraction from VirusTotal [4] and the JSON reports on each file sample. However, it is not necessarily the best approach, because we here only are relaying on one single tool to do

the job, hence it might not alone be up to the task, due to the limitations of the tool. In this case, it might not have all the signatures that we have in our dataset, hence packed and encrypted samples would go under the radar. A better approach would have been to relay on several tools, measuring them against each other, and also to self be writing the code that looks for the compiler signature, since the PEiD [18] signature database can be located here at GitHub [25], and newer signatures could also be added. Due to our limited time and resources, using several packer identifiers or to implement code to look for the signature by our self was not applicable, hence VirusTotal [4] was chosen instead. Another aspect that needs to be considered here is the fact that one signature might not necessarily be unique for the given compiler version. Hence for instance a UPX packer version 3.96 might be detected as 2.90, since they have the same signature, or that we are missing a part of the signature in order to detect and reveal that the newer version actually is 3.96. Therefore, we treated all the files from the same packer vendor as the same packer, hence UPX 2.90 and 3.96 is just UPX.

5.4 Future work

After the implementation of STAMINA and this experiment, there are a few things we think that further research should look into, divided into the entopy analysis described in paper [22], our implementation of a stratified k-fold cross-validation, how to train machine learning models with early stopping, and comparing fine-tuning vs feature extraction.

Our entropy analysis is not performing very well and further research should look into if the suggested approach in paper [22] is a better measurement against STAMINA, due to its four step verification tests to determine if the malware sample is packet or not. The test consists of first checking the entry point section of the file, then looks for a packer signature, then the WRITE attribute, and lastly it will perform an entropy analysis of the entry point section only.

Here we implemented a stratified k-fold cross-validation in order to be able to measure the performance of STAMINA. Another approach that might also be beneficial in order to get the best performing machine learning model, would be to train a model until it has a good performance (low loss and a hight accuracy), and first then use a stratified k-fold cross-validation to validate if that is true or not. The stratified k-fold cross-validation should then create k models from the good performance model, in order to validate and test them. We therefore think that this method is worth looking into and compare against our stratified k-fold cross-validation.

There are several approaches that can be applied when training a model in order to not be overfitting the model. Here, since we were fine-tuning a pre-trained neural network, we adjusted either the learning rate or the number of epochs manually, by increasing or decreasing these values (one at the time), in order to be able to predict how long the training of the model would take, due to that we have a limited time frame to do our research. Another approach that needs to be looking further into is called early stopping, and our framework therefore has that built in. The ability to set an early stop, is by the early patience variable in the model class. In our framework we set it to monitor the binary_accuracy, but this can be change to the val_loss etc. [68]. For instance, the early patience can be set to 3, meaning here with our binary_accuracy that means that if we are not improving in 3 epochs (then min_delta = 0.001 e.g. the binary_accuracy must be at least improving by 0.001 to count as an improvment) it will stop [68]. It is also then important to set a high epoch like 100, otherwise it might stop before on the set epoch [69].

According to [6] a feature extraction model is not applicable to use with STAMINA, eventhog that is out of the scoup for this thesis, we implementes a feature-extraction

model in our framework, so other researchers would be able to further investigate this. There is also possible to switch out the Inception V1 and replace it with an other model from Keras as seen in the list here [66] for fine-tuning or for feature extraction from from TensorFlowHub [70].

6 Conclusion

In this master thesis we were able to recreate STAtic Malware-as-Image Network Analysis (STAMINA) by using the open-source machine learning software library TensorFlow [57] and the deep learning API Keras [65]. This is a valuable contribution for future research into this method, by giving a valuable insight into many of the challenges faced during the implementation and when using STAMINA. Our contribution will therefore give researchers the benefit of getting STAMINA quick and easy up and running, in order to be able to focus on the method. The proposed method in this paper shows that we are able to get a 94.7 % accuracy, 7.9% false positive rate, 96.6 % precision and 95.8% recall, on our small dataset, compared to STAMINAs [6] much larger dataset, and their performance 99.07% accuracy, with a false positive rate 2.58%, the precision 99.09% and a recall at 99.66%.

We were also able to detect the packer UPX in both the benign and malicious samples, with a TP = 1085 and a TN = 1748 described in section 4.2.1. For the malicious files we also saw ASProtect with 1294 samples, PECompact with 433 samples and ASPack with 306 samples, the other packers and cryptos where too few to be able to draw any conclusion about. However, this result is not representative enough. In order to get a better detection, we would also recommend other researchers to implement a packer signature detection tool themselves, by using the information found under section xx. describing how a packer signature detection tool, either looks at the entry_point or the whole file, in order to detect a compiler, crypto or packer signature. By using this information and the PEiDs [18] database [25], it is possible to achieve this. The reason for this recommendation is to better have control of the packer signature detection process and be able to add new compiler, crypto and packer signatures, along with the ability to tag them also, as for instance compiler, crypto and packer. This will make it easier later on when comparing results and prevent having to add ask for all files containing "Microsoft" and ".NET" like we had to, in order to remove files that were not packed.

There is not enough information in our small dataset to say if STAMINA is better than entropy and packer signature detection when it comes to packer detection, it looks that way from pie chart 4.10, but there are not enough benign packed files and packed files in general to draw that conclusion. However, we can see that STAMINA has a better performance when it comes to detecting benign and malicious files, with an accuracy of 0.947 compared to 0.559. We can conclude that STAMINA detects benign and malicious files and has a great performance on our small dataset. We also see a small trend when it comes to detecting the packer UPX, but since the benign dataset has very few other packers, and is very small in general, this result is not representative. However, this result gives us a pinpoint on that this should be further looked into by other researchers. The dataset should then be closer in size to the one in STAMINA and if possible, have a 50/50

benign malicious ratio. Along with several different packers for both benign and malicious samples.

Appendices

Appendix 1: SQL getting all samples that are benign or malicious

#	**SQL 1. Looking at all samples**
	Base SQL with optional arguments [2-5] below, here for experiment 7
1	SELECT * FROM sample_filtype_view, experiment_results WHERE sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
2	AND sample_filtype_view.file_type_id = 2
	Get all samples that are predicted packed and are packed
3	AND sample_filtype_view.original_file_entropy >= 6.801 AND sample_filtype_view.original_file_entropy <= 8.0
	Get all the samples that are predicted as not packed
4	AND sample_filtype_view.original_file_entropy >= 0 AND sample_filtype_view.original_file_entropy <= 6.800

Appendix 2: SQL for getting TP, FP, FN, TN for benign and malicious packed samples

#	**SQL 2. Gets True Positive (TP), False Positive (FP), False negative (FN), and True Negative (TN) for benign and malicious packed samples**
	Base SQL below, here for experiment 7
	SELECT * FROM sample_filtype_view, virus_total_packer_cryptor_compiler_view, experiment_results WHERE sample_filtype_view.sample_id = virus_total_packer_cryptor_compiler_view.sample_id AND sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
1	AND sample_filtype_view.file_type_id = 1
	Here we then get the following: **Benign TN = percentage < 0.5 FN = percentage >= 0.5** **Malicious TP = percentage ≥ 0.5 FP = percentage < 0.5 AND percentage >= 0**
2	AND experiment_results.malicious_percentage < 0.5

Appendix 3: SQL finding all packers

#	**SQL 3. Finding packers**
	Base SQL with optional arguments [2-5] below, here for experiment 7
1	SELECT * FROM sample_filtype_view, virus_total_packer_cryptor_compiler_view, experiment_results WHERE sample_filtype_view.sample_id = virus_total_packer_cryptor_compiler_view.sample_id AND sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
2	AND sample_filtype_view.file_type_id = 1
	Here we then get the following: **Benign TN = percentage < 0.5 FN = percentage >= 0.5** **Malicious TP = percentage ≥ 0.5 FP = percentage < 0.5 AND percentage >= 0**
3	AND experiment_results.malicious_percentage < 0.5

Appendix 4: SQL for counting packers

#	**SQL 4. Counting packers**
	Base SQL with optional arguments [2-5] below, here for experiment 7
1	SELECT virus_total_packer_cryptor_compiler_view.name, COUNT(virus_total_packer_cryptor_compiler_view.name) FROM sample_filtype_view, virus_total_packer_cryptor_compiler_view, experiment_results WHERE sample_filtype_view.sample_id = virus_total_packer_cryptor_compiler_view.sample_id AND sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
2	AND sample_filtype_view.file_type_id = 1
	Here we then get the following: **Benign TN = percentage < 0.5 FN = percentage >= 0.5** **Malicious TP = percentage ≥ 0.5 FP = percentage < 0.5 AND percentage >= 0**
3	AND experiment_results.malicious_percentage < 0.5
	Group by packer and count descending
4	GROUP BY virus_total_packer_cryptor_compiler_view.name ORDER BY COUNT(virus_total_packer_cryptor_compiler_view.name) DESC;

Appendix 5: SQL for getting packed samples entropy and packer signature detection

#	**SQL 5. Looking at packed samples**
	Base SQL with optional arguments [2-5] below, here for experiment 7
1	SELECT * FROM sample_filtype_view, virus_total_packer_cryptor_compiler_view, experiment_results WHERE sample_filtype_view.sample_id = virus_total_packer_cryptor_compiler_view.sample_id AND sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
2	AND sample_filtype_view.file_type_id = 2
	Exclude the compiler signatures containing Microsoft and .NET
3	AND virus_total_packer_cryptor_compiler_view.name NOT LIKE '%Microsoft%' AND virus_total_packer_cryptor_compiler_view.name NOT LIKE '%.NET%'
	Get all samples that are predicted packed and are packed
4	AND sample_filtype_view.original_file_entropy >= 6.801 AND sample_filtype_view.original_file_entropy <= 8.0
	Get all the samples that are predicted as not packed
5	AND sample_filtype_view.original_file_entropy >= 0 AND sample_filtype_view.original_file_entropy <= 6.800

Appendix 6: SQL for getting all samples associated with Microsoft compiler signatures

#	**SQL 6. Get all samples associated with Microsoft compiler signatures**
	Base SQL with optional arguments [2-5] below, here for experiment 7
1	SELECT * FROM sample_filtype_view, virus_total_packer_cryptor_compiler_view, experiment_results WHERE sample_filtype_view.sample_id = virus_total_packer_cryptor_compiler_view.sample_id AND sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
2	AND sample_filtype_view.file_type_id = 2
	Get all samples that are predicted packed and are packed
3	AND sample_filtype_view.original_file_entropy >= 6.801 AND sample_filtype_view.original_file_entropy <= 8.0
	Get all the samples that are predicted as not packed
4	AND sample_filtype_view.original_file_entropy >= 0 AND sample_filtype_view.original_file_entropy <= 6.800
5	**Get all the compiler signatures containing Microsoft**
	AND virus_total_packer_cryptor_compiler_view.name LIKE '%Microsoft%'
6	**Get all the compiler signatures containing .NET**
	AND virus_total_packer_cryptor_compiler_view.name LIKE '%.NET%'

Appendix 7: SQL for getting TP, FP, FN, TN for benign and malicious samples

#	**SQL 7. Gets True Positive (TP), False Positive (FP), False negative (FN), and True Negative (TN) for benign and malicious samples**
	Base SQL below, here for experiment 7
	SELECT * FROM sample_filtype_view, experiment_results WHERE sample_filtype_view.sample_id = experiment_results.sample_id AND experiment_results.experiment_id = 7
	Get only benign samples by setting 1 and only malicious samples by setting 2
1	AND sample_filtype_view.file_type_id = 1
	Here we then get the following: **Benign TN = percentage < 0.5 FN = percentage >= 0.5** **Malicious TP = percentage ≥ 0.5 FP = percentage < 0.5 AND percentage >= 0**
2	AND experiment_results.malicious_percentage < 0.5

Appendix 8: Python source code can be found here:

https://github.com/robinntnu/STAMINA

Appendix 9**:** Python code showing how Shannon's entropy formula can be implemented

Python code implementation for calculating Shannon´s entropy of a file

```
def calculate_shannons_entropy(file_path):
  entropy = 0
  # 1. Opens the file in the file_path as read binary and reads it
  # to the variable binary_data
  with open(file_path, 'rb') as binary_file:
    binary_data = binary_file.read()

  # Calculate the file size
  file_size_in_bytes = len(binary_data)

  # 2. Creates a one dimensional array from the buffer, where the
  # return type is unsign 8-bit integers, meaning in range [0-255]
  one_dimensional_pixel_stream = np.frombuffer(binary_data,
                                    dtype=np.unit8)
  # 3. Creating an array with the length 256 to hold the found values
  array = [0] * 256

  # 4. Looping thru the one dimensional_pixel_stream and icrement
  # the array position by one, for every matching value we find
  for i in one_dimensional_pixel_stream:
    array[i] += 1

  # 5. Calculating the entropy according to Shannon´s formula
  for i in array:
    p = i / file_size_in_bytes
    if p > 0:
      entropy += p * log2(p)

  # 6. Returns the entropy
  return -entropy
```

Appendix 10**:** Python code showing how a binary file can be converted to an image [71]

Python code implementation for images conversion

```
def convert_binary_file_to_an_image(file_path, output_path,
                                    image_file_name):
  # Opens the file_path as read binary and reads it to a variable
  with open(file_path, 'rb') as binary_file:
    binary_data = binary_file.read()

  # Creates a one dimentional array with values in range [0-255]
  one_dimensional_pixel_stream = np.frombuffer(binary_data,
                                               dtype=np.uint8)

  # Converting the on dimensional array into a two dimensional
  # Getting the size of the on dimensional array
  one_dimensional_pixel_stream_size = len(one_dimensional_pixel_stream)

  # Calculating the width according to the table above
  if 0 >= one_dimensional_pixel_stream_size and
          one_dimensional_pixel_stream_size <= 10:
    image_width = 32
  if 10 >= one_dimensional_pixel_stream_size and
           one_dimensional_pixel_stream_size <= 30:
    image_width = 64
  if 30 >= one_dimensional_pixel_stream_size and
           one_dimensional_pixel_stream_size <= 60:
    image_width = 128
  if 60 >= one_dimensional_pixel_stream_size and
           one_dimensional_pixel_stream_size <= 100:
    image_width = 256
  if 100 >= one_dimensional_pixel_stream_size and
            one_dimensional_pixel_stream_size <= 200:
    image_width = 384
  if 200 >= one_dimensional_pixel_stream_size and
            one_dimensional_pixel_stream_size <= 1000:
    image_width = 512
  if 1000 >= one_dimensional_pixel_stream_size and
             one_dimensional_pixel_stream_size <= 1500:
    image_width = 1024
  if one_dimensional_pixel_stream_size > 1500:
    image_width = 2048

  # Calculate the image height (height / one_dimensional)
  image_height = int(ceil(one_dimensional_pixel_stream_size /
                          image_width))

  # Calculate the image height (width * height) - one_dimensional
  length_of_extra_pixels_as_zeros = (image_width * image_height) -
                                     one_dimensional_pixel_stream_size

  # Adding the extra zero pixels to the one_dimensional_pixel_stream
  one_dimensional_pixel_stream_with_padding = np.hstack(
  (one_dimensional_pixel_stream,
   np.zeros(length_of_extra_pixels_as_zeros, np.uint8)))

  # From 1D pixel strem to a 2D pixel stream
  two_dimensional_pixel_stream = np.reshape(
      one_dimensional_pixel_stream_with_padding,
      (image_height, image_width))

  # Save to path
```

```
    save_to_path = output_path + '/' + image_file_name + '.png'
    cv2.imwrite(save_to_path, two_dimensional_pixel_stream)

    # Returns where the image is saved
    return save_to_path
```

Appendix 11**:** Python code showing how an image can be resized [69]

Python code implementation for images resizing

```
def resize_image(file_path, output_path, image_name,
                 image_scale_width=299, image_scale_height=299):
  # Reads the given image from the given file_path
  image = tf.io.read_file(file_path)

  # Converts the given image into a tensor, channels=3 is RGB
  image = tf.image.decode_png(image, channels=3)

  # Converts the color channels from [0-255] to [0-1]
  image = tf.image.convert_image_dtype(image, tf.float32)

  # Resize the image to default 299 x 299 or custum size
  image = tf.image.resize(image, size=[image_scale_width,
                          image_scale_height])

  # Path to store the resized image
  save_to_path = output_path + '/' + image_name + '.png'

  # Returns the image from tensors to png
  tf.keras.preprocessing.image.save_img(save_to_path, image)

  # Returns where the resized image is saved
  return save_to_path
```

Appendix 12**:** Python code showing how a MD5 or SHA-256 file hash can be calculated from the given file [72]

Python code implementation for calculating a MD-5 and SHA-256 file hash

```
# Calculates a hash for the given file in file_path and desired hash_out
def calculate_hash(file_path, hash_out):
  # Creates a variable to build / hold the MD5 / SHA-256 hash
  if hash_out == 'MD5':
    hash_builder = hashlib.md5()
  elif hash_out == 'SHA-256'
    hash_builder = hashlib.sha256()

  # Opens the file and reads it in binary mode
  with open(file_path, 'rb') as binary_file:
    # Reads 512 bytes at a time
    for read_bytes in iter(lambda: binary_file_read(512), b""):
      hash_builder.update(read_bytes)

  # Returns the hash in HEX
  return hash_builder.hexdigest()
```

Appendix 13**:** Python code showing a feature extraction model [69], [73]

Python code implementation for a feature extraction model

```
# Input types
model_input_types = [«Benign», «Malicious»]

# Benign and malicious
output_shape = len(model_input_types)

# URL to the pretrained CNN model
feature_extractor_model = “https://tfhub.dev/google/tf2-
preview/inception_v3/feature_vector/4”

# Creates a sequential model with a dense output layer
model = tf.keras.Sequential(
      [hub.KerasLayer(feature_extractor_model, output_shape=[2048],
           trainable=False), # Freezing the convolutional base
      tf.keras.layers.Dense(output_shape), # 2
                           activation=’sigmoid’)]) # Sigmoid for binary

# Builds the model, batch input is none, image width and height is 299
# and color channels are 3 due to RGB (this is the image input shape)
model.build([None, 299, 299, 3])

# Compiling the model
model.compile(optimizer=keras.optimizers.Adam(),
              loss=keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[keras.metrics.BinaryAccuracy()],)
```

Appendix 14**:** Python code showing a fine-tuning model [69], [73], [74]

Python code implementation for a fine tuning model

```
# Input types
model_input_types = [«Benign», «Malicious»]

# Benign and malicious
output_shape = len(model_input_types)

# Sets the base learning rate (low, since we are fine-tunning)
base_learning_rate = 0.01

# Base model
base_model = InceptionV3(input_shape=(299, 299, 3), # Image input shape
                         weights='imagenet', # Weights from ImageNet
                         include_top=False)# Not include classifier

# Looping thru the layers and freezing every layer before number 300
index = 0
for layer in base_model.layers:
  if index == 300: # Last pooling layer
    break
  # Set the layer to not be trainable
  layer.trainable = False
  # Incrementing the index
  index += 1

# Creates a sequential model with a dense output layer
model = tf.keras.Sequential([base_model,
                             tf.keras.layers.GlobalAveragePolling2D(),
                             tf.keras.layers.Dense(units=output_shape,
                             activation='sigmoid')])

# Compiling the model
model.compile(optimizer=keras.optimizers.Adam(base_learning_rate),
              loss=keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[keras.metrics.BinaryAccuracy()],)
```

Table 0.1: Shows how a fine-tuning model can be coded in Python [69], [73], [74]

Appendix 15**:** Python code showing how a simple .exe verification can be coded

Python code implementation for extracting the

```
def is_exe_file(file_path):
  # Read the first 64 bytes
  binary_data = read_number_of_bytes_from_file(file_path, 64)

# If the first bytes are 0x4d or 0x5a or 4D 5A in HEX and MZ in ASCII
# code, the file is EXE
if binary_data[0:1].hex().upper() == '4D' and
   binary_data[0:1].hex().upper()== '5A':
    # Getting the pointer to the PE signature
    e_lfanew = get_e_lfanew(binary_data)
    # Gets the PE signature position in the binary_data
    pe_signature_position = get_pe_signature_position(e_lfanew)
    # Reads the last position of the PE signature + 2 to get the
    # executable type
    binary_data = read_number_of_bytes_from_file(file_path,
                  pe_signature_position[1] + 2)
    # Gets the executable type 32 or 64 bit
    executable_type = executable_is_32_or_64_bit(
                      pe_signature_position, binary_data)
    # The PE Signature is 50450000 is found
    if binary_data[pe_signature_position[0]:
                pe_signature_position[1].hex().upper() == '50450000']:
       # We are only interested in intel x86 and x64 e.g.
       # 64 AA = Arm, hence executable_type can be None
       if executable_type:
          return [True, executable_type]

  return [False, None]
```

Appendix 16**:** Python code showing how x number of bytes can be read from a given file

Python code implementation for reading x bytes from a file

```
def read_number_of_bytes_from_file(file_path, byte_length):
  # Read the first 64 bytes
  with open(file_path, 'rb') as binary_file:
    binary_data = binary_file.read(byte_length)
  # Returns the read data
  return binary_data
```

Appendix 17**:** Python code showing how to read the .exe magic bytes

Python code implementation for extracting the .exe magic bytes

```
# Read the first 64 bytes
binary_data = read_number_of_bytes_from_file(file_path, 64)
# If the first bytes are 0x4d or 0x5a or 4D 5A in HEX and MZ in ASCII
# code, the file is EXE
if binary_data[0:1].hex().upper() == '4D' and
   binary_data[0:1].hex().upper()== '5A':
```

Table 0.2: Shows how getting the .exe magic bytes can be coded in Python

Appendix 18**:** Python code showing how to get the e_lfanew pointer

Python code implementation for getting the e_lfanew pointer

```
def get_e_lfanew(binary_data):
  return f' binary_data[63:64].hex().upper()
            binary_data[62:63].hex().upper()' \
         f' binary_data[61:62].hex().upper()
            binary_data[60:61].hex().upper()'
```

Appendix 19**:** Python code showing how the PE Signature position can be found

Python code implementation for getting the PE Signature position

```
def get_pe_signature_position(e_lfanew):
  # Removes whitespace and converts the hex values to decimal
  start_position = int(hex_to_decimal(e_lfanew.replace(' ', '')))
  end_position = start_position + 4
  return [start_position, end_position]
```

Appendix 20: Python code showing how to convert a HEX value to decimal

Python code implementation for converting a HEX value to decimal
`def hex_to_decimal(hex_value):` `  return int(hex_value, 16)`

Appendix 21: Python code showing how to get the PE Signature

Python code implementation for getting the PE Signature

```
# Gets the PE signature position in the binary_data
pe_signature_position = get_pe_signature_position(e_lfanew)

# Reads the last position of the PE signature + 2 to get the
# executable type
binary_data = read_number_of_bytes_from_file(file_path,
              pe_signature_position[1] + 2)

# The PE Signature is 50450000 is found
if binary_data[pe_signature_position[0]:
             pe_signature_position[1].hex().upper() == '50450000']:
```

Appendix 22: Python code showing how the CPU architecture is extracted

Python code implementation for getting the CPU architecture x86 and x64

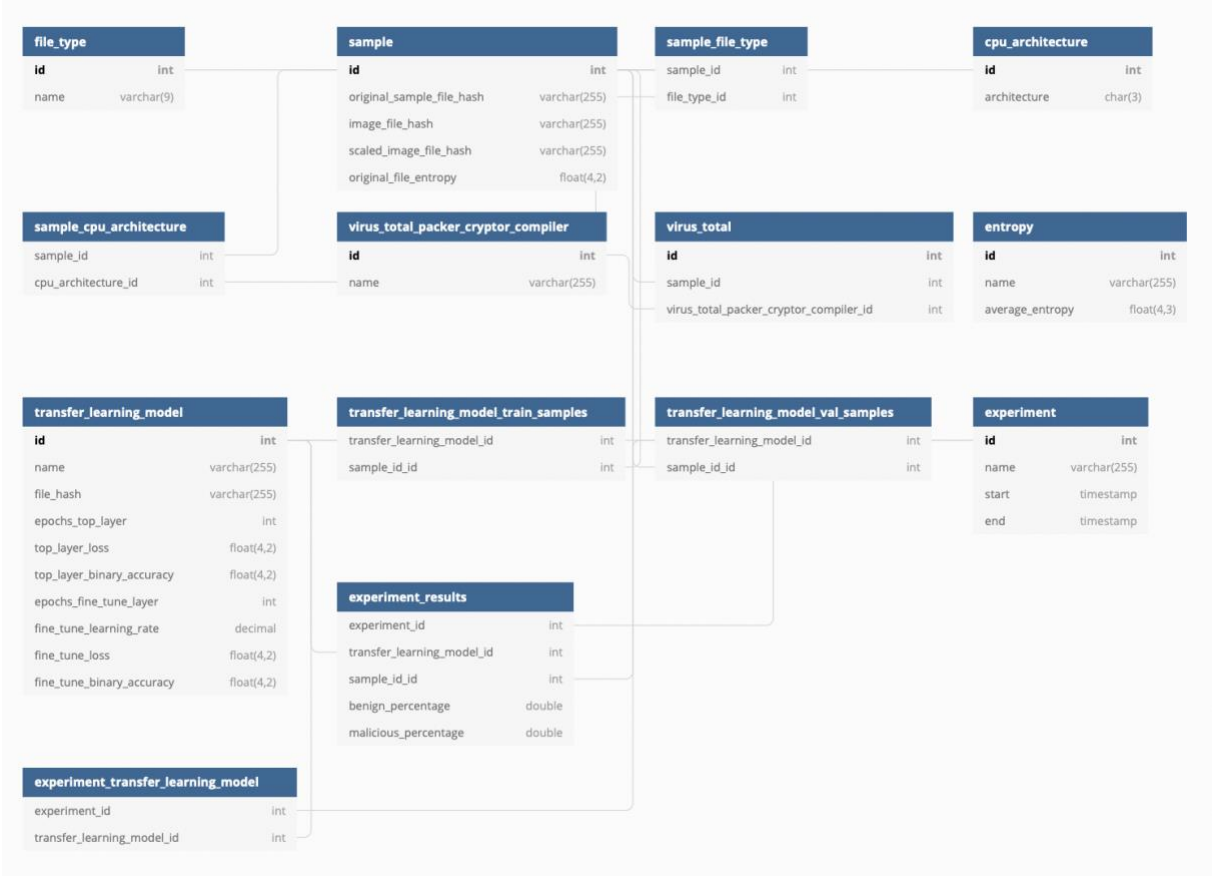
```
def executable_is_32_or_64_bit(pe_signature_position, binary_data):
  # Gets the executable signature
  executable_type_signature = binary_data[pe_signature_position[1]:
    pe_signature_position[1] + 2].hex().upper()
  # x86 or x64
  if executable_type_signature == '4C01': # x86 signature
    return 'x86'
  if executable_type_signature == '6486': # x64 signature
    return 'x64'
```

Appendix 23**:** Python libraries and their version number

Package	Version
absl-py	0.11.0
altgraph	0.17
astunparse	1.6.3
attrs	20.3.0
cachetools	4.2.1
capstone	4.0.2
certifi	2020.12.5
chardet	4.0.0
click	7.1.2
cmd2	0.9.12
colorama	0.4.4
cycler	0.10.0
Flask	1.1.2
flatbuffers	1.12
future	0.18.2
gast	0.3.3
gnureadline	8.0.0
google-auth	1.26.1
google-auth-oauthlib	0.4.2
google-pasta	0.2.0
grpcio	1.32.0
h5py	2.10.0
idna	2.10
itsdangerous	1.1.0
Jinja2	2.11.3
joblib	1.0.1
Keras-Preprocessing	1.1.2
kiwisolver	1.3.1
macholib	1.14
Markdown	3.3.3
MarkupSafe	1.1.1
matplotlib	3.3.4
numpy	1.19.5
oauthlib	3.1.0
opencv-python	4.5.1.48
opt-einsum	3.3.0
pefile	2019.4.18
Pillow	8.1.0
pip	21.1.1
protobuf	3.14.0
py-aho-corasick	1.1.0
pyasn1	0.4.8
pyasn1-modules	0.2.8
pyinstaller	4.3
pyinstaller-hooks-contrib	2021.1
pyparsing	2.4.7
pyperclip	1.8.2
python-dateutil	2.8.1
PyYAML	5.4.1
requests	2.25.1
requests-oauthlib	1.3.0
rsa	4.7
scikit-learn	0.24.1
scipy	1.6.0
setuptools	49.2.1
six	1.15.0
sklearn	0.0

```
tensorboard               2.4.1
tensorboard-plugin-wit    1.8.0
tensorflow                2.4.1
tensorflow-estimator      2.4.0
tensorflow-hub            0.11.0
termcolor                 1.1.0
threadpoolctl             2.1.0
typing-extensions         3.7.4.3
unicorn-unipacker         1.0.3b7
unipacker                 1.0.6
urllib3                   1.26.3
wcwidth                   0.2.5
Werkzeug                  1.0.1
wheel                     0.36.2
wrapt                     1.12.1
yara                      1.7.7
yara-python               4.1.0
```

Appendix 24**:** The Entity-Relationship diagram (ER-diagram) for the SQL result database



The SQL database can be viewed here: https://dbdiagram.io/d by copy pasting in the syntax code below:

```
// File type benign or malicious
Table file_type {
  id int [pk, increment] // auto-increment
  name varchar(9)
}
// Sample benign or malicious
Table sample {
  id int [pk, increment] // auto-increment
  original_sample_file_hash varchar(255)
  image_file_hash varchar(255)
  scaled_image_file_hash varchar(255)
  original_file_entropy float(4,2)
}

// Sample and file type
Table sample_file_type {
  sample_id int [ref: > sample.id] // inline relationship (many-to-one)
  file_type_id int [ref: > file_type.id] // inline relationship (many-to-one)
}

// CPU architecture x86 or x64
Table cpu_architecture {
  id int [pk, increment] // auto-increment
  architecture char(3)
```

```
}

// Sample and its CPU architecture
Table sample_cpu_architecture {
  sample_id int [ref: > sample.id] // inline relationship (many-to-one)
  cpu_architecture_id int [ref: > cpu_architecture.id] // inline relationship (many-to-one)
}
// Virus Total packers, cryptors and compilers
Table virus_total_packer_cryptor_compiler {
  id int [pk, increment] // auto-increment
  name varchar(255)
}
// Sample Virus Total
Table virus_total {
  id int [pk, increment] // auto-increment
  sample_id int [ref: > sample.id] // inline relationship (many-to-one)
  virus_total_packer_cryptor_compiler_id int [ref: >
virus_total_packer_cryptor_compiler.id] // inline relationship (many-to-one)
}

// Entropy Table
Table entropy {
  id int [pk, increment] // auto-increment
  name varchar(255)
  average_entropy float(4,3)
}

// Transfer learning model
Table transfer_learning_model {
  id int [pk, increment] // auto-increment
  name varchar(255)
  file_hash varchar(255)
  epochs_top_layer int
  top_layer_loss float(4,2)
  top_layer_binary_accuracy float(4,2)
  epochs_fine_tune_layer int
  fine_tune_learning_rate decimal
  fine_tune_loss float(4,2)
  fine_tune_binary_accuracy float(4,2)
}

// Transfer learning model and training sample
Table transfer_learning_model_train_samples {
  transfer_learning_model_id int [ref: > transfer_learning_model.id] // inline relationship
(many-to-one)
  sample_id_id int [ref: > sample.id] // inline relationship (many-to-one)
}

// Transfer learning model and validation sample
Table transfer_learning_model_val_samples {
  transfer_learning_model_id int [ref: > transfer_learning_model.id] // inline relationship
(many-to-one)
  sample_id_id int [ref: > sample.id] // inline relationship (many-to-one)
}

// Experiment
Table experiment {
```

```
  id int [pk, increment] // auto-increment
  name varchar(255)
  start timestamp
  end timestamp
}

// Experiment and transfer learning model
Table experiment_transfer_learning_model {
  experiment_id int [ref: > experiment.id] // inline relationship (many-to-one)
  transfer_learning_model_id int [ref: > transfer_learning_model.id] // inline relationship
(many-to-one)
}

// Experiment and results
Table experiment_results {
  experiment_id int [ref: > experiment.id] // inline relationship (many-to-one)
  transfer_learning_model_id int [ref: > transfer_learning_model.id] // inline relationship
(many-to-one)
  sample_id_id int [ref: > sample.id] // inline relationship (many-to-one)
  benign_percentage double
  malicious_percentage double
}
```

References

1. AVTest. Malware: AVTest; [Available from: https://www.av-test.org/en/statistics/malware/.
2. Microsoft. Microsoft Exchange Server Remote Code Execution Vulnerability: Microsoft; 2021 [updated 16.03.2021. Available from: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-26857.
3. Sterud MGaK. Seks hackergrupper utnyttet Microsoft-sårbarhetene før de ble kjent: NRK; 2021 [updated 13.03.2021. Available from: https://nrkbeta.no/2021/03/13/seks-hackergrupper-utnyttet-microsoft-sarbarhetene-for-de-ble-kjent/.
4. VirusTotal. VirusTotal.com [Malware and url scanner]. Available from: https://www.virustotal.com/gui/.
5. Bie T. Nå har det skjedd: Apple Silicon angripes: Iteavisen.no; 2021 [Available from: https://itavisen.no/2021/02/18/na-har-det-skjedd-apple-silicon-angripes/.
6. Intel-Microsoft Collaborated Project Turns Malware into Images. ICT Monitor Worldwide. 2020.
7. Chen L. Deep Transfer Learning for Static Malware Classification. 2018.
8. Dictionary OE. "malware, n.". . (Oxford University Press).
9. Sikorski M, Honig A. Practical malware analysis : the hands-on guide to dissecting malicious software. San Francisco: No Starch Press; 2012.
10. Eric M. Hutchins MJC, Rohan M. Amin, Ph.D. Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. Lockheed Martin Corporation.
11. Hofmann F. Understanding the ELF File Format 2019 [Available from: https://linuxhint.com/understanding_elf_file_format/.
12. magazine m. An In-Depth Look into the Win32 Portable Executable File Format 2002 [Available from: https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail.
13. tutorialspoint. Operating System - Overview [Available from: https://www.tutorialspoint.com/operating_system/os_overview.htm.
14. The VAX/VMS Virtual Memory System [Available from: https://pages.cs.wisc.edu/~remzi/OSTEP/vm-vax.pdf.
15. Chebbi C. Portable Executable format files [Available from: https://www.oreilly.com/library/view/mastering-machine-learning/9781788997409/aaaa9d8c-8722-43cd-a065-6dd850c29d67.xhtml.
16. ASCII [Available from: http://www.asciitable.com.
17. Microsoft. PE Format: Microsoft Windows Developer; [updated 31.03.2021. Available from: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.
18. PEiD [Packer crypto and compiler signature detection tool]. Available from: https://www.aldeid.com/wiki/PEiD.
19. Cloudflare. What is encryption? | Types of encryption [Available from: https://www.cloudflare.com/en-gb/learning/ssl/what-is-encryption/.
20. Naval S, Laxmi V, Gaur M, Vinod P. ESCAPE: entropy score analysis of packed executable. International Conference on Security of Information and Networks: ACM; 2012. p. 197-200.
21. Bat-Erdene M, Kim T, Park H, Lee H. Packer Detection for Multi-Layer Executables Using Entropy Analysis. Entropy (Basel, Switzerland). 2017;19(3):125.

22. Mi-Jung C, Bang J, Kim J, Kim H, Yang-Sae M. All-in-One Framework for Detection, Unpacking, and Verification for Malware Analysis. Security and Communication Networks. 2019;2019:16.
23. Microsoft. /SECTION (Specify Section Attributes [Available from: https://docs.microsoft.com/en-us/cpp/build/reference/section-specify-section-attributes?view=msvc-160.
24. Lyda R, Hamrock J. Using Entropy Analysis to Find Encrypted and Packed Malware. IEEE security & privacy. 2007;5(2):40-5.
25. PEiD Signature Database [Available from: https://raw.githubusercontent.com/guelfoweb/peframe/5beta/peframe/signatures/userdb.txt.
26. Wikipedia. Disassembler [Available from: https://en.wikipedia.org/wiki/Disassembler.
27. Wikipedia. Debugger [Available from: https://en.wikipedia.org/wiki/Debugger.
28. Amazon. Conversational AI: Amazon; [Available from: https://developer.amazon.com/en-US/alexa/alexa-skills-kit/conversational-ai.
29. Wu Y. Smart Compose: Using Neural Networks to Help Write Emails: Google Brain Team; 2018 [Available from: https://ai.googleblog.com/2018/05/smart-compose-using-neural-networks-to.html.
30. Tesla. Autopilot: Tesla; [Available from: https://www.tesla.com/autopilotAI.
31. Education IC. Artificial Intelligence (AI) https://www.ibm.com/cloud/learn/what-is-artificial-intelligence: IBM; 2020 [updated 03.06.2020.
:[
32. Kirsch JHaD. Machine Learning For Dummies®, IBM Limited Edition: John Wiley & Sons, Inc.; 2018. Available from: https://www.ibm.com/downloads/cas/GB8ZMQZ3.
33. Lee KC. Machine Learning 101 — Classification vs. Clustering 2020 [Available from: https://kevin-c-lee26.medium.com/machine-learning-101-classification-vs-clustering-e11b12c71243.
34. Gill JK. Automatic Log Analysis using Deep Learning and AI XENOSTACK2020 [Available from: https://www.xenonstack.com/blog/log-analytics-deep-machine-learning/.
35. Shadforth J. Understanding Backpropagation in Neural Network 2020 [Available from: https://jacqui.sh/understanding-backpropagation-in-neural-networks/.
36. Dommaraju G. Keras' Accuracy Metrics 2020 [Available from: https://towardsdatascience.com/keras-accuracy-metrics-8572eb479ec7.
37. Saxena S. Binary Cross Entropy/Log Loss for Binary Classification 2021 [Available from: https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/.
38. Developers G. Classification: Precision and Recall [Available from: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.
39. Seb. The Sigmoid Function and Binary Logistic Regression [Available from: https://programmathically.com/the-sigmoid-function-and-binary-logistic-regression/.
40. Karakaya M. How to solve Binary Classification Problems in Deep Learning with Tensorflow & Keras? 2020 [Available from: https://medium.com/deep-learning-with-keras/which-activation-loss-functions-part-a-e16f5ad6d82a.
41. Keras. Adam [Available from: https://keras.io/api/optimizers/adam/.
42. Brownlee J. Overfitting and Underfitting With Machine Learning Algorithms 2016 [Available from: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/.
43. Brownlee J. A Gentle Introduction to k-fold Cross-Validation 2018 [updated 03.08.2020. Available from: https://machinelearningmastery.com/k-fold-cross-validation/.
44. sklearn.model_selection.StratifiedKFold: scikit-learn [Available from: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html.

45. Gibert D, Mateu C, Planes J, Vicens R. Using convolutional neural networks for classification of malware represented as images. Journal of Computer Virology and Hacking Techniques. 2019;15(1):15-28.
46. Rohrer D. How to Convert an RGB Image to Grayscale.
47. Google. imagenet/inception_v1/classification [Available from: https://tfhub.dev/google/imagenet/inception_v1/classification/5.
48. Wikipedia. Nearest neighbour algorithm [Available from: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.
49. Wikipedia. Bilinear interpolation [Available from: https://en.wikipedia.org/wiki/Bilinear_interpolation.
50. Tabora V. JPEG Image Scaling Algorithms: Medium.com; 2019 [Available from: https://medium.com/hd-pro/jpeg-image-scaling-algorithms-913987c9d588.
51. Uniqtech. Understand the Softmax Function in Minutes 2018 [Available from: https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d.
52. Nehemiah A. Deep Learning Tutorial Series 2017 [Available from: https://blogs.mathworks.com/pick/2017/06/02/deep-learning-tutorial-series/.
53. Regua HD. Introducing Transfer Learning as Your Next Engine to Drive Future Innovations 2020 [Available from: https://medium.datadriveninvestor.com/introducing-transfer-learning-as-your-next-engine-to-drive-future-innovations-5e81a15bb567.
54. Banin S, Dyrkolbotn GO. Detection of Running Malware Before it Becomes Malicious. 2020.
55. Google. imagenet/inception_v3/classification [10.04.2021]. Available from: https://tfhub.dev/google/imagenet/inception_v3/classification/4
56. TensorFlow. tf.image.resize [cited 2021 11.04.2021]. Available from: https://www.tensorflow.org/api_docs/python/tf/image/resize
57. TensorFlow [Available from: https://www.tensorflow.org.
58. NortonLifeLock [Available from: https://www.nortonlifelock.com/us/en/.
59. Introduction to JSON [Available from: https://www.json.org/json-en.html.
60. VirusTotal. VirusTotal API v3 Overview [Available from: https://developers.virustotal.com/v3.0/reference#public-vs-premium-api.
61. UPX. UPX the Ultimate Packer for eXecutables [executable packer for executable files]. Available from: https://upx.github.io.
62. Homebrew. Homebrew Package Manager for MacOS or Linux [Available from: https://brew.sh.
63. UPX Homebrew terminal command [Available from: https://formulae.brew.sh/formula/upx.
64. TensorFlow. tf.io.decode_png [Method used for converting a grayscale image to RGB]. Available from: https://www.tensorflow.org/api_docs/python/tf/io/decode_png.
65. Keras [Available from: https://keras.io/.
66. Keras Applications [Available from: https://keras.io/api/applications/.
67. Bensaoud A, Abudawaood N, Kalita J. Classifying Malware Images with Convolutional Neural Network Models. 2020.
68. Chen B. A Practical Introduction to Keras Callbacks in TensorFlow 2 2020 [Available from: https://towardsdatascience.com/a-practical-introduction-to-keras-callbacks-in-tensorflow-2-705d0c584966.
69. Sarang P. Artificial neural networks with TensorFlow 2 : ANN architecture machine learning projects. Place of publication not identified: Apress; 2021.
70. TensorFlow. TensorFlowHub [Available from: https://www.tensorflow.org/hub.
71. Rotem. convert file into grayscale image 2021 [Available from: https://stackoverflow.com/questions/60193896/convert-file-into-grayscale-image.
72. quantumSoup. Generating an MD5 checksum of a file 2010 [Available from: https://stackoverflow.com/questions/3431825/generating-an-md5-checksum-of-a-file.
73. TensorFlow. tf2-preview/inception_v3/feature_vector [Available from: https://tfhub.dev/google/tf2-preview/inception_v3/feature_vector/4.
74. fchollet. Transfer learning & fine-tuning: Keras; 2020 [Available from: https://keras.io/guides/transfer_learning/.