Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Alexander Daniel Forfot

Exploring the PE header and the Rich header for effective Malware Classification and Triage

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2021

NTNU
Norwegian University of
Science and Technology

Alexander Daniel

Exploring the PE header and the Rich header for effective Malware Classification and Triage

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Abstract

The use of executables to introduce and embed malware within systems has been widely used by malicious actors since the internet was introduced. This progression has only increased in parallel with the technological dependency within today's society. Given the massive amounts of malicious files detected each day security analysts are simply not able to process and review all of these manually, although it could be theoretically possible given an infinite amount of human resources. Such a unrealistic requirement is however not feasible to achieve. Because of this challenge, focus on classification and triage of malware is very important. Automated classification allows analysts to only focus on files that actually are malicious, and a triage process aims to filter out files that are less important. Less important does not mean it is not malware, but it indicates that it is a type of malware that has already been analyzed or is very similar to files that have been analyzed. To further simplify the process of automated classification and triage we have explored the possibilities of using the PE headers and rich headers of PE files. While existing research into the headers have shown promising results separately, significantly less effort has been put into combining them. Through a combination of features from the headers we have been able to present one type of technique in which they can be used to classify malware and be used within a triage process. This has been done with the intention of reducing the workload of unnecessary analysis for each security analyst.

Sammendrag

Bruken av kjørbare filer for å introdusere og legge til skadelig programvare i systemer har vært mye brukt av ondsinnede aktører siden internett først ble introdusert. Denne progresjonen har bare økt parallelt med den teknologiske avhengigheten i dagens samfunn. Gitt de enorme mengdene skadelige filer som oppdages hver dag er sikkerhetsanalytikere rett og slett ikke i stand til å behandle og analysere alle disse manuelt, selv om det teoretisk sett kan være mulig med en uendelig mengde menneskelige ressurser. Et slikt urealistisk krav er imidlertid ikke oppnåelig. På grunn av denne utfordringen er fokus på klassifisering og triage av skadelig programvare veldig viktig. Automatisert klassifisering tillater analytikere å kun fokusere på filer som faktisk er skadelige, og en triage prosess har som mål å filtrere ut filer som er mindre viktige. Mindre viktig betyr ikke at det ikke er skadelig programvare, men det indikerer at det er en type skadelig programvare som allerede er analysert eller er veldig lik andre filer. For å forenkle prosessen med automatisert klassifisering og triage ytterligere, har vi utforsket mulighetene for å bruke "PE headeren" og "rich headeren" til PE filer. Eksisterende forskning har vist lovende resultater for disse headerene hver for seg, men vi har fokusert på å utforske kombinasjonen av dem. Gjennom en kombinasjon av metadata fra headerene har vi vært i stand til å presentere en type teknikk der de kan brukes til å klassifisere skadelig programvare og brukes i en triage prosess. Dette er gjort med den hensikt å redusere arbeidsbelastningen for unødvendig analyse for enhver sikkerhetsanalytiker.

Acknowledgements

First and foremost I would like to thank my supervisor, Geir Olav Dyrkolbotn, for his guidance throughout the semester and his valuable focus on continued progress.

I would also like to thank my co-supervisor from Norton LifeLock, Trygve Brox, for his valuable input and the knowledge he provided through his expertise within the field.

A special thanks also goes to Robin Berg. He was a great sparring partner to bounce ideas and concerns with throughout the entire process.

Lastly I would like to thank my friends and family for the continued support, as the thesis would not have been possible without them.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

This chapter provides a introduction to the topics that are covered in the thesis. A brief overview of the problem and our motivation behind the research project. Further we provide a outline of our research questions and contributions.

1.1 Topics covered

Our thesis is based on the topic of malware analysis. *Malware* is a type of software that has been created with the intention of performing harmful activities on computers, this could include anything from extracting sensitive data to physical harm of components, systems or people, a well known example of the latter from modern times includes Stuxnet [1]. With the continued growth of the internet and our dependency on technology the possibilities with malware has also expanded. Newer and more advanced malware is constantly in development, as seen by the recent campaigns targeting vulnerabilities in Microsoft Exchange [2] and the SolarWinds incident [3], and deployed by a vast range of threat actors. A threat actor can be defined as follows: "*a threat actor is a group or person behind a malicious incident*" [4].
While new and advanced malware is a challenge by itself, the biggest challenge can be considered the number of malicious samples [5]. Malware is often re-used with minor modifications to try and remain undetected, as this would reduce the cost and effort of developing new malware. Code that targets known vulnerabilities may also become publicly available or be shared among threat actors. This can be used to target potential victims that have not patched their systems. The most important thing to note is that every single malicious file that is used can be considered a sample. This includes both new and old malware, and all the different variations these may come in. A process of prioritizing these samples is commonly called a triage process. Such a process aims to remove samples that are considered insignificant. This could include samples that have already been analyzed previously or are very similar to further reduce the need for manual analysis.
An important distinction throughout our research is what we consider to be samples.

Samples can generally be any type of software that has a malicious purpose, this includes the *.exe, *.cmd, *.vbs, *.js, *.pdf file types to name a few [6]. We have specifically chosen to focus on Portable Executable files, also referred to as windows executable files, *.exe files or PE files, and as such when we refer to samples we refer to these kinds of files.
Various methods exist using statistics and machine learning to help automate the process of triage. Our approach has utilized anomaly based detection and classification in a combination with frequency analysis. Through the use of these techniques we have attempted to classify samples as well as conduct a triage to identify samples that are of further interest for manual analysis. Anomaly detection and classification is a binary classification technique[1], where an x amount of training data is used to represent normal and where the input data is classed as either normal or not normal based on an automated analysis. To be able to compare our samples we extracted attributes from the headers of the PE files and used a frequency analysis on the training dataset to establish what was to be considered normal.

1.2 Keywords

Malware analysis; PE header; Rich header; Classification; Triage

1.3 Problem description

Anti-virus vendors have throughout the decades been considered as the good actors that have developed methods to detect malicious files. These vendors have specialized in the detection and prevention of harmful activities caused by malware. While this is still their role today, organizations dedicated to information security have also become more common in modern times. Organizations such as Norton LifeLock, with a expansive portfolio of security services, are becoming necessary to handle analysis of all the different types of malicious samples that appear. One of the main abilities of such organizations is the capability to perform extensive malware analysis. This is both due to their experience within the domain as well as the resources and capabilities they possess. A larger, specialized, organization will have better capabilities compared to a single person in terms of analysis.
We believe that the main problem with malware analysis is the significant number of samples that are received. These samples are usually collected through tools and software installed on computers or servers, or even dedicated hardware installed in networks to detect and collect files that could potentially be malicious. These techniques will lead to a very large amount of samples being gathered since a lot of files are transferred within- and to organizations on a day-to-day basis. Another method of collection is through manual submissions from organizations

[1]A classification technique where there are only two possible outcomes.

and people. Organizations and people often have an interest in verifying whether software is malicious or not, and submitting it to a anti-virus vendor enables them to do this. This simultaneously leads to the collection of additional samples.
Sufficient analysis of each sample without investing large amounts of organizational resources, such as hardware for computations or payment for a large number of employed analysts. The process of finding the necessary number of analysts can be considered a problem in itself since an endless supply does not exist. The need for resources also vary over time; it is difficult to pinpoint the workload on a day-to-day basis. This affects the demand for both computational power and analysts. These points in combination with the lack of competence and the time consuming hiring processes makes it a difficult process even for large international organizations. The large amount of samples has resulted in an increased use of machine learning techniques to reduce the number of samples that need manual analysis, as highlighed by known vendors such as Bitdefender [7], Avast [8] and Comodo Cybersecurity [9].

1.4 Justification, motivation and benefits

Analysis of malware is very important because it helps researchers and organizations gain further insight into the functionality and purpose of malicious software. This insight aids in the protection of organizations; potential victims, from threat actors that utilize malware. The process of analysis is often time consuming since the number of samples that require analysis is large and require resources. This includes both computational power and human resources for manual analysis. The work related to malware analysis is a very important aspect of information security. Information security is defined by SANS as
"*the processes and methodologies which are designed and implemented to protect print, electronic, or any other form of confidential, private and sensitive information or data from unauthorized access, use, misuse, disclosure, destruction, modification, or disruption.*"
The ability to implement processes and methods to protect sensitive information or data would be very difficult without the process of malware analysis. Without the insights that are gained one would not be able to determine how threat actors attempt to target victims and the methods they use in their attacks. This means that malware analysis is a fundamental aspect of defining what the focus within information security needs to be. Malicious actors often use different techniques to try and infiltrate a system or a specific target and as such the need to analyze malware will also be a continuous process. The techniques that are used by threat actors will, if successful, result in unwanted and malicious software being placed within the targeted system. Effective and automated methods for analysis of such files are a necessity.
This necessity becomes especially clear then reviewing the number of malware samples that needs analysis. Norton LifeLock alone receive up to 700 000 new malware specimen every day. This gives an analyst 0.1 seconds if he or she were to

look at each sample. Regardless of experience and expertise this is not a sufficient amount of time to conduct thorough analysis by one analyst. Valuable time will be wasted on insignificant samples and more sophisticated samples in need of in-depth analysis will not be able to receive an appropriate analysis. Roughly 1000 analysts would be required to perform sufficient analysis on each of the 700 000 samples. A requirement that is very hard to fulfill, not scalable and is an unrealistic use of organizational resources. This research will focus on ways to help reduce the time wasted on insignificant samples; effectively reducing the number of samples to only the ones that require manual analysis.
A reduction in the number of samples is commonly associated with a triage process. The word triage stems from the french word "trier" and was originally used to describe a process of sorting [10]. It was, and is also today, often referred to within the medical profession as a method of prioritization. Originally it primarily focused on situations which involved mass casualties, separating casualties into groups based on the need for treatment as resources were often limited.
Malware triage builds on the same principles, but applies them to the analysis process of malicious samples. It is there to help make the analyst more efficient as well as reducing the time spent on unnecessary or known samples. By reducing the total number of samples that needs to be analyzed by a given analyst one is able to achieve this. The process of reducing the number of samples and prioritizing them is considered triage. Malware analysts can be considered a limited resource, as a company is not able to maintain an endless supply, nor does an endless supply even exist. This means that triage is a very important aspect of modern malware analysis, as the total number of samples that can be observed on a given day varies as well as being in the millions.

1.5 Scope

We have found it necessary to narrow the scope of research to be able to produce results within a limited timeframe. Malware can come in many shapes and forms, but statistics show that the majority of files that are submitted to anti-virus vendors and VirusTotal are PE files [11]. This lead us to chose PE files as our area of focus. We gain additional insights into these files through a combination of features from the PE header and rich header, as this is a area that has not been extensively researched previously. We do not go further into problems surrounding packed and unpacked files, or extensively modify our technique for high efficiency, as this would have been time consuming and further affected our ability to obtain sufficient results in a negative manor.

1.6 Research questions

The research questions in this thesis is defined as follows:

- What is the current state of the art for malware analysis using the PE header and/or the rich header?
- Will classification based upon features from both the PE header and rich header give better results compared to the headers separately?
- How can a combination of features from the PE header and rich header lead to a more effective triage of samples?

1.7 Contributions

Previous research has focused on malware classification and triage through the use of the PE header and some has been conducted with the rich header, i.e. looking at the two headers separately. The main contribution that we provide is the unique approach using anomaly detection and frequency analysis to conduct classification and triage. We present prominent results using both the header types, but especially when considering triage.

1.8 Thesis outline

This thesis is divided into a total of six chapters: starting with the introduction, chapter 2 presents an overview of the relevant theoretical material and existing research into the domain. Chapter 3 describes the methodology that was used and our two datasets. The results that were obtained is presented in chapter 4, and these are discussed in more detail in chapter 5. Chapter 6 consists of our conclusion and proposals for future work.

Chapter 2

Background

This chapter aims to provide more detail about malware and some of the relevant related subtopics, anomaly-based detection and classification, frequency analysis, and the PE file format. We will conclude the chapter by providing related articles that cover classification and triage through the use of the PE header and rich header. The articles that are present have been published recently and are considered as state of the art.

2.1 Malware

Software is considered as malware when it has the ability to cause harm or purposely impact a organization, network, computer or user in a negative manner. This type of software plays a large part in computer attacks and intrusions today, as seen in the recent campaigns targeting vulnerabilities in Microsoft Exchange servers [2] and the SolarWinds breach [3]. Malware can come in all shapes and forms, commonly known types include: adware, backdoors, spyware, trojans, rootkits, viruses, worms, etc. [12].

2.1.1 Analysis

Analysis of malicious software is called *malware analysis*. Sikorski and Honig describes it further as "*[...] the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it.*"[12]. Malware analysis can be divided into two main subcategories:

- "*Static analysis is the testing and evaluation of an application by examining the code without executing the application* " [13].
- "*Dynamic analysis is the testing and evaluation of an application during runtime*" [13].

In malware analysis, static analysis will focus on reviewing the source code and metadata to try and determine the functionality and purpose of the software. This is a process that may provide useful insights while being easy to automate. It is

simultaneously prone to various obfuscation techniques. The use of obfuscation makes it harder for analysts and automated processes to appropriately analyze software. A bit more detail regarding this is presented in section 2.1.3.
Dynamic malware analysis is the process of executing; running the software and observing its behavior. These observations reveal functionality that may not have been easily identifiable through the static analysis, such as the software unpacking itself during runtime or spawning new processes to hide certain functionality. Methods of obfuscation can be easier to detect based on the analysis of runtime behavior. Actual behavior is harder to detect through static analysis in comparison. Dynamic analysis is however a process that is considered harder to automate, compared to static analysis, and require more processing power as software is being analyzed line by line. However, automation of this kind of analysis has been a focus area for anti-virus vendors to try and handle the massive number of samples they are met with [14].

2.1.2 Classification

Classification is a task that is commonly associated with machine learning. Fundamentally it is where a object is described through a given amount of features. These features represent the object. Together the features will enable the object to be attributed to a overall class that could be further used to describe a group of similar objects [15]. This is further clarified through an example:
An object *o* is represented by the features *a, b, c*. These features become a way of representing *o* and can be used to compare the object with other objects. The object *p* is represented by the features *a, d, e*. These two objects have the common feature of *a*, but have no other features in common. If these objects were to be classified into groups and all features are equally weighted, *o* and *p* would not be in the same group. This is because having 1 out of 3 features in common would not be similar enough. If the features were not equally weighted, and feature *a* was considered more important, objects *o* and *p* could be classified into the same group.
In relation to malware, classification acts as a method to determine whether the given software is in fact malicious or not and it can also be used to determine which type of malware a sample actually represents [12]. This is a process that can be done through static analysis, analysis without executing the code, and dynamic analysis, analysis by executing the code. Analysis is something that can be conducted by experts that have specialized knowledge within the domain and are subsequently able to classify the software through manual analysis, or by the use of machine learning techniques to enable an increased amount of automation in the process.

2.1.3 Obfuscation

Malware authors often use specific techniques to obfuscate their malware and make it more difficult for analysts to detect and analyze. A large variety of techniques exist.

Packers

A subset of obfuscated malicious programs are packed programs [12]. This is where a malicious program is compressed. Figure 2.1 illustrates the difference between a unpacked and packed executable. Packers will commonly pack the contents of the executable and create a unpacking stub. This stub contains the necessary information to initiate the unpacking and it is where the entry point is moved to when a file is packed. The header will remain the same, however some of the contained information will be modified as some metadata related to the executable will have changed due to the packing e.g. file size, number of imports, section sizes, etc. This means that it is still possible to conduct some analysis on packed executables, such as packer detection. However, this approach is not perfect and the use of more uncommon methods, as well as packing in multiple layers, still makes it a very prominent method of obfuscation - especially considering how available and easy to use packers are. A program will often need to be unpacked to reveal its functionality completely.

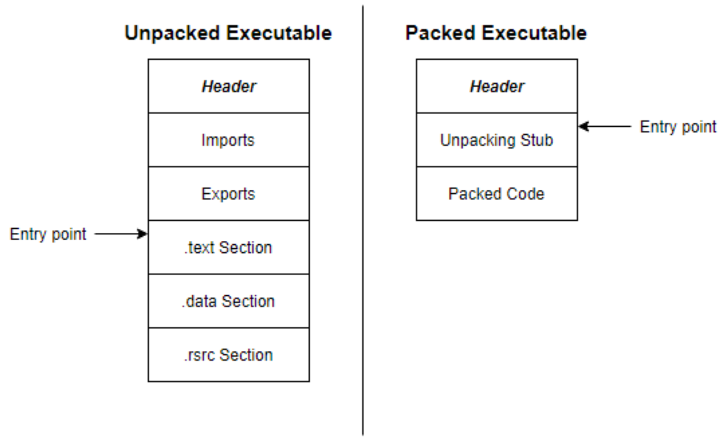


Figure 2.1: Structure of a unpacked and a packed executable [12].

Encryption

Another method of obfuscation that malware can use is encryption. One or multiple methods of encryption can be used to encrypt software. This makes it difficult to conduct thorough analysis and encryption will modify the metadata related to software until a specific key is submitted. It is also a very simple method that can be used to change the signature; hash[1] of malware since encryption is based on a input key [17]. Very minor modifications will lead to different signatures [12]. Changing just a single bit in the metadata will lead to a new signature. This is further illustrated in the example provided in table 2.1 However, while multiple input keys can be used the method of decryption will often remain the same.

Table 2.1: Exact hashes of two very similar identical files. File 1 is a *.txt file with the string "This is a test." and file 2 is a *.txt file with the string "This is a test".

MD5	1b172ccdeb2f51452b5c56351c6cbba6
	61fa840406674ddb0aafd4fceea78420
SHA256	3b7dd38c649a6e0fd98cf21c3ae22be1124024829857e60ae47cf8498c426aac
	026ce0c5cc4f6785bf0893c44d2276993176c100e18943cbc1770ac124eaf509

As encryption is a form of modification of the software and its contents it will also lead to a modification of certain parts of the file header [18].

2.2 Anomaly-based detection and classification

Anomaly-based detection is a concept that is more thoroughly used and explored within intrusion detection systems [19]. Such a technique is commonly used when one looks for traffic that can be considered abnormal; out of the ordinary. E.g. detecting an attempt to access an internal server from a completely unknown IP address. This would usually lead to warnings, many of which are false positives[2] as there is no specific way to determine what is to be considered normal traffic.
Anomaly-based classification is a very similar approach, but it focuses on classification instead. It is commonly when one looks at feature representations that are abnormal. E.g. detecting that a unknown executable file contains header features that are not before seen within known benign files. This will commonly lead to the unknown file being classified as potentially malicious and further flagged for analysis. In particular cases where features are completely unknown a thorough manual analysis would be necessary to determine the purpose and functionality. However, in a lot of cases some features can be attributed to known malware and can more easily be classified as malicious based on its deviation from the known benign features. A lot of executables also share common features even though their intentions differ: benign or malicious. This leaves room for false positives, where benign files are classified as malicious. However, false negatives[3]; malicious files classified as benign, are worse and one should focus on a reduction of them.

[1]A file hash is a unique identifier used to identify data, often a message or a file [16].

[2]A false positive is described by Merriam Webster as "*a result that shows something is present when it really is not*" [20]; a misclassification.

[3]A false negative is a result that shows nothing is present while there is actually something there.

2.3 Frequency analysis

Frequency analysis is a sub-field within descriptive statistics [21]. It is used as a method to gain more insight into the data that is to be analyzed. Frequency is commonly in statistics known as the number of occurrences of an event or a thing. While there are a number of measures that are used within frequency analysis, namely: measures of central tendency dispersion and percentile values [21]. Our researched focused on a combination of the central tendency and dispersion. The central tendency is a measure that is used to describe the data through a value considered the central position; middle. Commonly used measures are mean, mode and median. Dispersion is a measure that focuses on the spread or variability within the data.
We used the central tendency, and the mode, to see what was considered "normal" among features from a training dataset. Comparison of feature values with the central tendency of the training dataset and a given sample in the testing dataset is what we used to determine how a given sample would be classified.

2.4 PE file format

PE stands for "Portable Executable" and it is a file format that is used within the Windows operating system [12]. Its purpose is to provide the structure that the operating system loader needs to manage the contained code within the executable files[4]. The file format is used by: "*executables, object code and DLLs*" [12].

2.4.1 PE header

The PE file header consists of three main headers and two main tables. It starts with the **DOS Header** which contains the string "MZ", the DOS stub and the file offset to the PE signature. This file offset is always located at 0x3c and it enables Windows to properly execute the file. The file signature is the letters P and E followed by two null bytes. Following the PE signature is the **COFF File Header**. This contains information about the type of machine the file is intended for and some basic flags that provide further information about the file. Then comes the **Optional Header**. It is an optional header because it is not present in all files, specifically: object files[5], but it is required by image files[6]. The optional header contains extended fields that aim to provide further information about the executable. It has three main sections, standard fields: has general information that is useful when loading and executing the file, windows-specific fields: an extension to the COFF optional header that contains information required by the loader and

[4]An executable file can be described as a type of file that aims to provide specific functionality based on instructions provided as code.

[5]An object file is a type of file that is provided as input to the linker. The linker will use this input to create an image file [22].

[6]A image file is a executable file, being either a .EXE file or a DLL file [22].

linker in Windows, and **Data Directory Table**: pointers and sizes for specific segments of the files' data. Lastly there is the **Section Table**, consisting of multiple rows, also called section headers, that represent various sections of the executable. These sections combined contain the program itself, e.g. the instructions. A more detailed overview of the contents of the different parts is displayed in table 2.2, based on the documentation provided by Microsoft [18]. An even more detailed overview off functionality can be found in the original documentation available in [18].

Table 2.2: PE Header contents [18].

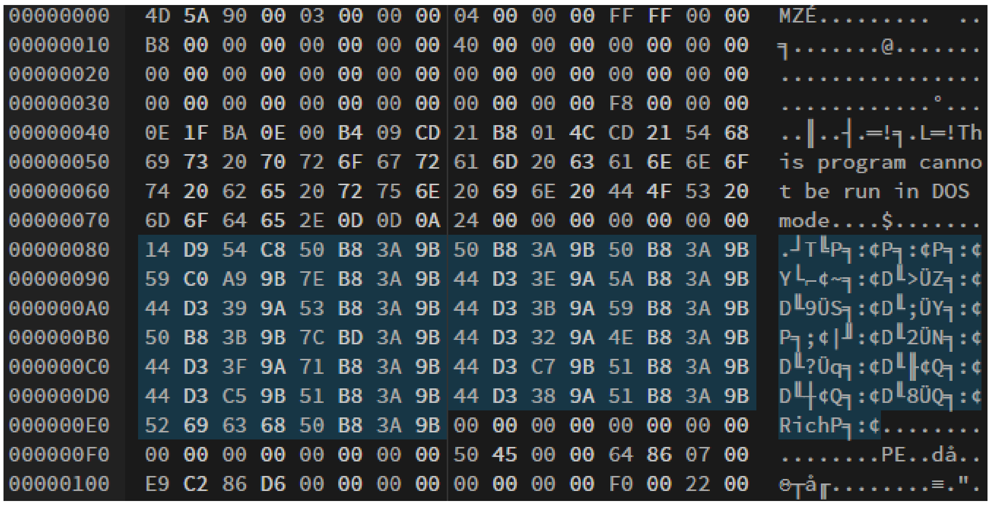
DOS Header	
DOS Stub	PE signature offset
COFF File Header	
Machine	NumberOfSections
TimeDateStamp	PointerToSymbolTable
NumberOfSymbols	SizeOfOptionalHeader
Characteristics	
Optional Header - Standard Fields	
Magic	MajorLinkerVersion
MinorLinkerVersion	SizeOfCode
SizeOfInitializedData	SizeOfUninitializedData
AddressOfEntryPoint	BaseOfCode
BaseOfData	
Optional Header - Windows-Specific Fields	
ImageBase	SectionAlignment
FileAlignment	MajorOperatingSystemVersion
MinorOperatingSystemVersion	MajorImageVersion
MinorImageVersion	MajorSubsystemVersion
MinorSubsystemVersion	Win32VersionValue
SizeOfImage	SizeOfHeaders
CheckSum	Subsystem
DllCharacteristics	SizeOfStackReserve
SizeOfStackCommit	SizeOfHeapReserve
SizeOfHeapCommit	LoaderFlags
NumberOfRvaAndSizes	
Optional Header - Data Directory Table	
Export Table	Import Table
Resource Table	Exception Table
Certificate Table	Base Relocation Table
Debug	Architecture
Global Ptr	TLS Table
Load Config Table	Bound Import
IAT	Delay Import Descriptor
CLR Runtime Header	Reserved, must be zero
Section Table	
Name	VirtualSize
VirtualAddress	SizeOfRawData
PointerToRawData	PointerToRelocations
PointerToLinenumbers	NumberOfRelocations
NumberOfLinenumbers	Characteristics

Manipulated PE header

Manipulation of the PE header is something that is relatively easy to accomplish [23]. The header builds on metadata related to the file and its contents. This means that a simple action such as modifying the code will indirectly impact the header and act as a form of obfuscation e.g. adding additional code that is not actually used by the program. Additionally, some of the header data can have a range of values and some are even arbitrary [18]. Arbitrary, in the sense that the Windows loader does not read certain sections. This means that the header is able to be manipulated without necessarily changing the way the program is interpreted.

2.4.2 Rich header

Documentation for the PE header is well established and known among developers and analysts. However, within the PE header there is a undocumented structure that is only present within Microsoft-produced executables: The rich header [24]. An example of how the header looks like is provided in Figure 2.2. The image also illustrates a few key features of the header: its location and footer. The rich header is embedded within the DOS stub and resides between the "This program cannot be run in DOS mode" string and the PE signature (at 0xF0). The footer will always contain the string "Rich" and it makes it easy to identify the presence or absence of the rich header.



```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZÉ.........  ..
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 F8 00 00 00  ............°...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..║..┤.═!╕.L═!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  14 D9 54 C8 50 B8 3A 9B 50 B8 3A 9B 50 B8 3A 9B  .┘T╚P╕:¢P╕:¢P╕:¢
00000090  59 C0 A9 9B 7E B8 3A 9B 44 D3 3E 9A 5A B8 3A 9B  Y└⌐¢~╕:¢D╙>ÜZ╕:¢
000000A0  44 D3 39 9A 53 B8 3A 9B 44 D3 3B 9A 59 B8 3A 9B  D╙9ÜS╕:¢D╙;ÜY╕:¢
000000B0  50 B8 3B 9B 7C BD 3A 9B 44 D3 32 9A 4E B8 3A 9B  P╕;¢|╜:¢D╙2ÜN╕:¢
000000C0  44 D3 3F 9A 71 B8 3A 9B 44 D3 C7 9B 51 B8 3A 9B  D╙?Üq╕:¢D╙╟¢Q╕:¢
000000D0  44 D3 C5 9B 51 B8 3A 9B 44 D3 38 9A 51 B8 3A 9B  D╙┼¢Q╕:¢D╙8ÜQ╕:¢
000000E0  52 69 63 68 50 B8 3A 9B 00 00 00 00 00 00 00 00  RichP╕:¢........
000000F0  00 00 00 00 00 00 00 00 50 45 00 00 64 86 07 00  ........PE..då..
00000100  E9 C2 86 D6 00 00 00 00 00 00 00 00 F0 00 22 00  Θ┬å╓........≡.".
```

Figure 2.2: Hex editor view of notepad.exe displaying the rich header, highlighted in blue.

The rich header appears to have been introduced in 1997 with the final Service Pack for Visual Studio 5.0 with the first "Rich" capable linker. While first being introduced with this linker, it only had the capability of creating empty data structures. This was due to compilers not yet being able to emit the "@comp.id" sym-

bol[7]. With the release of Visual Studio 6.0 in 1998 this changed and the rich header started to contain data.
Analysis done by stephen[8] in [25] provides a overview of the contents of the rich header. It starts with a marker, followed by a checksum (x3), encoded values, the string "Rich" and then ends with the same checksum. Further insight reveals that the marker is the value 0x536e6144 (DanS in ascii - an identifier similar to MZ, PE and Rich) and it is XORed with the calculated checksum value [24–26]. The format of the encoded values is @comp.id XORed with the checksum and the number of occurrences of the corresponding @comp.id XORed with the checksum.
A large part of the rich header relies on the calculated checksum, one way it has been described is as follows: "*[...] computed by iterating over every byte of the DOS Header, skipping the elfanew field, copying the byte into a 32-bit field, rotating the field left by the field's offset in the PE header, and then adding that to a sum. But that is not all. Next, that sum is then added to each compid XORed with its occurrence count. That is, the 32-bit sum is added to the value resulting from compid XOR occurrence, for every compid in the list.*" [25].

Manipulated Rich header

Direct manipulation of the rich header is also something that could occur. While research shows that anomalies in rich headers are a good feature for malware detection, as it renders the checksum invalid and this is something which does not occur for benign executables. However, it becomes significantly more difficult when the header is modified in such a way that it tries to mimic other "known" rich headers. This first appeared in the attacks targeting the Pyeongcheng Winter Olympics in 2018 [27]. The malicious actors had used several techniques to make their malware samples look similar to samples that had previously been attributed to the Lazarus group, a APT (Advanced Persistant Threat) with strong links to North Korea [28]. One of the techniques that were used was manipulation of the rich header.
This article showcases the potential to use the rich header to mislead investigators. However, it does not seem to be a very established technique. This is the first time it has been detected in the wild. There is no way of knowing how many times it may actually have been used. While not being very widespread as of now, it could be a technique that becomes more viable and used in the future.

2.5 Related work

This chapter aims to showcase some of the related research that has been conducted into both the PE header and rich header within malware detection and triage. We believe that these articles present the state of the art solutions based on their time of publication and significance. We deem significance as the number

[7]A attribute that represents the "compiler build number" and "id" [24].
[8]A username used by the author. We were unable to find the authors full name.

of citations and downloads, in addition to briefly reviewing other works by the author. These factors combined makes us confident in our ability to present state of the work arts related to our research.
We would like to highlight one key takeaway from these articles: they showcase the potential with these headers separately, but they do not look at both the headers simultaneously.

2.5.1 Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage [26]

Webster et. al has provided further research into the rich header and the potential it has within malware triage [26]. They present an in-depth analysis of the undocumented rich header and attempt to highlight how it is viable when attempting to conduct triage.
Through the analysis the authors have been able to present the usefulness of the rich header to: cheaply identify packed malware, similarity matching and identification of malware that has been developed using similar build environments [26]. What further makes this useful is the fact that 71% of the 964 816 samples that were analyzed included the rich header. Their researched showed that even samples that had manipulated PE32 headers or PE32 headers with little information still maintained useful information for analysts and a source of data for rapid triage within the rich header. This indicates that the rich header was a largely neglected part of the PE32 file format by malware authors at the time the research was published.
These are all aspects that make the rich header a very interesting resource when conducting malware triage. It adds another feature that can be analyzed and this can further enable a broader approach towards the triage. Analysis of the rich header by itself is not as interesting, as this is presented in the research paper [26], but rather a combination of the rich header with other sources of data is of interest for further research. This approach is also highlighted by the authors as a potential for future work.

2.5.2 Detecting anomalies in the RICH header [29]

Kwiatkowski looks further at how anomalies within the rich header and how they could be used for malware detection [29]. The main motivation for the author came from the article published by Kaspersky looking at rich header manipulation [27], and how the rich header could be leveraged to improve malware detection in Manalyze[9]. While highlighting multiple ways it could potentially be used, the author has however found one specific feature that he has implemented into Manalyze. The rich header will often report a number of imports larger than the actual number of imports, but a legitimate executable will never report *less* imports. The author generally has two hypothesis when there is a discrep-

[9]"*A static analyzer for PE files*" [30].

ancy between the number of imports and actual imports: (1) the PE file has been packed and replaced the import address table, and (2) the original rich header has been modified or replaced [29].
This can not be considered expansive research and it does not present any evidence to back up its claims. There is no way of knowing how many samples were analyzed and what the true positive[10] rates could potentially be, as this was not presented. It does however remain a interesting observation that could be further analyzed in future research. As it is published by a senior researcher at Kaspersky and the creator of Manalyze, a popular open source tool for static file analysis, it does have some merit and credibility attached to it.

2.5.3 Leveraging the PE Rich Header for Static Malware Detection and Linking [31]

Dubyk has looked further at the PE Rich Header and how it can be used for static malware detection and creating links between different samples [31]. The author's approach consists of looking at 350 samples, extracting their rich headers, creating hashes for the extracted data, and creating links between the samples in a graph overview. Links were created based on similarity between samples, using a exact match for Imphash, Rich, and RichPV. Ssdeep and impfuzzy links were created based on a pre-determined threshold of 80%. A threshold that has been determined based on previous research which highlighted it as a good threshold for high fidelity similarity comparisons [32]. The research was focused on the Rich and RichPV hashes, as these were the hashes produced based on the rich header. Other known hashing techniques were also used for benchmarking purposes: md5, ssdeep, impfuzzy, and imphash. Rich and RichPV was evaluated using two techniques: (1) looking at the generated graph's density and (2) looking at the network density. These are graph theory Link-based Object Classifcation techniques [31].
Results indicate that the proposed hashes are able to outperform some of the traditional techniques related to malware detection and classification [31]. Showcasing that the rich header is a powerful source of data that could be leveraged to further improve the capabilities of malware analysts in the future.
While results have been promising, the author has also highlighted some limitations that exist with the rich header. The two proposed hashes require the rich header to be sufficiently long to ensure that the created hashes are unique [31]. Furthermore, the rich header is not required for a executable to be functional; it is not relied upon. We would also like to highlight that only 350 samples were analyzed. This is a very minimal amount of samples and results may largely differ if this number was increased.

[10]A true positive shows something is there when there is something there; correct classification.

2.5.4 PE-Header-Based Malware Study and Detection [33]

The PE header consists of multiple sections that are of interest for malware analysis. Yibin Liao has in his research paper looked further at 3 of these: the file header, optional header and section header [33]. He has identified a total of 5 features that are used in his malware detection approach, displayed in table 2.3. A combination of all the features results in a 99.5% detection rate of malware and a false positive rate of 0.16% for benign samples. While looking at the PE header, he also identified the file icon as another feature of interest. Showing indications that malware commonly uses icons that intend to be misleading, and even in some cases uses icons that are seldom used by benign software.

Table 2.3: A table of key features in the PE header for malware detection [33].

Index	Key Features	Malware (5598)	Normal (1237)	Difference
1	Size Of Initialized Data == 0	1626 (29%)	0 (0%)	29%
2	Unknown Section Name	2709 (48.4%)	16 (1.3%)	47.1%
3	DLL Characteristics == 0	5335 (95.3%)	401 (32.4%)	62.9%
4	Major Image Version == 0	5305 (94.8%)	486 (39.3%)	55.5%
5	Checksum == 0	5084 (90.8%)	474 (38.3%)	52.5%

This brief research showcases the potential within the PE header and how it can be used to identify malware. The limited amount of features used also indicate that large sections of the header is not needed for successful analysis.
The author does however highlight a weakness with his approach: it is not able to detect all malware. He attributes this to the features that have been used, where usage of different or additional features will potentially be able to aid in the detection of different types of malware.

2.5.5 A New Classification Based Model for Malicious PE Files Detection [34]

A lot of research has been put into the problem surrounding classification of malware. Common with all these methods are that they aim to look at a dataset of both malicious and benign files, determining a set of features, and performing specific techniques to try and best determine the type of file being analysed. In 2019, Abdessadki and Lazaar published a research paper providing a substantial look into existing methods and tried to find a new and improved classification model [34]. A goal that they successfully managed to accomplish.
The authors used pefile[11] to extract relevant features, 54 to be exact, from the

[11]A python module that reads PE files [35].

headers of the files and used these as input in their classifier. To be able to determine the best classifier, they used their features on multiple classification methods, and thereby statistically determining which was the best in terms of accuracy and speed. Based on this approach their best classifier was with the use of Random Forests, reaching an accuracy of 99.74%. Also concluding that it used a reasonable amount of time to achieve this result: under 2 seconds to classify the 211 067 samples. An improvement of 0.24% to the best existing method. This method is also highlighted in this thesis, in section 2.5.4.
While showing a substantial ability to efficiently classify files as benign or malicious, there is no mention of packed or obfuscated files. Thereby it is not possible to determine whether the dataset that has been used has excluded such files. As it is not mentioned in the research paper, our assumption would be that they have chosen to exclude such files.

Chapter 3

Method

This chapter focuses on our methodology and presents our approach regarding classification, triage, feature selection and pre-processing. We have also included further information about the datasets that were used and our hardware setup. Some of the challenges we faced in regards to our hardware setup has also been included, as it could provide a useful insight for similar research in the future.

3.1 Classification and triage

Classification of samples into either benign or malicious is an important aspect of malware analysis. Before one can perform a subsequent triage of samples one would ideally also try to differentiate between the two classes. In relation to classification we used a anomaly detection approach. This meant that we would analyze one part of the dataset, either benign or malicious, and use this as the baseline, also referred to as the training set, for normal; representation of that specific class. For the training dataset we reviewed the frequency of the features, i.e. how often certain features occurred and the number of times they were observed. The more often a feature was observed in the training dataset the stronger the relationship between that feature and the class became. The training set essentially provided an overview of features that one would expect for that certain class. Features from the testing dataset was gathered in similar fashion but these were compared to the the training dataset and classified one by one. As a sample in the testing dataset was handled it would also be classified before moving on to the next sample. If a significant amount of feature values in a sample from the testing dataset was the same as the ones observed in the training dataset the sample would lean more towards the class used in the training dataset. If a significant number of unknown feature values were observed the sample would shift towards the other class. To further specify: the training and testing datasets consisted of only one type of class, either benign or malicious, and not a mix of the two.
What determines a significant number of observations and what threshold values were used to allow for classification? Throughout our research we used a threshold value of 0 for the section names and a threshold of 0.5 for compids

from the rich header. These thresholds were found by manually analyzing the output from the feature extraction process and how well different thresholds were able to classify samples. While the section names and rich header used specific thresholds, the PE header features were treated differently. For the PE header features, each feature was ranked based on its frequency in the training dataset. A feature could be observed in 0-20%, 20-40%, 40-60% or 80-100% of all samples in the training dataset. To better represent the nature of the observations related to the PE header we added exponential weighting, where features that were observed less were weighted more heavily. We wanted to increase the significance of the features observed less in the training dataset as this would mean the file would lean more towards the opposite class. The method we used to add weighting is presented in 3.1. A sample from the testing dataset would lean more towards the same group as the training dataset if the number of feature in the 80-100% would be larger than the ones in the 0-20%, after weights had been added.

Code listing 3.1: A function that adds weights to the PE header features.

```
#Reduces the significance of features that are further away from the index 0 in
#freq_dp
def _add_weighting_freq(self, freq_dp):
        i = 0

        while i < len(freq_dp):
            freq_dp[i] = freq_dp[i]/(i+1)
            i += 1
        self.freq_dp = freq_dp
```

Our classifier would use three main groups that acted as a collection of features. These collections were used to represent the PE header, section names and rich header. When a sample from the testing dataset was analyzed it would group feature values in the PE header based on the number of occurrences it had in total in the training dataset. A simple example: if the feature *Checksum* had the value *0* in 0-20% and value *128* in 0-40% of the samples in the training dataset, a sample with the *Checksum* value of *0* would add 1 to a counter within the 0-20% group. Similar comparisons would be done for each individual feature in the PE header. Features in the section names and rich header were compared on a 1-to-1 basis. A list of all feature values that had occurred would be available from the training dataset. Each feature value in the sample from the testing dataset would be compared and a percentage would be calculated based on the number of features that were the same. More in depth detail regarding our implemented method of classification is added in Appendix A, specifically code listings A.5 and A.6.
On a high level the approach we had towards the classification and triage problem can be regarded as a try-it-all approach. We wanted to try as many different combinations as we had the time to do and try to gain insight based on the difference in results. Classification was accredited to tests that used opposing datasets and the benign dataset comparisons. The triage technique was represented by the malicious dataset comparisons. This meant that when we tested using e.g. benign

as the baseline and malicious as the testing dataset, we were looking at our ability to correctly classify samples. A possibility we had as the datasets were labelled. This gave us a insight into how different combinations of feature groups impacted our ability to classify and allowed us to find the best combinations. These results are presented in sections 4.1 and 4.2. By doing identical tests with the NTNU dataset and the larger Norton LifeLock dataset we were also able to determine our abilities to classify when the number of samples were increased significantly. The results obtained through the comparison of both the malicious datasets, as presented in section 4.4 is the foundation of our triage suggestion. This is because it is able to determine which samples from the testing dataset is similar to already observed samples in the training dataset. Testing different feature combinations allowed us to find the combinations that were able to provide the most solid results. The comparison of benign datasets, presented in section 4.3, followed the same process as the triage technique but used a different class of samples. These results were not considered as a triage approach, but a test to see the ability of our classifier to correctly classify samples when the same class was used in both training and testing. In addition, it acted as a test to see whether features were impacted by the time the datasets were created - a possibility we had since the NTNU dataset was roughly 6 years old and to our knowledge the Norton LifeLock dataset was collected more recently.

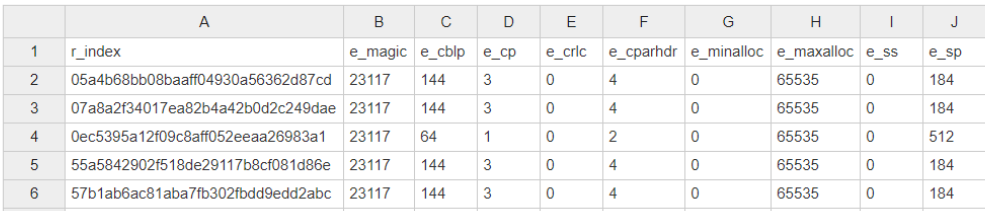
3.2 Feature selection and pre-processing

The ability to pre-process samples and select features was an important first step in the analysis stages. It was the fundamental aspects that a future comparison of samples would be based on. We first had to identify the set of features we would use to represent a given sample, and the amount of features that were to be used. Simultaneously we had to consider the need for storage space and how the processing times were impacted by the increased complexity. Increased complexity is considered when more features need to be processed, if a sample is represented by one feature it would be significantly less complex compared to a sample represented by 100 features. The latter would require more processing time. To reduce the complexity of our research, both in terms of necessary processing complexity and a reduction in the risk of accidentally infecting the host system with malware, we decided that we would only use static analysis of the samples. All feature extraction was conducted without executing; running the samples. The main feature groups that were used in our research is presented in table 3.1. The DOS header, NT header, File header and Optional header provided a total of 55 individual features, the Rich header provided two and the section names provided one. A more detailed overview of all the individual features that were extracted from these groups is attached in Appendix B.
A brief overview of figure 3.1 displays that each sample is represented by one entry, or row, in a larger table. Each column represents an individual feature and the related value that a specific sample has. All these features combined was what

Table 3.1: A simplified list of the feature groups that were used.

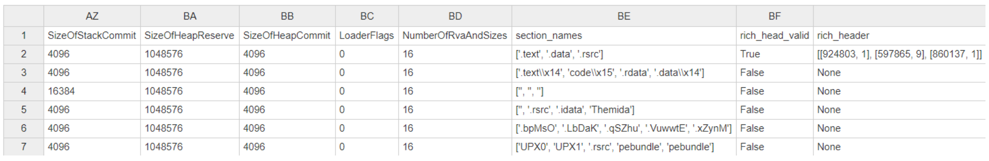
	PE header	**Rich header**	**Section names**
	DOS header	Rich header	Section names
	NT header		
	File header		
	Optional header		
Total # features	55	2	1

we used to describe the samples and it created the foundation for further analysis. Figures 3.1 and 3.2 provides an illustration of how the beginning and end of such a row would look like. Figure 3.2 also clarifies how the section names and rich header is included. The section names are added as a list within the row, as they are a list of names where additional unnecessary formatting had been removed. Additional formatting was present as the section names was originally a byte representation which we converted to strings for easier processing. The rich header was treated the same way as the section names, but where no additional removal of unnecessary formatting was needed.



	A	B	C	D	E	F	G	H	I	J
1	r_index	e_magic	e_cblp	e_cp	e_crlc	e_cparhdr	e_minalloc	e_maxalloc	e_ss	e_sp
2	05a4b68bb08baaff04930a56362d87cd	23117	144	3	0	4	0	65535	0	184
3	07a8a2f34017ea82b4a42b0d2c249dae	23117	144	3	0	4	0	65535	0	184
4	0ec5395a12f09c8aff052eeaa26983a1	23117	64	1	0	2	0	65535	0	512
5	55a5842902f518de29117b8cf081d86e	23117	144	3	0	4	0	65535	0	184
6	57b1ab6ac81aba7fb302fbdd9edd2abc	23117	144	3	0	4	0	65535	0	184

Figure 3.1: Displays the beginning of the row.



	AZ	BA	BB	BC	BD	BE	BF	
1	SizeOfStackCommit	SizeOfHeapReserve	SizeOfHeapCommit	LoaderFlags	NumberOfRvaAndSizes	section_names	rich_head_valid	rich_header
2	4096	1048576	4096	0	16	['.text', '.data', '.rsrc']	True	[[924803, 1], [597865, 9], [860137, 1]]
3	4096	1048576	4096	0	16	['.text\\x14', 'code\\x15', '.rdata', '.data\\x14']	False	None
4	16384	1048576	4096	0	16	['', '', '']	False	None
5	4096	1048576	4096	0	16	['', '.rsrc', '.idata', 'Themida']	False	None
6	4096	1048576	4096	0	16	['.bpMsO', '.LbDaK', '.qSZhu', '.VuwwtE', '.xZynM']	False	None
7	4096	1048576	4096	0	16	['UPX0', 'UPX1', '.rsrc', 'pebundle', 'pebundle']	False	None

Figure 3.2: Displays the end of the row. This includes elements from all feature groups.

3.3 Datasets

Our research was based on two different datasets. These were used to determine which combination of features were the best by reviewing our approach on both a smaller dataset and a larger one. The datasets were divided into two parts where one part consisted of benign samples and the second of malicious samples. In both

datasets the number of malicious samples were significantly higher than benign ones.
Working with labelled datasets has both its advantages and disadvantages. We were able to continuously test and verify our ability to correctly classify samples. A significant advantage during development as we could see how the classifier improved or worsened depending on the configuration that was used. As both datasets were separated into benign and malicious samples we could also see how our classification was able to perform by gathering results by using both classes as the baseline dataset. While this gives us a good understanding of how the classifier is able to perform, it does not tell us how well it would handle a realistic dataset. We did also not have any pre-determined knowledge about the samples. This resulted in our triage process being more exploratory.
Our research was based on the analysis of PE files. This is also represented in the datasets we used, as all samples that were used in our analysis were valid PE files. A valid PE file can be determined by its header - a invalid PE file will be detected by the pefile python package. When we claim that all files used in our analysis were valid it is because we removed the files that were deemed as invalid by pefile. This did not mean that the samples themselves were invalid files, but it meant that they were not valid PE files. These files were most likely different file types that had been added during the initial collection. Because our research was limited to the analysis of PE files we removed the other samples before any further analysis was conducted. The number of files that were removed and the datasets they came from is presented in table 3.2.

Table 3.2: The number of invalid samples found in the datasets.

Dataset	Invalid samples
NTNU dataset (Benign)	38
NTNU dataset (Malicious)	0
NortonLifeLock dataset (Benign)	0
NortonLifeLock dataset (Malicious)	227

3.3.1 NTNU dataset

The first dataset that was acquired had initially been developed for various research projects at NTNU [36, 37]. While it is decent in size it is a subset of a larger dataset that had been used in previous research projects by the Testimon Research group at NTNU. Due to its origin we have called it the NTNU dataset, hereby referred to as the small dataset, but it is also known as the IJCNN dataset. The dataset consisted of a total of 9823 samples from 10 different malware families, a more detailed breakdown is presented in table 3.3.
Samples in the small dataset were collected in 2015 from the following sources: maltrieve, VirusShare, VxHeaven and various samples that had been shared by students [36]. The samples are evenly distributed between packed and unpacked

files with an approximate 50% split.
A concern to highlight with this dataset is its age. At the time of writing the samples are at a minimum 6 years old. This may have caused misleading results as technology and malware authors will have changed and improved their techniques over the years that have passed. Malware from several years ago may look drastically different than malware that is being used today. This could be anything from minor modifications on older malware to the development of new and more sophisticated malware.

Table 3.3: Details of malware samples in the small dataset

Family	**Samples**
agent	1000
hupigon	1000
obfuscator	1000
onlinegames	1000
renos	1000
small	1000
vb	1000
vbinject	1000
vundo	823
zlob	1000
Sum	**9823**

To supplement the malicious samples we also included benign samples that also were provided by NTNU to complete the small dataset. The samples were collected in relation to existing research projects at NTNU [38, 39]. They were collected from Portable Apps in 2019 in a grab-it all approach - where as many samples were needed in a short amount of time [38]. The dataset consisted of various types of software: such as editors, games, and various software provided by Windows. As portable apps is a webpage that consists of samples that are updated periodically there may be minor differences in a newer collection compared to the 2019 version. The dataset itself consisted of 1832 benign samples; x64 Windows executables. However, 38 samples had invalid PE headers and were not included in further analysis. This left us with a total of 1794 benign samples that were included in the small dataset.

3.3.2 Norton LifeLock dataset

After initial development and testing through the use of the small dataset provided by NTNU we were provided with a much larger dataset from our partner, Norton LifeLock. We will hereby refer to this dataset as the large dataset. The large dataset consisted of both benign and malicious samples, but the number of malicious samples significantly outnumbered the benign ones. There were a total of 10 891 benign samples and 328 800 malicious samples. The combination of these samples

created what we refer to as the large dataset.
To provide more detail about the malicious samples we have included a more detailed overview in table 3.4. Unlike the details provided about the small dataset, this table is not a 1:1 listing of the malicious samples. Samples can be a part of none, one or multiple malware families. This causes some discrepancy in the overview of the samples compared to table 3.3 from the previous section. However, it aims to provide some insight into the samples that were used in our research.

Table 3.4: Details of malware samples in the large dataset.

Family	Samples	Family	Samples
adposhel	1656	agent	13438
allaple	5021	alman	180
bancteian	912	banker	310
benjamin	8753	bifrose	196
bladabindi	1368	blocker	148
brontok	201	coinminer	10657
cosmu	251	crypt	273
darkkomet	232	detroie	182
dialer	607	dinwod	3745
eggnog	657	emotet	2796
expiro	1489	farfli	164
fasong	607	fearso	344
floxif	2758	fsysna	3299
fugrafa	170	gandcrab	6333
glupteba	404	hematite	9989
hupigon	371	ipamor	1283
ircbot	656	jeefo	410
juched	133	keylogger	122
koutodoor	154	lamer	5058
locky	107	lolbot	1150
lunam	5646	mabezat	176
madangel	109	mansabo	4910
mydoom	3838	neshta	4416
nimnul	271	nitol	1255
nymaim	216	padodor	9137
pakes	613	parite	1086
picsys	4488	pluto	1754
qqpass	120	qukart	4279
ramnit	2449	renamer	434
ribaj	1174	rozena	129
runouce	1198	sality	5678
shipup	1917	shodi	571
sivis	4665	skybag	100
small	6528	softcnapp	4492

Table 3.4 continued from previous page.

Family	Samples	Family	Samples
soltern	5250	starter	224
startpage	4579	stormattack	1330
swisyn	769	swrort	396
sytro	9182	tinba	154
toffus	256	tofsee	178
unruy	9455	upatre	4796
urelas	2989	ursnif	317
vbkrypt	162	viking	513
vilsel	638	virlock	9986
virut	9988	vobfus	586
vundo	1921	wabot	9905
warezov	248	xorist	129
zegost	118		
Sum	**232302**		
Unknown	96498		
Total	**328800**		

The benign samples in the large dataset were gathered through the use of a Windows 10 VM. All community maintained packages from the Chocolatey package manager were installed on the VM. After which all executables found on the system were collected and scanned through VirusTotal[1]. Executables that were found to have more than 3 detections in the VirusTotal analysis were removed from the dataset, which left 95% of the remaining files with 0 detections. A detection in VirusTotal is when a existing AntiVirus or URL/domain blacklisting service flags the analyzed sample or URL as malicious [40]. This left 5% of the files within the 1-2 detections range, meaning not all samples could be considered 100% benign. Additionally, samples that have 0 detections in VirusTotal could still be malicious as it could be malware that is yet to be detected. As community maintained packages from Chocolatey go through rigorous review and testing we will assume, for our research, that the files that remained and were used in the benign dataset were benign.

3.4 Hardware setup

Programming and analysis of samples were conducted through the use of a virtual machine with a 64-bit version of Ubuntu 16.04 through the VMware Workstation 16 player. The VM was stored on a external SSD with a 128GB capacity. The main host system consisted of a Intel(R) Core(TM) i5-8600k CPU @ 4.7GHz with 6 cores and 6 threads and 16GB DDR4 RAM @ 3200MHz. The purpose of storing the VM on an external SSD was to enable mobility in terms of workspace. A addi-

[1]A website that allows analysis of files and URLs to detect different types of malware [40].

tional host system was used when on the move. It had the following specifications: Intel(R) Core(TM) i5-7200U CPU @ 3.1GHz with 2 cores and 4 threads and 8GB DDR4 RAM @ 2133 MHz. Both systems utilized NVMe SSD storage. The VM was allocated 40GB at the beginning of the project, but this was further increased to 75GB. An additional 500GB SSD storage device was used to store the samples from the larger dataset locally.

3.4.1 Challenges

One of the larger challenges within our research was the use of the VM and allocation of appropriate space. We significantly underestimated the amount of storage space that would be needed and created a VM with a total of 40GB of storage. This was filled relatively fast when we started to conduct analysis with our small dataset, as this was stored on the VM. The lack of space started to cause issues with the VM. It would crash and be unable to boot from preserved states; it required a full restart every time. Fortunately more space was available and we further increased the storage space to 75GB shortly after it was filled. The increase was sufficient when working with the small dataset, however this was not the case when provided with the large dataset. To allow us to download and work with the large dataset we had to acquire a new storage device that would be able to store all the samples. To solve this issue an existing SSD device was reformatted and repurposed to only store samples from the large dataset. The SSD had a capacity of 500GB. This was a important lesson for us. It is important to try and anticipate much larger need for storage space in the beginning, especially when working with datasets similar to ours. This would have reduced the amount of time we had to spend during the research to work around and expand our storage capacity. A unnecessary challenge that removed time from our research. This is especially the case when working when working with samples directly. The use of the cloud as a main storage medium and using the VM as a cache to temporarily store and extract features from the executables before deletion would have reduced the need for physical storage. However, due to time constraints of the thesis this did not become a focus area within our research.

Chapter 4

Results

This chapter presents the results that have been obtained through the analysis of the features collected from the PE header and Rich header of the various samples. Section 4.1 presents the results that were obtained by using the small dataset, see section 3.3.1 for more detail. Section 4.2 presents the results that were obtained using the large dataset, see section 3.3.2 for more detail. In addition to looking at the datasets isolated from each other, we also gathered results where samples from the same class was used. Here a combination of both the small and large datasets were used. Results obtained through comparison of the benign samples are presented in section 4.3 and malicious samples in section 4.4.
The feature field type in the tables presents the combination of features that were used in the classification process, which builds from the three main groups: PE header, section names and rich header. A combination would consist of the different combinations of the three main groups that were used to obtain the specific results. This could range from simply using one of them, or a combination of multiple. The feature type column aims to highlight the combination of the specific row. Our research also reviewed how the use of different operators would impact our results, as such we experimented with different combinations of *AND* and *OR* to observe if this would cause any large variations in the results.
Results presented in sections 4.1, 4.2 and 4.3 displays percentages in relation to the classification rate. Percentages on the left represent correct classifications and the right represent misclassifications. The percentage itself is a representation of the portion of files that have been classified, either correctly or incorrectly, out of the total number of files in the testing dataset. In section 4.4 we presented the percentages in the form of malicious- and benign classifications. This was to make it easier to distinguish our results as the focus in this section was on triage and not correct classification.
The percentages were rounded up to the nearest second decimal and the elapsed time was rounded to the nearest second.

4.1 Small dataset

Results gathered from the small dataset is split in two: Table 4.1 used the benign samples as a method to try and create a feature representation of benign. Features from the malicious samples were compared against the established baseline to see how well different feature combinations were able to classify. In table 4.2 we switched the dataset around and instead used the malicious samples as the initial feature representation.

Table 4.1: Benign as the baseline for normal and malicious samples as input.

Feature type	Class. %	Misclass. %	Elapsed time (s)
PE header	77.83%	22.17%	164
Section names	26.30%	73.70%	162
Rich header	37.00%	63.00%	160
PE header and section names and rich header	2.21%	97.79%	157
PE header and section names or rich header	55.43%	44.57%	158
PE header or section names and rich header	79.47%	20.53%	**156**
PE header or section names or rich header	**94.09%**	**5.91%**	160
PE header and section names	20.66%	79.34%	163
PE header or section names	83.47%	16.53%	164
PE header and rich header	24.74%	75.26%	165
PE header or rich header	90.08%	9.92%	163
Section names and rich header	3.86%	96.14%	163
Section names or rich header	59.43%	40.57%	164

Table 4.2: Malicious as the baseline for normal and benign samples as input.

Feature type	Class. %	Misclass. %	Elapsed time (s)
PE header	19.12%	80.88%	**82**
Section names	66.56%	33.44%	85
Rich header	46.10%	53.90%	**82**
PE header and section names and rich header	**92.36%**	**7.64%**	83
PE header and section names or rich header	25.75%	74.25%	83
PE header or section names and rich header	16.83%	83.17%	**82**
PE header or section names or rich header	8.58%	91.42%	88
PE header and section names	72.01%	27.99%	86
PE header or section names	13.66%	86.34%	86
PE header and rich header	53.46%	46.54%	87
PE header or rich header	11.76%	88.24%	**82**
Section names and rich header	90.08%	9.92%	**82**
Section names or rich header	22.58%	77.42%	83

4.2 Large dataset

Results from the large dataset were gathered the same way as the small dataset, where we looked at benign and malicious as the baseline in the two different tables. The results are presented in tables 4.3 and 4.4.

Table 4.3: Benign as the baseline for normal and malicious samples as input.

Feature type	**Class. %**	**Misclass. %**	**Elapsed time (s)**
PE header	80.65%	19.35%	19698
Section names	31.76%	68.24%	20059
Rich header	39.11%	60.89%	19718
PE header and section names and rich header	8.15%	91.85%	20300
PE header and section names or rich header	59.97%	40.03%	19705
PE header or section names and rich header	81.52%	18.48%	20395
PE header or section names or rich header	**87.50%**	**12.50%**	20854
PE header and section names	29.01%	70.99%	21520
PE header or section names	83.40%	16.60%	20405
PE header and rich header	34.15%	65.85%	19238
PE header or rich header	85.61%	14.39%	**18958**
Section names and rich header	9.01%	90.99%	19147
Section names or rich header	61.86%	38.14%	19106

Table 4.4: Malicious as the baseline for normal and benign samples as input.

Feature type	**Class. %**	**Misclass. %**	**Elapsed time (s)**
PE header	13.61%	86.39%	4004
Section names	98.83%	1.17%	**3946**
Rich header	47.10%	52.90%	4180
PE header and section names and rich header	**99.69**%	**0.31**%	4373
PE header and section names or rich header	46.34%	53.66%	4465
PE header or section names and rich header	13.59%	86.41%	4074
PE header or section names or rich header	5.09%	94.91%	4342
PE header and section names	98.93%	1.07%	4268
PE header or section names	13.52%	86.48%	4183
PE header and rich header	55.55%	44.45%	4086
PE header or rich header	5.16%	94.84%	4211
Section names and rich header	99.66%	0.34%	4337
Section names or rich header	46.27%	53.73%	4835

4.3 Benign dataset comparison

As we worked with classification of malicious and benign samples we also wanted to see how well we would be able to classify samples as benign when using benign samples from both our datasets against each other. These results are presented in tables 4.5 and 4.6.

Table 4.5: Benign samples from the small dataset as baseline for normal and the benign samples from the large dataset as input.

Feature type	Class. %	Misclass. %	Elapsed time (s)
PE header	35.30%	64.70%	225
Section names	96.77%	3.23%	275
Rich header	47.04%	52.96%	217
PE header and section names and rich header	**99.03%**	**0.97%**	236
PE header and section names or rich header	46.23%	53.77%	**216**
PE header or section names and rich header	34.86%	65.14%	217
PE header or section names or rich header	11.99%	88.01%	226
PE header and section names	98.21%	1.79%	229
PE header or section names	33.86%	66.14%	218
PE header and rich header	69.36%	30.64%	255
PE header or rich header	12.99%	87.01%	217
Section names and rich header	98.59%	1.41%	243
Section names or rich header	45.23%	54.77%	225

Table 4.6: Benign samples from large dataset as baseline for normal and the benign samples from the small dataset as input.

Feature type	Class. %	Misclass. %	Elapsed time (s)
PE header	16.05%	83.95%	149
Section names	87.96%	12.04%	150
Rich header	46.10%	53.90%	183
PE header and section names and rich header	**96.66%**	**3.34%**	166
PE header and section names or rich header	41.03%	58.97%	200
PE header or section names and rich header	13.49%	86.51%	164
PE header or section names or rich header	7.36%	92.64%	229
PE header and section names	91.58%	8.42%	147
PE header or section names	12.43%	87.57%	183
PE header and rich header	53.73%	46.27%	181
PE header or rich header	8.42%	91.58%	196
Section names and rich header	94.10%	5.90%	**146**
Section names or rich header	39.97%	60.03%	167

4.4 Malicious dataset comparison

As the previous section looked at our ability to classify samples into the benign category, this section further explores the possibility of triage based on the given classification. Here our results are presented in the percentage of malicious and benign, instead of classification and misclassification. This is to make the results easier to reflect upon since the priority was triage and not classification. The obtained results are presented in tables 4.7 and 4.8. The reason we view this as a triage technique is because of the samples that are classified as benign. These are more interesting to review as they deviate from what is considered normal based on the already analyzed malicious samples.

Table 4.7: Malicious samples from the small dataset as the baseline for normal and malicious samples from the large dataset as input.

Feature type	**Malicious %**	**Benign %**	**Elapsed time (s)**
PE header	53.31%	46.69%	13654
Section names	30.70%	69.30%	13580
Rich header	39.11%	60.89%	13643
PE header and section names and rich header	6.03%	93.97%	13589
PE header and section names or rich header	52.49%	47.51%	14393
PE header or section names and rich header	56.66%	43.34%	14347
PE header or section names or rich header	**76.53**%	**23.47**%	13638
PE header and section names	19.41%	80.59%	14163
PE header or section names	64.60%	35.40%	13560
PE header and rich header	23.83%	76.17%	**13562**
PE header or rich header	68.59%	31.41%	14551
Section names and rich header	9.38%	90.62%	14590
Section names or rich header	60.43%	39.57%	13881

Table 4.8: Malicious samples from the large dataset as the baseline for normal and malicious samples from the small dataset as input.

Feature type	**Malicious %**	**Benign %**	**Elapsed time (s)**
PE header	71.46%	28.54%	4050
Section names	16.32%	83.68%	3845
Rich header	32.52%	67.48%	**3785**
PE header and section names and rich header	0.76%	99.24%	4043
PE header and section names or rich header	43.69%	56.31%	4175
PE header or section names and rich header	71.96%	28.04%	4304
PE header or section names or rich header	**84.65**%	**15.35**%	3949
PE header and section names	11.94%	88.06%	4258
PE header or section names	75.84%	24.16%	4289
PE header and rich header	23.21%	76.79%	4252
PE header or rich header	80.77%	19.23%	4259
Section names and rich header	1.26%	98.74%	4196
Section names or rich header	47.57%	52.43%	3958

Chapter 5

Discussion

This chapter discusses the results presented in chapter 4 and aims to highlight both the insight gained as well as the weaknesses with the conducted research. We start by reviewing the results where opposite datasets were used, benign and malicious, before discussing our findings when the same dataset types were used. In the end we will provide suggestions for future work based on our observations and findings.

5.1 Opposing datasets

It was really interesting to see how much the results varied based on the baseline dataset, i.e. the dataset that was used for training. Results based on benign samples has a larger spread compared to the results based on the malicious samples. This was the case for both the classification itself and the different feature combinations. The latter of which was the most interesting. Feature combinations that would give a very strong result in one table could be completely different in the other, e.g. PE header in table 4.3 and PE header in table 4.4 had the classification rate of 80.65% in the first table and 13.61% in the latter. Similarly section names had 31.76% in the first and 98.83% in the latter. As this is apparent throughout our results it is important to consider what one wants to use as the baseline. Should it be by looking at benign samples and classification based on these, or could one use malicious samples instead? Research into the field commonly sees a much larger sample size of malicious samples compared to benign samples. As a result, utilization of malicious samples to a larger degree could be more beneficial as well as in the comparison of similar malicious samples to perform a triage. However, one key issue here is new and unknown malware. Malicious samples that deviate more from the norm have a larger chance of being misclassified as benign, at least in the approach we used. Another way this could be approached is the comparison of malicious samples with other malicious samples, as we have done in section 4.4. This becomes a technique in a prioritization process; triage. In this type of comparison malicious samples that were classified as benign would be the samples that one would want to review further. The ones that were correctly classified would be of less interest. This is because the samples that are correctly classified as malicious would be relatively similar to samples that have already been seen before in the baseline dataset. Here it is important to maintain that the baseline dataset should consist of samples that have already been analyzed in this case, as usage of unknown samples would defeat the purpose of the aforementioned theory.

One early observation that we made was the importance of the benign dataset. A larger benign dataset will give a better foundation for our ability to classify, which is reflected in our results in tables 4.1 and 4.3. The tables are relatively similar showing that our classification ability was somewhat viable. Drastically different results would have indicated larger uncertainty. Based on our results we found that the use of a larger benign dataset somewhat lowered the classification rate. This was not unexpected as the increase in the number of benign samples makes comparisons more ambiguous; introduces more variance. When one attempts to classify based on a lower amount of samples it becomes easier to overfit as the proper data foundation is not present. While the majority of classification rates are relatively similar among the two tables some combinations are improved when using a larger dataset, e.g. PE header and section names improves from 20.06% to 29.01%. Another common factor within both tables is the feature combination that gives the best classification rate, this remains the "PE header or section names or rich header". Important factors to keep in mind when comparing these two tables is that the data foundation is relatively different. Benign samples were collected using different methodologies and at different time periods. This could impact some of the features that are used within our program, e.g. the compiler ids within the rich header will be different in a a dataset collected in 2021 compared to 2015. This impacts the ability to determine a good basis for benign files, without a large enough sample size. If benign samples from different time periods are used the possibilities to determine malicious samples from a larger time span increases. Similarly if benign samples are compared with benign samples from a different time period one will see relatively large differentials, as presented in section 4.3. Through the comparison of the benign datasets one can see that the PE header and rich header are features that can vary quite a bit over time, as our ability to classify is quite weak with these features. Section names seems to remain more stable over time. We have not been able to obtain results with the combination of benign samples from both datasets to verify that our theory regarding classification over a longer time span holds. However, it could be considered a natural assumption as an increase in the data foundation to represent a larger time span should also improve our ability to correctly classify over that time span.

Existing research, some of which we presented in section 2.5, already points to the potential of a combination of the PE header and rich header for analysis of malicious samples. Both for detection of malicious samples and as a source of information for a preemptive triage process. An approach that focuses on classification based on anomaly detection where a frequency analysis as the basis does not seem to provide solid results, as showcased in chapter 4. There is a large variety in the results based on the features that were used. This is somewhat logical, as the use of a stricter approach leads to less samples being classified as malicious and vice versa, but it also leads to a larger number of false positives. As the classifier skews towards one class it will either favor classification of benign or malicious samples which results in a large number of true positives for one side but also false positives for the opposite.

When reviewing the results where benign is the baseline dataset, the strictest combination type provided us with a significantly small number of malicious classifications compared to a more open combination type. 2.21% of the samples were correctly classified as malicious in the small dataset, see table 4.1, and 8.15% in the large dataset, see table 4.3. If only considering classification this would be a very bad result, however since we also consider the triage possibility in our research this could act as a method to find the samples that are the most significant in the dataset - the ones that could be flagged for further analysis. Only having to review 2.21% or 8.15% of the malicious samples significantly lessens the burden on physical analysts and the resources and time they need. The reason that these files are of interest are because a significant amount of header features, section names and compiler ids deviate from the benign dataset. This means that these are the files that have very little in common with the benign samples. While this comparison is somewhat present through comparison of benign and malicious samples, one could change the approach such that two malicious datasets are compared for triage purposes. This would find the samples that are the most unique in terms of the reviewed features. These results have been gathered through our research and was presented in section 4.4.

One aspect with our research that has become more clear throughout the process is that our approach is not sufficient when purely considering classification. We have been working with determined datasets with benign and malicious samples. The ability of our program to correctly classify an "unknown" sample into benign or malicious would be very inconclusive based on the results we have obtained so far. Better methods for classification already exist, some of which we have presented in section 2.5, and due to this we reduced our focus on finding better methods of classification in our thesis. Another factor that lead to us moving away from focusing on classification was because of its time consuming nature. Our ability to provide results would have been heavily reduced if we would have dedicated even more time towards the improvement of the classification.

5.2 Same type datasets

Within the benign results presented in section 4.3 there is one major takeaway: The section names are a major feature that can be used to determine whether a sample is benign or not during classification. Section names alone were able to correctly classify 96.77% of the samples in table 4.5 and 87.96% in table 4.6. This shows that benign samples have a lot in common in terms of the section names that are set. In comparison the malicious analysis in section 4.4 only had a 30.70%

and 16.32% classification rate. These results indicate that it is easier to classify benign samples based on section names compared to malicious ones. This is further amplified when viewing tables 4.2 and 4.4, when using benign samples as the input and malicious datasets as the baseline we can see a similar pattern. In the first table section names alone lead to a 66.56% classification rate and 98.83% in the latter table. Section names in combination with other features also lead to an even greater classification rate, as seen in "PE header and section names and rich header", "PE header and section names" and "Section names and rich header". The most successful classification in terms of determining benign samples was the combination of all features: "PE header and section names and rich header", as presented in tables 4.2 (92.36%), 4.5 (99.03%) and 4.6 (96.66%). A combination of the section names and rich header provided the second best results with the largest differential being 2.56%. As the results are relatively similar, the removal of the main PE header features to reduce complexity, albeit at the cost of slightly worse results, could be a interesting area to explore further. However, one thing to keep in mind is what the main priority is. If performance is a major factor one could start to consider removal of features, but if the priority is correct classification one would want to keep the features and potentially segment them even further.

Sections 4.3 and 4.4 has one main pattern in common. The strictest combination of features leads to a better benign classification and a more open combination leads to more malicious classifications. This can be considered normal behavior, as methods that are more lenient in classification will lead to more malicious classifications but this also introduces more potential false positives.

An interesting observation is that the results were very strong when we compared benign datasets but relatively weak when we compared the malicious datasets. In the latter the strongest result was 84.65%, 12.01% worse than the worst result in the benign dataset comparison. This is based on a comparison of the best result from each respective table in sections 4.3 and 4.4. We also observed a more even spread among the results where the baseline dataset came from the small dataset, compared to tests where the baseline dataset came from the large dataset. This could be attributed to the size of dataset, where the small dataset does not contain enough samples to appropriately classify all the incoming samples. In cases where the baseline dataset contains very few samples compared to the input dataset it is more likely that the input dataset has a larger amount of features that have not been previously observed. This could subsequently lead to more samples being misclassified. We believe that this shows how it is important to have a relatively large dataset when utilizing anomaly detection techniques for the purpose of malware classification. It is very hard for a small dataset to be an appropriate representation of what is to be considered normal in the domain of all benign executables. This is also something which is represented in our results, where the use of a larger dataset lead to more confidence in the observed results. A smaller dataset could in certain domains be a good representation of "normal", e.g. communication between nodes in a industrial control system has a very specific purpose and variation in data is often minimal. Very minor deviations would in this case be considered abnormal. In this scenario one would not need a significant dataset to represent normal behavior. For PE files however, this is not very viable. Malware developers constantly update and develop new samples to try and remain undetected. This means that there will commonly be a large variation in observed samples over time.

There are some observed differences in the results in section 4.4. We believe that the main reason these difference exist can be attributed to the size of the baseline dataset. Table 4.7 uses the small dataset and table 4.8 uses the large dataset. This means that the best representation of the results should be in table 4.8, since the dataset used as the baseline is much larger. As previously mentioned we believe that the malicious dataset comparisons could be used as a component within a triage process. From our results we want to highlight how benign classifications, which in this case could be considered misclassifications, can be used as a means to perform triage on new samples. If the sample is already known to be malicious one can view the samples that are classified as malicious as uninteresting. This is because such a classification means that the sample already has a lot of features in common with existing data. Samples that are classified as benign are more interesting as these have less features in common with existing data; they are less similar. As the samples have less in common one would want to further review them manually. However, this leaves some expectations on the dataset. The baseline dataset should consist of malicious samples that have already been analyzed and determined to be uninteresting for further manual review. The input dataset should consist of samples that have already been classified as malicious. With the introduction of a dataset consisting of both malicious and benign samples as the testing dataset one may also observe a larger amount of true positives in terms of classification; benign samples that are classified as benign. While this is generally good, it partially defeats the purpose of using the samples classified as benign in a triage process. It could also lead to more uncertainty for an analyst because one would also need to spend time reviewing benign samples that have been flagged for manual analysis. This could be attributed to the predetermined expectation that the samples are inherently malicious.

The best results that were gathered in section 4.4 was where 84.65% of the samples were classified as malicious. This left 15.35% of the samples classified as benign. Based on our research and the data we used we believe that these are the samples that should be prioritized for manual analysis. However, this still leads to a relatively large amount of samples that needs to be analyzed. Roughly 281 samples is still a large amount of samples to analyze manually, even though it is significantly less than 1832. Due to this we believe that it may be beneficial to use this triage method as a component in a larger process. Based on our results we can not with confidence say that the triage technique is effective enough to be the only method that is used. A longer and more effective process would be beneficial to further help reduce the samples that should be analyzed manually. This may in addition improve the confidence in the samples that are chosen, as

the ability of finding the most significant samples would likely improve.
Unlike the other results we obtained there is specifically one combination of features in section 4.4 that shows greater results compared to the others: "PE header or section names or rich header", especially in table 4.8 where the second best result is only 68.59% compared to 76.53%. This combination of features provides the best results in both tables. The "PE header or rich header" is the second best combination and it slightly improved its differential between the best result to only 3.88% in table 4.8, where the large dataset was the baseline dataset. This combination shows promise in terms of the gathered results, but is not as significant. The differential is not small enough to consider a removal of features for effectiveness of the program, an action that our results related to the elapsed times would not support. The combination that used all features was already more effective in terms of the results as well as the elapsed time.

5.3 Elapsed time

An aspect of our research also focused on the effectiveness of the program and how the use of different datasets and feature combinations impacted this statistic. This is presented in the elapsed time in all tables within chapter 4. The elapsed time shows the time that was used to run through one specific combination within our program. Elapsed times for a specific combination included the entire process of pre-processing, data extraction, analysis of both the baseline and testing datasets and using the specified combination as a method of classification. The number was rounded to the nearest whole number and is displayed in seconds. Each run through used the same methodology. Methods to improve the times were not introduced in our research, e.g. caching to improve the elapsed time between the first combination type and the remaining ones. While it is a parameter that we measured and chose to include and display in our results, they do not give us a clear indication of how different feature combinations impact the elapsed times. If we exclusively looked at the results we obtained one could say that no specific combination is a clear favorite, and that a reduction of features should be deemed unnecessary. However, our results have not been verified and as such we do not deem it as a absolute conclusion. Further tests would need to be performed to verify whether or not the theory is viable. Feature reduction is known as a process to remove features for another type of gain. A gain could include improvements such as faster processing times, need for less storage space, need for less computational power, etc. Before we conducted our research we believed feature reduction would be necessary when analyzing the large datasets. However, our results indicate that this may not be the case. It is possible that the number of features may not be significant enough to matter in terms of efficiency. Perhaps an even further segmentation of the features would be viable.
The one thing our research managed to showcase through the elapsed times was how the increased size of the datasets significantly impacted the time needed to analyze them. This is not something that was unexpected because increased

datasets naturally lead to more complexity in the pre-processing, data extraction and comparison stages. Our results also show that a significantly less amount of time was needed to analyze the samples when the largest dataset was used as the baseline dataset, as seen in the elapsed times of the tables in section 4.2. The large malicious dataset as the baseline lead to the elapsed times being reduced by roughly 20-25% compared to when the large benign dataset was used. We believe this is attributed to the difference in the number of samples, where the large malicious dataset consisted of 328 800 samples compared to the 10 891 samples in the large benign dataset. This indicates that the process of analysing samples in the testing dataset and classifying them was a more time consuming process. This means that further effectivization of the code, and potentially similar techniques, should focus on performance improvements for the incoming dataset and the comparison between the existing and new data. We believe that the introduction of techniques such as parallel processing to be able to utilize more of the processing power within CPU's would be a viable option, or potentially exploring the impact of GPU processing. A bit more surrounding this topic is presented in future work; section 6.1.

5.4 Strengths and weaknesses

Through our research we have been able to find and present which combination of features work better for classification of samples than others. Our results indicated that there were major differences in which features were significant, as seen by the large differences in classification rates in chapter 4. This also depended on the dataset that was being analyzed, as the features when a benign dataset was used as the baseline would be very different compared to analysis when the malicious dataset was used as the baseline.
While we are able to verify our results - to some extent, based on the comparisons of the same dataset types we believe that this was not sufficient. More effort needs to be made to verify that the claims we have made throughout this research are viable techniques in future malware classification and triage methodologies. More elaborate use of statistical methods would be able to accomplish this. Our approach can be considered a less established approach to the problem but it does not make it irrelevant. It brings a new perspective to the domain and shows a different way that this common problem could be approached. It may not be a viable option to use as the only means of classification and triage, but it is thought processes and components that could potentially be used within a much larger process. This would naturally require some modification and additional work in terms of verification but the foundation is present, as the source code that was used is also included in appendix A.
An unfortunate effect of the limited time we had to conduct our research lead to the problem surrounding packed and unpacked files not being considered. A file being packed or unpacked will lead to some differences in the header fields, and specifically in terms of the section names as packers can modify these names [41].

As an example we can consider a file packed through the use of UPX[1], one of the common packers that can be observed used for both benign and malicious samples - but it is more commonly used by malicious actors as a simple means of obfuscation. Files that are packed using the same packer will already have some similar features because the section names of both files will be identical. As these similarities exist it potentially became harder for our classifier to distinguish between benign and malicious as a side effect. However, we did not verify this as we did not have a specific statistic on the different packers used by all the packed samples in the dataset. We did not know the distribution between packed and unpacked either, except for within the malicious small dataset where it was roughly 50%.
Another side-effect of not having significant insight into the dataset was related to header obfuscation. Header obfuscation is a technique that is used in the wild by malware authors. Our approach should be well suited to classify files with obfuscated headers and correctly classify them, in the case where a benign dataset is used as the baseline, because an obfuscated header would deviate from a regular benign header. This is due to the fact that benign files would have no reason to have obfuscated headers. A file that deviated heavily from what was considered normal would be classified accordingly by our program and due to this we believe that the correct classification of obfuscated files would be a trivial matter. However, we do not have any solid evidence or tests to prove this as our research did not go further into the details of our datasets. We were not aware if any of the samples contained obfuscated headers or not, but considering the size of the datasets it is very likely that some did.
On a positive note we were confidently able to show what impacted the elapsed times of the program: the dataset not features. As previously mentioned, our research did not only focus on classification and triage of samples, but also on the elapsed times of our program and at which speeds we were able to analyze the datasets. As we presented in chapter 4. Noticeably the elapsed time was more affected by the size of the baseline dataset compared to the second dataset. In tests where the largest of the two datasets was used as the baseline the elapsed times were quicker compared to when the smallest dataset was used, e.g. tables 4.1 and 4.2.
While there was a clear pattern in what caused the elapsed times we observed, there was no clear indication of a combination of features that was better or more efficient than others. Results between the datasets varied significantly and based on our tests we could not conclude with a significant finding based on the observed elapsed times. Simultaneously, there was no large variation within the elapsed times when smaller datasets were used. They were very similar with minor differences. A reduction in features is something that could cause important data to be removed unknowingly and would not be recommended unless it could significantly improve the elapsed times. Our observations indicate that there are no true benefit to reducing the number of features that are used
An external factor that may have impacted the elapsed times of our program

[1]A free and high performance open source executable packer [42].

would be by not having a dedicated machine for analysis. While the analysis was executed on a VM, the main machine itself was in some cases still in regular or extensive use. This may have lead to irregular elapsed times in some cases. This raises concerns about the reliability of the gathered results in relation to the elapsed times. We do however believe that they give a firm indication of results one may expect from other tests where similar specifications are used. Naturally the elapsed times would be faster with a machine that was more capable to be used for analysis, as well as with a analysis program that would be able to utilize more of the available processing power.

Chapter 6

Conclusion and future work

We have successfully been able to present and review some of the current state of the art within malware analysis using the PE header and/or rich header as the main basis for analysis. While we have been able to present what we believe to be the current state of the art, we acknowledge that this is a domain in constant development. More techniques and different research will continuously appear with the dedication of malware researchers in academia, commercial research and development teams, analysts, and even enthusiasts. New techniques created by malicious threat actors often emerge and as a result research into this domain will most likely newer cease to exist as long as malware is being developed and used. Throughout our research we have displayed a variety of results that indicate that the combination of the PE header and rich header could be effective for analysis, but also at times ineffective. Not all combinations of features have been equally successful. While in some cases there are really strong results pointing at a specific combination, it was not equally strong when utilizing a different dataset. This could be attributed to different factors, as we have discussed in the previous chapter. Our research would benefit from additional focus on the verification of results, and due to the variety we observed we do not have the ability to provide a definite conclusion of our second research question: "Will classification based upon features from both the PE header and rich header give better results compared to the headers separately?". More work needs to be done utilizing these features before a scientifically based conclusion can be made. Our research does however indicate that there are some features combinations that show signs of promise, these may in turn be researched further to see if there are any merit to their potential.
In terms of triage techniques, and our third research question "How can a combination of features from the PE header and rich header lead to a more effective triage of samples?", we have been able to present some of the potential that the combination of features have. It was the combination of all features that lead to the most successful results in terms of triage, but our method still has some caveats. The number of samples that remained were less than the total number of samples, and it can as such be considered a promising triage technique. Even

though we saw an ability to reduce the number of samples by a relatively large amount we still believe that there is a potential to reduce even further. As such we believe that our method may be best suited as a component in a larger triage process. This would enable it to be the initial stage where unknown samples are filtered, before they go through a more substantial; dynamic analysis to determine if there is a need for manual analysis. Similarly to our other results this was also an area which would require further testing and verification before one would be able to confidently rely on the technique.
Our research acts as a new addition to the large number of solutions that aim to tackle the common problems surrounding malware classification and triage. It sheds a light on the benefits of using both the PE header and rich header for analysis. We believe our results alone can not be considered extensive enough to reach a final verdict in terms of the headers, but the potential has been presented. Additional methods using both of these headers may be able to further solidify the claims and hypotheses that we have made.

6.1 Future work

The program that has been developed can be considered a good basis for further development. We have not utilized techniques that would help improve runtimes and it became code that functions for its specific purpose; it is not instantly adaptable elsewhere. Modification and adaption would be necessary. Introduction of methods that would increase efficiency should be a priority in future or similar work. As we have presented, the focus should be more towards the incoming samples that need classification because the size of the input dataset seems to have heavily impacted the elapsed times. This could include methods such as parallel programming, where multiple threads, cores, CPU's or GPU's are used to process all of the data, addition of more "best practice" approaches to the program structure itself, or utilization of the cloud to improve the efficiency of the analysis.
Utilization of the cloud in itself should be its own key point within future work. We did not have the time to review these possibilities due to lack of experience and prioritizations that had to be made due to time limitations, but it remains an area we believe would be promising. We believe there are two main methods that one could use to create a hybrid solution: (1) Use the cloud as a storage medium. Only store files for analysis temporarily on local disks. After the features have been extracted the sample could be removed from the local disk. Essentially using the local disk as a cache and the main storage being in the cloud. A on-prem NaS[1] unit may also be used in a similar way, but very strict measures would need to be in place to avoid malware being spread unintentionally through bad handling. (2) Move the actual analysis to the cloud as well, instead of running it locally. Utilization of the cloud would allow ffor further benefits through the potential of horizontal scalability. Analysis would be able to be conducted more efficiently as

[1]Network-attached Storage [43].

more processing power would be easy to access as the demand for it increased based on the dataset size.

Our research did not look at the differences when one tries to group packed and unpacked files. We focused on the pure analysis of the samples regardless if they were packed or not. An aspect where this impact can be theorized was through the use of common packers, such as UPX. When samples from both classes can be observed using the same packer it may lead to more misclassification. In turn the use of less common packers can be an indication of malicious samples by itself. To further review the impact that packed and unpacked files have on the ability to classify and perform triage it would be beneficial to detect and split the datasets based on the type of sample it is; packed or unpacked. By doing this one would be able to collect information about the differences that may occur. Analysis of unpacked samples would be the most ideal for our classifier, as packers would impact the accuracy of the classifier and may lead to an increased number of false positives.

Further segmentation of the features may be an area to explore further, as our results did not indicate a strong relationship between different combinations of features and the elapsed times. We believe that segmenting the data even further may lead to better classification and triage results as samples and data are given a richer context. One would also be able to better determine the features that are more significant compared to others. We only dealt with three main groups from two header types. More research into this area would be needed to determine whether this would be a possibility or not based on our existing approach.

Verification of the results we gathered has been mentioned numerous times throughout this thesis. As such we find this to be a natural continuation of the work and effort that has been put into this research. The results themselves may be decent, but they may also be unreliable. Further efforts into the verification of the results, potentially through testing with cross-validation, would be beneficial to this process. At this current time a lot of statements are left up in the air as the results lack sufficient validation, at least in the researcher's opinion. With the ability to further validate or disprove the results that have been obtained one would be able to gain a better understanding of how well this approach may or may not function in its intended use case.

Bibliography

[1] P. Shakarian, J. Shakarian and A. Ruef, 'Chapter 13 - attacking iranian nuclear facilities: Stuxnet,' in *Introduction to Cyber-Warfare*, P. Shakarian, J. Shakarian and A. Ruef, Eds., Boston: Syngress, 2013, pp. 223–239, ISBN: 978-0-12-407814-7. DOI: https://doi.org/10.1016/B978-0-12-407814-7.00013-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124078147000130.

[2] Microsoft, *HAFNIUM targeting Exchange Servers with 0-day exploits*, https://www.microsoft.com/security/blog/2021/03/02/hafnium-targeting-exchange-servers/, [accessed May 20, 2021], 2021.

[3] T. Cook, *Analysis of the SolarWinds Supply Chain Attack*, https://www.guidepointsecurity.com/analysis-of-the-solarwinds-supply-chain-attack/, [accessed May 20, 2021], 2021.

[4] Malwarebytes, *Threat actor*, https://blog.malwarebytes.com/glossary/threat-actor/, [accessed May 20, 2021], 2021.

[5] FireEye Mandiant, *M-Trends 2020 Report*, https://content.fireeye.com/m-trends/rpt-m-trends-2020, [accessed May 20, 2021], 2020.

[6] Microsoft, *An overview of unsafe file types in Microsoft products*, https://support.microsoft.com/en-us/topic/an-overview-of-unsafe-file-types-in-microsoft-products-266a9bd3-50d7-d65a-8fe0-ec4e486c3a14, [accessed May 21, 2021], 2021.

[7] D. Gavrilut, *The Value Beyond the Hype: Applying Machine Learning in APT Detection*, businessinsights.bitdefender.com/machine-learning-apt-detection, [accessed May 20, 2021], 2016.

[8] Avast, *Avast Technology - Combining human genius and artificial intelligence to keep the world safe*, https://www.avast.com/technology, [accessed May 20, 2021], 2021.

[9] Comodo Cybersecurity, *Valkyrie*, enterprise.comodo.com/valkyrie/, [accessed May 20, 2021], 2021.

[10] I. Robertson-Steel, 'Evolution of triage systems,' *Emergency Medicine Journal*, vol. 23, no. 2, pp. 154–155, 2006, ISSN: 1472-0205. DOI: 10.1136/emj.2005.030270. eprint: https://emj.bmj.com/content/23/2/154.full.pdf. [Online]. Available: https://emj.bmj.com/content/23/2/154.

[11] VirusTotal, *File statistics*, https://www.virustotal.com/en/statistics/, [accessed May 21, 2021], 2021.

[12] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[13] Intel, *Intel® Inspector User Guide for Windows* OS*, https://software.intel.com/content/www/us/en/develop/documentation/inspector-user-guide-windows/top/getting-started/dynamic-analysis-vs-static-analysis.html, [accessed May 22, 2021], 2021.

[14] M. Egele, T. Scholte, E. Kirda and C. Kruegel, 'A survey on automated dynamic malware-analysis techniques and tools,' *ACM Comput. Surv.*, vol. 44, no. 2, Mar. 2008, ISSN: 0360-0300. DOI: 10.1145/2089125.2089126. [Online]. Available: https://doi.org/10.1145/2089125.2089126.

[15] I. Kononenko and M. Kukar, *Machine learning and data mining*. Horwood Publishing, 2007.

[16] The National Cybersecurity and Communications Integration Center, *File Hashing*, https://us-cert.cisa.gov/sites/default/files/FactSheets/NCCIC%20ICS_Factsheet_File_Hashing_S508C.pdf, [accessed May 21, 2021], 2021.

[17] D. Uppal, V. Mehra and V. Verma, 'Basic survey on malware analysis, tools and techniques,' *International Journal on Computational Sciences & Applications (IJCSA)*, vol. 4, no. 1, p. 103, 2014.

[18] Microsoft, *PE Format*, https://docs.microsoft.com/en-us/windows/win32/debug/pe-format, [accessed November 22, 2020], 2020.

[19] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández and E. Vázquez, 'Anomaly-based network intrusion detection: Techniques, systems and challenges,' *Computers Security*, vol. 28, no. 1, pp. 18–28, 2009, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2008.08.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404808000692.

[20] Merriam Webster, *false positive*, https://www.merriam-webster.com/dictionary/false%20positive, [accessed May 27, 2021], 2021.

[21] Research Optimus, *What is Frequency Analysis?* https://www.researchoptimus.com/article/frequency-analysis.php, [accessed May 3, 2021], 2021.

[22] Microsoft, *Portable Executable and Common Object File Format Specification 4.1*, http://bytepointer.com/resources/pecoff_v4.1.htm, [accessed November 22, 2020], 1994.

[23] A. Walenstein, D. J. Hefner and J. Wichers, 'Header information in malware families and impact on automated classifiers,' in *2010 5th International Conference on Malicious and Unwanted Software*, 2010, pp. 15–22. DOI: 10.1109/MALWARE.2010.5665799.

[24] Bytepointer, *The Undocumented Microsoft "Rich" Header*, `http://bytepointer.com/articles/the_microsoft_rich_header.htm`, [accessed November 23, 2020], 2018.

[25] stephen, *Rich Header*, `http://trendystephen.blogspot.com/2008/01/rich-header.html`, [accessed November 23, 2020], 2008.

[26] G. D. Webster, B. Kolosnjaji, C. von Pentz, J. Kirsch, Z. D. Hanif, A. Zarras and C. Eckert, 'Finding the needle: A study of the pe32 rich header and respective malware triage,' in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017, pp. 119–138.

[27] GReAT, *The devil's in the Rich header*, `https://securelist.com/the-devils-in-the-rich-header/84348/`, [accessed November 28, 2020], 2018.

[28] E. Chanlett-Avery, J. W. Rollins, L. W. Rosen and C. A. Theohary, *North Korean Cyber Capabilities: In Brief*. Congressional Research Service, 2017.

[29] I. Kwiatkowski, *Detecting anomalies in the RICH header*, `https://blog.kwiatkowski.fr/?q=en/rich-header`, [accessed November 29, 2020], 2018.

[30] I. Kwiatkowski, *Manalyze*, `https://github.com/JusticeRage/Manalyze`, [accessed November 29, 2020], 2020.

[31] M. Dubyk, 'Leveraging the pe rich header for static malware detection and linking,' 2019.

[32] J. Oliver, C. Cheng and Y. Chen, 'Tlsh–a locality sensitive hash,' in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, IEEE, 2013, pp. 7–13.

[33] Y. Liao, 'Pe-header-based malware study and detection,' *Retrieved from the University of Georgia: http://www. cs. uga. edu/ liao/PE_Final_Report. pdf*, 2012.

[34] I. Abdessadki and S. Lazaar, 'New classification based model for malicious pe files detection.,' *International Journal of Computer Network & Information Security*, vol. 11, no. 6, 2019.

[35] E. Carrera, *Python PE parsing module*, `https://pypi.org/project/pefile/`, [accessed November 22, 2020], 2019.

[36] A. Shalaginov and K. Franke, 'Automated intelligent multinomial classification of malware species using dynamic behavioural analysis,' in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, 2016, pp. 70–77. DOI: `10.1109/PST.2016.7906939`.

[37] A. Shalaginov, L. S. Grini and K. Franke, 'Understanding neuro-fuzzy on a class of multinomial malware detection problems,' in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 684–691. DOI: `10.1109/IJCNN.2016.7727266`.

[38] S. Banin and G. O. Dyrkolbotn, 'Detection of running malware before it becomes malicious,' in *Advances in Information and Computer Security*, K. Aoki and A. Kanaoka, Eds., Cham: Springer International Publishing, 2020, pp. 57–73, ISBN: 978-3-030-58208-1.

[39] S. Banin, 'Fast and straightforward feature selection method,' in *Malware Analysis Using Artificial Intelligence and Deep Learning*, M. Stamp, M. Alazab and A. Shalaginov, Eds. Cham: Springer International Publishing, 2021, pp. 455–476, ISBN: 978-3-030-62582-5. DOI: `10.1007/978-3-030-62582-5_18`. [Online]. Available: `https://doi.org/10.1007/978-3-030-62582-5_18`.

[40] Virustotal, *How it works*, `https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works`, [accessed April 26, 2021], 2021.

[41] B. Jung, S. I. Bae, C. Choi and E. G. Im, 'Packer identification method based on byte sequences,' *Concurrency and Computation: Practice and Experience*, vol. 32, no. 8, e5082, 2020.

[42] The UPX Team, *UPX, the Ultimate Packer for eXecutables*, `https://upx.github.io/`, [accessed May 16, 2021], 2020.

[43] Wikipedia, *Network-attached storage*, `https://en.wikipedia.org/wiki/Network-attached_storage`, [accessed May 16, 2021], 2021.

Appendix A

Code listings

Code listing A.1: Dependencies in the code base.

```python
import os
import pandas as pd
import pefile
import richheader
# doc:https://github.com/CIRCL/PyRichHeader/blob/master/richheader/richheader.py
import numpy as np
import time
from collections import Counter
from numpy import arange
```

Code listing A.2: Code that was used to extract the features.

```python

# Returns the features and their data in 2 lists. Feature / Data pairs are
# matched by their index in the lists
def get_header_data(header):
    # Iterates each header type related to a file and returns a filled list

    # Removes the structure tag (header type)
    del header['Structure']

    keys_dict = list(header.keys())
    values_dict = list(header.values())

    i = 0  # iterator

    # Initialize empty list of lists
    # len() of keys and values will always match
    # data = [[] for _ in range(len(keys_dict))]
    columns = []
    values = []

    # Fills a list with pairs of keys() and values()
    while i < len(keys_dict):
        # data[i].append(keys_dict[i])
        # data[i].append(values_dict[i].get('Value'))

        columns.append(keys_dict[i])
        values.append(values_dict[i].get('Value'))
        i += 1

    return columns, values

# Returns a item based on the directory name that is given
def handle_directory(dir_name):
    c_dir = f'data/{dir_name}/'  # directory where files are
    try:
        dir_files = os.listdir(c_dir)  # files in the directory
        idx, data, df = create_item(dir_files, c_dir)  # creates new item based on
                                                       # files in directory
    except:
        print(f'Error:␣Directory␣{dir_name}␣not␣found,␣exiting...')
        exit()

    return idx, data, df

# Iterates through a dataset and creates two lists. The first contains index values
# and the second contains data
def create_item(files, data_dir):
    pe_header_data = []
    cont = True  # variable to make sure that the columns are only added to
                 # pe_header_data once, important!
    files_dropped = []

    # Iterates each file in the directory
    for f in files:
        try:
            cur_file = FileAnalyzer(f, data_dir)

            # Build a list containing PE header data of all files being analyzed
            while cont:
                pe_header_idx = cur_file.header_data_columns  # append columns
                                                              # only once
                cont = False

            pe_header_data.append(cur_file.header_data_values)

        except:
            # Files that do not have a valid header will not be added to the
            # dataframe
            files_dropped.append(cur_file.file_name)
            continue

    if files_dropped:
        print('The␣following␣files␣have␣invalid␣PE␣headers␣and␣are␣not␣included
␣␣␣␣␣␣␣␣in␣the␣analysis:')
        print(files_dropped)
        print(f'#␣of␣files␣dropped:␣{len(files_dropped)}\n')

    df = fill_dataframe(pe_header_idx, pe_header_data)

    del pe_header_data[0]  # first element in list somehow are the indexes, remove.

    return pe_header_idx, pe_header_data, df
```

Code listing A.3: Code that was used in feature handling.

```
# Returns all features related to a file, including the PE header, Section names
# and Rich header
def feature_handling(idx, data, df):
    features = []
    tmp_section_list = []
    tmp_rich_list = []
    # average = True
    thresh_feature_freq = 0.01  # only include {index, value} pairs with a
                                # frequency above the given threshold

    for file in data:  # iterate each file
        i = 0
        for index in idx:  # iterate based on length of index, used to handle idx,
                           # data pair
            # Handles section names
            if index == 'section_names':
                for e in file[i]:
                    tmp_section_list.append(e)
            # Handles the rich header
            elif index == 'rich_header':
                for e in file[i]:
                    if type(e) == str:  # checks if the value is 'None'
                        tmp_rich_list.append(file[i])  # add the string
                    else:
                        tmp_rich_list.append(e[0])  # only takes compids, not how
                                                    # many times it occurs for that
                                                    # file
            i += 1

    # Count how many times a specific element occurs
    section_cnt = Counter(tmp_section_list)
    rich_cnt = Counter(tmp_rich_list)
    # Convert the Counter into a list of lists containing (string, # occurred)
    section_list = [list(i) for i in section_cnt.items()]
    rich_list = [list(i) for i in rich_cnt.items()]

    features = find_features(df, False)

    # Add the section names and rich header to the feature list
    features.append(section_list)
    features.append(rich_list)

    return features

# Returns a list of PE header features and their frequency in the dataset
def find_features(df, average):
    features = []
    for col_name, col_data in df.iteritems():

        if average:
            tmp = df[col_name].value_counts(normalize=True).nlargest(n=1)
        else:
            tmp = df[col_name].value_counts(normalize=True)

        for col_val, freq in tmp.items():
            # Add all features except section names and rich header - these are
            # handled independently
            if col_name != 'section_names' and col_name != 'rich_header':
                features.append([col_name, col_val, freq])
    return features
```

Code listing A.4: Class that was used to analyze a given file.

```
class FileAnalyzer:
    def __init__(self, file, data_dir):
        self.file_name = file
        self.file_loc = data_dir + file
        self._get_pe_header()
        self._get_rich_header()
        self._header_handler()

    # Initializes the PE header
    def _get_pe_header(self):
        # fast_load prevents parsing the directories, which are uninteresting for
        # analysis
        self.pe_header = pefile.PE(self.file_loc, fast_load=True)

    # Initializes the Rich header
    def _get_rich_header(self):
        try: # if rich header exists
            self.rich_header = richheader.RichHeader()
            self.rich_header.parse_path(self.file_loc)
        except: # else
            self.rich_header.valid_checksum = False
            self.rich_header.compids = None

    # Initializes the header data
    def _header_handler(self):
        f_dos_header = list(get_header_data(self.pe_header.DOS_HEADER.dump_dict()))
        f_nt_header = list(get_header_data(self.pe_header.NT_HEADERS.dump_dict()))
        f_file_header =
        list(get_header_data(self.pe_header.FILE_HEADER.dump_dict()))
        f_optional_header =
        list(get_header_data(self.pe_header.OPTIONAL_HEADER.dump_dict()))

        i = 0  # counter to remove items based on index

        # Iterates through each item in the optional header and looks for two
        # specific fields. If they are found they are dropped from both the
        # columns and values of the header. These are two fields that occur
        # occasionally for some executables, for simplicity in the implementation
        # I have chosen to drop these as trying to work around them has lead
        # to a lot of unnecessary time being wasted...
        for item in f_optional_header[0]:
            if item == 'Reserved1':
                f_optional_header[0].pop(i)  # columns
                f_optional_header[1].pop(i)  # values
            if item == 'BaseOfData':
                f_optional_header[0].pop(i)  # columns
                f_optional_header[1].pop(i)  # values
            i += 1

        section_names = []
        section_names_column = ['section_names']

        for section in self.pe_header.sections:
            section_names.append(section.Name)

        # Converts the section names to a str type so it can be added to the
        # dataframe
        str_section_names = str(section_names)
        # Removes spaces, additional formatting and the byte notation.
        # Returns a comma separated string.
        str_section_names = str_section_names.replace('\\x00', '')
        str_section_names = str_section_names.replace('␣', '')
        str_section_names = str_section_names.replace("b'", '')
        str_section_names = str_section_names.replace("'", '')
        str_section_names = str_section_names.replace("[", '')
        str_section_names = str_section_names.replace("]", '')

        section_names = [str_section_names.split(',')]

        # Concatenate lists to create two lists, representing the column indices
        # and their values
        self.header_data_columns = f_dos_header[0] + f_nt_header[0] +
        f_file_header[0] + f_optional_header[0] + section_names_column
        self.header_data_values = f_dos_header[1] + f_nt_header[1] +
        f_file_header[1] + f_optional_header[1] + section_names

        # Makes the row index available from the start of each list
        self.header_data_columns.insert(0, 'r_index')
        self.header_data_values.insert(0, self.file_name)

        # Starts adding rich header data to the list
        self.header_data_columns.append('rich_head_valid')
        self.header_data_values.append(self.rich_header.valid_checksum)

        self.header_data_columns.append('rich_header')
        if self.rich_header.valid_checksum:
            rich_header_data = []
            for k, v in self.rich_header.compids.items():
                tmp = [k, v]
                rich_header_data.append(tmp)
        else:
            rich_header_data = 'None'

        self.header_data_values.append(rich_header_data)
```

Code listing A.5: Code that was used to analyze the testing dataset.

```
def process_unknown_df(df, idx):

    m_counter = 0  # unknown files classed as malware
    b_counter = 0  # unknown files classed as benign

    analyzed_files = []  # fills list with information about files that have been
                         # analyzed

    # Iterate each file (row) in the dataframe
    for f_name, data in df.iterrows():

        random_count = 0

        index_count = 1  # used to iterate through the indexes of features. Start
                         # at 1 to skip r_index (file name)
        benign_occ = 0  # counts number of times a feature occurs in the benign
                        # dataset
        feature_app = 0  # number of features that are identified
```

```
unkn_features = []  # contains all the features of the current file
unkn_section_n = []  # contains section names of current file
unkn_rich_h = []  # list of lists containing rich header compids and their
                  # occurrence for the current file
section_n_found = []  # section names found in both benign base and current
                      # file
compid_found = []  # compids found in both benign base and current file
compid_not_found = []  # compids from current file not found in benign base
freq_dist_app = [0, 0, 0, 0, 0]  # stores the frequency "type" of the
                                 # features, 0-0.2, 0.2-0.4, 0.4-0.6,
                                 # 0.6-0.8, 0.8-1.0

for e in data:
    unkn_features.append([idx[index_count], e])
    index_count += 1

for feature in benign:
    for item in unkn_features:
        # if the column names and their values match i.e.
        # how many times the given column and value has appeared in benign
        # samples
        if item[0] == feature[0] and item[1] == feature[1]:
            benign_occ += feature[2]  # increases based on frequency of
                                      # value in benign dataset
            feature_app += 1  # increases for each feature that is
                              # identified
            for counter in arange(0.2, 1.1, 0.2):  # iterates to 1.1 in
                                                   # increments of 0.2
                if counter == 0.2 and (counter - 0.2) <= feature[2] <=
                counter:
                    # print(f’Weak strength for {feature[0]}, benign
                    # frequency {feature[2]}’)
                    freq_dist_app[0] += 1
                    break
                elif counter == 0.4 and (counter - 0.2) <= feature[2] <=
                counter:
                    # print(f’Weak/Med strength for {feature[0]}, benign
                    # frequency {feature[2]}’)
                    freq_dist_app[1] += 1
                    break
                elif counter == 0.6 and (counter - 0.2) <= feature[2] <=
                counter:
                    # print(f’Med strength for {feature[0]}, benign
                    # frequency {feature[2]}’)
                    freq_dist_app[2] += 1
                    break
                elif counter == 0.8 and (counter - 0.2) <= feature[2] <=
                counter:
                    # print(f’Med/High strength for {feature[0]},
                    # benign frequency {feature[2]}’)
                    freq_dist_app[3] += 1
                    break
                elif counter == 1.0 and (counter - 0.2) <= feature[2] <=
                counter:
                    # print(f’High strength for {feature[0]}, benign
                    # frequency {feature[2]}’)
                    freq_dist_app[4] += 1
                    break

for i in unkn_features:
```

```
            # Handles section names
            if i[0] == 'section_names':
                unkn_section_n = i[1]  # fills a list with section name features
            # Handles rich header
            elif i[0] == 'rich_header' and i[1] == 'None':
                unkn_rich_h = [['None']]  # ensures that None is returned as a list
                                          # not a string
            elif i[0] == 'rich_header':
                unkn_rich_h = i[1]  # fills a list with rich header features

        for ben_e in ben_section_n:
            for unkn_e in unkn_section_n:
                if unkn_e == ben_e[0]:  # if section name is found in benign and
                                        # current file
                    # print(f'Section name {mal_e}, frequency in benign set:
                    # {ben_e[1]}')
                    section_n_found.append(unkn_e)  # add to found

        # names that were not found
        section_n_not_found = np.setdiff1d(unkn_section_n, section_n_found)

        # print(f'Unknown section names found: {len(not_found)}')
        section_n_ratio = len(section_n_not_found) / (len(section_n_found) +
        len(section_n_not_found))
        # print(f'Unknown to Benign ratio: {section_n_ratio}')

        # Look for matching compids between benign set and incoming file
        for ben_compid in ben_rich_h:
            for unkn_compid in unkn_rich_h:
                if ben_compid[0] == unkn_compid[0]:  # if compids match
                    compid_found.append(unkn_compid)

        for i in unkn_rich_h:
            if compid_found:  # if any matching rich header compids are found
                for j in compid_found:  # iterate found list
                    if i[0] != j[0]:  # each feature that is not in the found list
                                      # is added to "not found"
                        compid_not_found.append(i)
            else:  # if no matches are found all compids are marked as "not found"
                compid_not_found = unkn_rich_h

        # Calculate the ratio of "not found" features
        compid_ratio = len(compid_not_found) / (len(compid_found) +
        len(compid_not_found))

        # print(f'Rich header compids also found in benign {compid_found}')
        # print(f'Unknown rich header compids found: {len(compid_not_found)}')
        # print(f'Unknown to Benign ratio: {compid_ratio}')

        # print(f'Feature distribution (weighted): {freq_dist_app}')

        f_class_obj = Classify(freq_dist_app, section_n_ratio, compid_ratio)

        f_class = f_class_obj.all_features(0, 0.5)
        class_header = f_class_obj.feature_freq()
        class_section = f_class_obj.section_names(0.5)
        class_compid = f_class_obj.compids(0.5)

        if f_class == 'Malicious':
            m_counter += 1
        else:
            b_counter += 1

        analyzed_files.append([f_name, freq_dist_app, compid_ratio, section_n_ratio,
                               class_header, class_compid, class_section, f_class])

    analyzed_files_idx = ['file_name', 'feature_frequency_base_vs_unknown',
                          'unknown_compid_ratio', 'unknown_section_n_ratio',
                          'class_header', 'class_compids',
                          'class_section_names', 'classification']

    return m_counter, b_counter, analyzed_files, analyzed_files_idx
```

Code listing A.6: Class that was used to classify samples.

```
class Classify:
    def __init__(self, freq_dist_app, section_n_ratio, compid_ratio):
        self._add_weighting_freq(freq_dist_app)
        # self.freq_dp = freq_dist_app
        self.section_nr = section_n_ratio
        self.compid_r = compid_ratio

    def _add_weighting_freq(self, freq_dp):
        i = 0

        while i < len(freq_dp):
            freq_dp[i] = freq_dp[i]/(i+1)
            i += 1
        self.freq_dp = freq_dp

    def feature_freq(self):
        # Only uses index 0 and 4 as there are very few entries in the indexes
        # between
        if self.freq_dp[0] > self.freq_dp[4]:
            return 'Malicious'
        else:
            return 'Benign'

    def section_names(self, t):
        if self.section_nr > t:
            return 'Malicious'
        else:
            return 'Benign'

    def compids(self, t):
        if self.compid_r > t:
            return 'Malicious'
        else:
            return 'Benign'

    def all_features(self, t_section, t_compid):
        if self.freq_dp[0] > self.freq_dp[4] and self.section_nr > t_section
        and self.compid_r > t_compid:
            return 'Malicious'
        else:
            return 'Benign'
```

Code listing A.7: Code that was used to provide information surrounding the datasets that currently were in use.

```
def analyze_files_in_dir(b_df, df, idx, dataset_name):

    print(f'Dataset:␣{dataset_name}')
    print(f'Data␣distribution:␣Malicious_NTNU␣({len(b_df)})␣and␣Malicious_NLL
␣␣␣␣({len(df)})')

    mal_counter, ben_counter, analyzed_f, analyzed_f_idx =
    process_unknown_df(df, idx)

    mal_class_rate = mal_counter / (mal_counter + ben_counter)
    ben_class_rate = 1 - mal_class_rate

    print(f'Malicious␣files:␣{mal_counter},␣Benign␣files␣{ben_counter}')
    print(f'Malicious␣classification␣rate:␣{mal_class_rate}')
    print(f'Benign␣classification␣rate:␣{ben_class_rate}\n')

    # analyzed_df = fill_dataframe(analyzed_f_idx, analyzed_f)
    # analyzed_df.to_csv(f'{dataset_name}.csv')
```

Code listing A.8: Code that was used to fill dataframes with the correct indexes and data.

```
# Creates a dataframe based on index values (features) and their data
def fill_dataframe(idx, data):
    # Creates a dataframe

    data.insert(0, idx)  # combines the lists, purpose: column creation

    df = pd.DataFrame(data, columns=idx)

    # Sets the filename as the row index
    df = df.set_index(idx[0])
    # Removes the row containing row index and column indexes after they have been
    # added (not needed)
    df = df.drop([idx[0]])
    # Sorts by filename
    df = df.sort_values(by=[idx[0]])

    # Stores dataframe as a CSV file
    df.to_csv('output.csv')

    return df
```

Code listing A.9: Example of how the main part of the code could look like.

```
if __name__ == '__main__':
    start = time.perf_counter()  # time at start

    benign_directory = '/samples/benign'
    malware_directory = '/samples/malicious'

    ntnu_ben_idx, ntnu_ben_data, ntnu_ben_df = handle_directory(benign_directory)
    ntnu_mal_idx, ntnu_mal_data, ntnu_mal_df = handle_directory(malware_directory)

    benign = feature_handling(ntnu_ben_idx, ntnu_ben_data, ntnu_ben_df)
    ben_section_n = []
    ben_rich_h = []
    keeper_tracker = True

    for feature in benign:
        # Start handling the section names and rich header - they are stored as
        # a list within the feature list
        if type(feature[0]) == list:  # first e is section names, second is
                                      # rich headers
            if keeper_tracker:
                ben_section_n = feature
                keeper_tracker = False
            else:
                ben_rich_h = feature

    analyze_files_in_dir(ntnu_ben_df, ntnu_mal_df, ntnu_mal_idx,
    'ntnu␣&␣norton␣lifelock')

    end = time.perf_counter()  # time when complete
    elapsed_time = end - start
    print(f'\nElapsed␣time:␣{elapsed_time}')
```

Appendix B

Complete feature list

The index of each sample was based on the file name as this would be the best way to identify the files. The PE header builds on all features from the headers that were specified in section 3.2. Section names is considered to be based on one feature: a list of section names. The rich header uses two features. A binary verification to indicate whether the file actually has a rich header, and a following list of compiler IDs if the rich header is valid.

Table B.1: Attributes that were gathered and used to represent each sample that was analyzed.

Index	PE header	Section names	Rich header
file name	e_magic	section names (list)	rich_header_valid
	e_cblp		comp_ids (list)
	e_cp		
	e_crlc		
	e_cparhdr		
	e_minalloc		
	e_maxalloc		
	e_ss		
	e_sp		
	e_csum		
	e_ip		
	e_cs		
	e_lfarlc		
	e_ovno		
	e_res		
	e_oemid		
	e_oeminfo		
	e_res2		
	e_lfanew		
	Signature		
	Machine		

Table B.1 continued from previous page

Index	PE header	Section names	Rich header
	NumberOfSections		
	TimeDateStamp		
	PointerToSymbolTable		
	NumberOfSymbols		
	SizeOfOptionalHeader		
	Characteristics		
	Magic		
	MajorLinkerVersion		
	MinorLinkerVersion		
	SizeOfCode		
	SizeOfInitializedData		
	SizeOfUninitializedData		
	AddressOfEntryPoint		
	BaseOfCode		
	ImageBase		
	SectionAlignment		
	FileAlignment		
	MajorOperatingSystemVersion		
	MinorOperatingSystemVersion		
	MajorImageVersion		
	MinorImageVersion		
	MajorSubsystemVersion		
	MinorSubsystemVersion		
	SizeOfImage		
	SizeOfHeaders		
	CheckSum		
	Subsystem		
	DllCharacteristics		
	SizeOfStackReserve		
	SizeOfStackCommit		
	SizeOfHeapReserve		
	SizeOfHeapCommit		
	LoaderFlags		
	NumberOfRvaAndSizes		