Master's thesis

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

Sturla Høgdahl Bae

Triage of PE-files through divide-and-conquer clustering

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2020

NTNU
Norwegian University of Science and Technology

Triage of PE-files through divide-and-conquer clustering

Sturla Høgdahl Bae

Abstract

The number of new, unique malicious files detected every day is steadily increasing. The reason why so many new files are detected, is not that so many new families of malware are created every day, but because minor modifications are made to existing malicious files. Changing a single bit is enough to make a file appear as an entirely new file, even though the behaviour of the file remains the same. As the number of new files grows, it can eventually become infeasible to analyse all new files in-depth. Worst case, this could lead to new variants of malicious files going undetected for a longer period.

Since a large share of the "new" files are mere variations of other files, in-depth analysis should not be necessary for all files. By analysing a single file in-depth, one can predict the label of all similar files. This way, the number of files in need of in-depth analysis would be reduced greatly.

A method that allows quick identification of similar files, is clustering based on static properties. Numerous features can be used in clustering. By combining features that complement each other, it can be possible to identify more files that are similar. It can therefore be wise to use several features. Some features allow clustering to be performed with quicker methods than others. How time-consuming and costly it is to cluster files, can therefore be determined by the choice of features.

In an attempt to increase the precision or reduce the execution time of clustering files, the divide-and-conquer algorithm could be applied. This involves clustering files differently, based on the properties of the files. Primarily the files should be clustered using features that allow quick clustering, and if a label cannot be determined by this, a new attempt can be made with features that are slow to cluster files by. Provided that using fast features will be enough in most cases, it will be possible to cluster files quicker than if all features were used on all files.

This thesis describes the proposed method, a proof of concept implemented to evaluate if the method has merit, a comparison between the proposed method and more basic clustering methods, and the challenges related to performing triage and evaluating result.

Sammendrag

Antallet nye, unike ondsinnede filer som oppdages hver dag øker stadig. Grunnen til at det oppdages så mange nye filer, er ikke at det lages så mange helt nye varianter av skadevare hver dag, men fordi små endringer blir gjort på eksisterende ondsinnede filer. Det kan være nok å endre én enkelt bit for å få en fil til å fremstå som en helt ny fil, samtidig som filen fortsatt vil utføre akkurat samme handling. Etter hvert som antallet nye filer vokser, kan det bli umulig å analysere alle nye filer i dybden. I verste fall, kan dette føre til at nye varianter av ondsinnede filer forblir uoppdaget over en lengre tidsperiode.

Etter som en stor andel av de "nye" filene bare er mindre variasjoner av andre filer, bør det ikke være nødvendig å analysere alle filer i dybden. Ved å analysere én enkelt fil i dybden, kan man forutsi hvilken familie av skadevare lignende filer tilhører, og merke disse filene basert på dette. Ved å gjøre dette, vil antallet filer som må analyseres i dybden reduseres i stor grad.

En metode som gjør det mulig å identifisere lignede filer raskt og effektivt, er gruppering av filer basert på statiske egenskaper. Mange egenskaper kan benyttes for å gruppere filer. Ved å kombinere egenskaper som utfyller hverandre, kan man potensielt finne flere filer som ligner på hverandre enn hvis man kun benytter én egenskap. Det kan derfor være nyttig å benytte flere egenskaper. Noen egenskaper gjør det mulig å gruppere filer på raskere måter enn andre. Hvilke egenskaper man benytter kan dermed avgjøre hvor tidkrevende og kostbart det vil være å gruppere filer.

I et forsøk på å øke presisjonen eller redusere tiden det tar å gruppere filer, kan man benytte splitt-og-hersk algoritmen. Dette innebærer å gruppere filer ulikt, basert på egenskapene til filene. Først og fremst grupperes filene ved hjelp av egenskaper som er raske å gruppere etter, og dersom man ikke klarer å identifisere en merkelapp for filer basert på dette, blir det gjort et nytt forsøk på å gruppere filene med egenskaper som er tregere å gruppere etter.

Denne rapporten beskriver den foreslåtte metoden, et konseptbevis som har blitt implementert for å evaluere om hvor nyttig metoden er, en sammenligning mellom den foreslåtte metoden og enklere metoder for å gruppere filer, samt utfordringene ved å prioritere filer og å evaluere resultater.

Acknowledgements

I would like to thank my supervisor, Geir Olav Dyrkolbotn, for his valuable guidance throughout the semester.

I would also like to express my deepest gratitude to my co-supervisor, Trygve Brox at NortonLifeLock. This project would likely not have been possible to carry out without his expertise in the field.

Finally, I would like to thank my friends and family for all the support I have received.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

This chapter first introduces the topics covered by this thesis. A description of the problem as well as the motivation to carry out this research projects is then given, followed by the research questions and contributions.

1.1 Topics covered

Malware is a term used to describe malicious software – software made with the intention of performing harmful activities on computers. The attack surface of malware has increased rapidly in the past decade, together with a significant rise in the number of people with access to a computer and the Internet, as well as the Internet of Things trend where more devices than ever before are connected to the Internet. The potential consequences that malware can cause, have also become more severe since many critical societal functions have become dependent on computers and computer networks.
To defend against malware, Anti-virus software is typically used. This software defends against malware by detecting and blocking or removing malicious files. On personal computers, anti-virus is strongly recommended since most people can be tricked into downloading and executing malicious files. For detection of this to be possible, the anti-virus software must have a method of identifying whether files are malicious or not.
Malware analysis is an activity typically performed by anti-virus software vendors, to identify new malware variants and new methods for detecting malicious files. If a malicious file is detected by anti-virus software, it can be stopped from performing harmful activities on the system. Authors of malware are therefore continuously trying to find new methods for avoiding detection of their malware, which is why anti-virus software vendors always have to be in search for new malicious files.
When the number of files in need of analysis increases beyond the number of files one can analyse in-depth, anti-virus software vendors must perform triage on the files. *Triage* involves sorting and prioritising items based on the perceived severity. This can be done, since it is critical that certain files are analysed immediately,

while there would be little to no consequence of postponing or not analysing other files.
One method that can help with triage is clustering. *Clustering* is the process of grouping items together based on how similar they are. A cluster of files should therefore contain files that are similar to each other. Triage can then involve prioritising files that are not similar to previously analysed files. The key to successful triage is therefore to use a suitable clustering algorithm and suitable features, which is the focus of this thesis.
The thesis also covers *unpacking*, which is a method for de-obfuscating files, in an attempt to cluster files more precisely.

1.2 Keywords

Malware triage, Static analysis, Divide-and-Conquer clustering, Semi-supervised learning, Static unpacking, Generic unpacking

1.3 Problem description

As mentioned in the beginning of this chapter, anti-virus software vendors must be able to find and analyse new files in order to identify new malware variants. Anti-virus software vendors can collect new files for analysis with various methods. One method is to utilise telemetry where the anti-virus software installed by users will submit unknown files for analysis. Another method for identifying new files is to subscribe to a feed from VirusTotal. This feed contains files submitted by users wishing to scan a file with multiple anti-virus applications, and can typically contain 1.2 million distinct new files (of any type) per day, of which at least 350 000 files are malicious executable files [1–3]. Anti-virus software vendors subscribed to this feed, can then retrieve the new files that they wish to add to their database of analysed files.
One method used by anti-virus software for identifying if an executable file is malicious, has been to check if the file matches any entries in a blacklist. These blacklists typically contains checksums of malicious files or byte signatures of specific file sections and must be updated regularly by the anti-virus software vendors. By making small modifications to the files, or developing the malicious files such that they modify themselves when propagating, the checksum of the file will change although the functionality remains the same. In addition to making small modifications to files, malware authors are also using packing as a technique to avoid detection. Packing involves compressing or encrypting a file, and then adding code that unpacks (decompresses / decrypts) the original file, before executing it. These methods used by malware authors to avoid detection, are the main reasons why there are so many new, unique files.
Anti-virus software vendors are facing a major challenge related to identifying all new malware variants as the number of new, unique files increase faster than

the computation cost decreases. Without any method to identify which files are likely to be interesting, files would have to be randomly selected or dropped. Doing so, could lead to new variants of malware remaining undetected. To mitigate this issue, triage must be performed to filter out files that are less interesting, and prioritising files that are likely to be interesting. Files that typically would be interesting are for instance new variants of malicious software that utilise new methods for avoiding detection.
Dynamic analysis is an effective method that can be used to extract features for identification of similar files. Dynamic analysis involves executing the file in a controlled environment and monitor the behaviour of the file. This is an effective method for clustering files, since two files that exhibit the same behaviour, likely are related. The problem with dynamic analysis, is that it is computationally expensive. A method that can be significantly cheaper in terms of computation is static analysis. Static analysis involves inspecting the properties of a file without executing it. This allows multiple files per second to be processed on a single CPU core, while dynamic analysis typically takes 2 to 5 minutes per file on a single CPU core [4]. For this reason, dynamic analysis should only be used in situations where static analysis is ineffective.
Multiple methods for scalable malware triage have been proposed previously, but no definite solution has been identified yet. Some methods are only able to filter out a small number of files, while other triage methods are computationally expensive themselves, significantly reducing the benefit of performing triage. After investigating some proposed methods, the following question arose: Why are they treating all files identically? Malware authors often try to obfuscate their malware heavily, so is there really "one features fits all" or "one clustering method fits all"? An alternative to treating all files identically would be to use the Divide-And-Conquer (DAC) algorithm. This algorithm involves breaking down a problem into smaller problems that are easier to handle. Is it possible to achieve greater speeds or greater precision, by attempting to use different methods for clustering different files?

1.4 Justification, motivation and benefits

Without improving the triage of executable files, new malware variants could remain undetected due to the cost of analysing all files rising to unacceptably high levels. This could result in new variants drowning in the amount of polymorphic and metamorphic files. The consequences of malicious files not being detected by anti-virus can be severe, as the "WannaCry" and "NotPetya" attacks have shown [5].
If a new method is identified that improves the performance of triage, it might be possible to detect new variants of malware faster, or reduce the cost of performing malware analysis in large scale.

1.5 Scope

To ensure that the project could be completed within the given time, the scope of the project had to be reduced. Although triage is relevant for all types of files, the majority of files submitted to anti-virus software vendors and VirusTotal are Portable Executable files ("exe" and "dll" files) [1], hence referred to as PE-files. The focus of this thesis was therefore on triage of PE-files, but findings can be applicable to triage of other file types as well.

1.6 Research questions

The research question of this thesis is defined as the following:

- How does divide-and-conquer clustering perform in triage of PE-files, compared to more naive clustering methods?
- How feasible it is to perform unpacking in large-scale triage of PE-files?
- To which extent does unpacking contribute to improved performance in triage of PE-files?

Improving malware triage involves reducing the need of analysing files using computationally expensive analysis methods such as dynamic or manual analysis or achieving greater speeds and accuracy of triage. It is likely not possible to achieve all these at once, but improving a single one of these measures could be a significant contribution.

1.7 Contributions

The main contribution of this thesis is a detailed description of the method, a proof of concept and an evaluation of the method based on results from experiments.
Additional contributions involve improved domain knowledge related to topics such as static unpacking, generic/dynamic unpacking, and the usefulness of various PE features.
The proof of concept is released as open-source software available for researchers wishing to verify or improve upon the implementation. This project has also resulted in other tools that have been released as open-source software. These open-source implementations are briefly described in appendix C, and can hopefully be of use for other researchers and malware analysts in the future.

1.8 Thesis outline

The thesis is divided into six chapters. Following this introduction, chapter 2, provides an overview of the concepts and previous findings / approaches to triage of PE-files and related topics. Chapter 3 then describes the proposed method and the proof of concept that has been implemented to evaluate if the method can be

used to improve large scale triage of PE-files. The results of the conducted experiments is presented in chapter 4, and then discussed in chapter 5. The thesis is finally concluded in chapter 6.

Chapter 2

Theory and related work

This chapter first covers some basic concepts related to PE-files as well as clustering and triage of PE-files. The chapter then covers previous findings and approaches to clustering and triage of malware and/or PE-files.

2.1 Concepts related to PE-files, clustering and triage

2.1.1 Portable Executable file

Portable Executable, typically abbreviated as "PE" is the name of the executable files for Windows operating systems [6].
A PE-file generally consists of a PE header and various sections. The PE header contains fields specifying basic properties of the file, such as the machine type the file was compiled for (e.g. x86), number of sections, a timestamp of when the file was created and the entry point address (address of where the execution of code should start) [6].
Each section consists of a section header and section data. A section header specifies the name of the section the virtual size of the section, the size of the raw data and flags specifying the characteristics of the section. Flags can typically indicate whether a section can be read or written to. A section should generally not be executable and writable, but there are exceptions to this.
It is possible to create arbitrary sections, but some sections are considered as special sections. Some of them are [6]:

- .bss: Uninitialized data
- .data: Initialized data
- .edata: Export tables; Contains information on exported functions
- .idata: Import tables; Contains information on imported functions
- .rdata: Read-only initialised data
- .reloc: Image relocations; Holds information that is required to find the correct addresses if a file could not be loaded at the preferred address, because something was already mapped to it.
- .rsrc: Resource directory; Typically contains icons and images that are re-

sources used by the file.
- .text: Executable code; Instructions that should be executed.

The .idata section contains a table known as the Import Address Table (IAT). Most PE-files import functions from libraries, and large programs can typically import hundreds of functions. The import address table contains function pointers that are used to look up the address of an imported function. When a PE-file is loaded, the address that should be pointed to, is inserted into the IAT [7].

2.1.2 PE-file features

Static analysis of PE-files is based on properties of PE-files that can be determined without executing the file. Some functionality can often be deduced by investigating the static properties of a PE-file. A PE-file importing the function "InternetOpenURL", will for instance likely connect to the Internet in order to upload or download data. For clustering files however, deducing the functionality of files is not important. Such analysis is rather performed after files have been clustered.
The features of different files are compared, and one can often assume that files exhibiting the same static features are related to each other. Some basic features that are commonly found in or easily calculated from the static properties of PE-files are [8]:

- Timestamp
- Number of sections
- Number of symbols
- Entry point address
- Section size (raw and virtual)
- Number of relocations
- Section entropy
- Resource attributes
    - Language
    - Resource size
    - Code page
    - Resource name

While some features can be directly extracted, such as timestamp, other features are derived. The entropy of a section must for instance be calculated in order to be used.
Certain features are optional and are not present on all PE-files. Icon is an example of such a feature. If a PE-file contains an icon in the resource directory, the icon will be displayed when a file is shown on the Windows desktop or in the file explorer.

2.1.3 Finding similar PE-files by comparing hashes

A *fuzzy hash* is a type of hash that allow identification of similar data. Unlike cryptographic hashes, a fuzzy hash will not change significantly due to small changes

in a file. Creating fuzzy hashes of two similar files should therefore result in similar hashes. To identify how similar the hashes (and thus files are), a distance or a similarity score can be calculated. Comparing two small hashes is much faster than comparing two complete files. Fuzzy hashes are therefore preferred when comparing many files against each other.
While a fuzzy hash is not directly a feature of a PE-file, it can be derived from a PE-file. It can therefore be considered as a derived feature.
A type of hash that is similar to fuzzy hashes, but not based on the raw data of the file, is perceptual hashes. Perceptual hashes are made to identify similar multimedia. They are typically based on how the multimedia is presented, for instance the RGB values of pixels in an image or a video. A perceptual hash can therefore identify similar images, even though the images have been compressed with different compression algorithms and thus shares no similarity in the raw data [9]. Perceptual hashes cannot be used to identify similar PE-files, but they can be used to identify similar icons.
Another type of non-cryptographic hashes allows identification of similar files by simply checking if the hashes are equal. To create these hashes, features that seem to remain identical across similar versions of files are used. Some of the proposed hashes based on this technique are described in section 2.2.2.

2.1.4 Obfuscation

To prevent malicious files from being detected as malicious by anti-virus software, malware authors use various obfuscation techniques. Obfuscation can involve modifying fields that do not affect the execution of the executable (e.g. timestamp), inserting redundant instructions that does not affect the overall functionality or hiding (compressing or encrypting) code that performs malicious actions.

Packing

Since performing manual obfuscation of files can be tedious, malware authors often use tools that compress or encrypt files as an obfuscation method. The process of compressing or encrypting an executable file is known as packing, and tools made to perform packing is known as packers.
After the original file has been compressed or encrypted, a piece of code known as an unpacking stub is added. The sole purpose of the unpacking stub is to decompress or decrypt the contained file and transfer execution to it. The resulting file will therefore only consist of the unpacking stub and seemingly random data [10].
Some packers copy properties from the contained PE-file onto the new, exposed header, while others do not. This varies from packer to packer. Unless properties of the contained PE-file are copied onto the new header, it is difficult to cluster files based on static features.
Packing is not only utilised to obfuscate malicious software, as it can also be used for benign software. PE-files can for instance be compressed in order to reduce

the size of a file or encrypted in order to make reverse engineering more difficult. One can therefore not assume that all packed files are malicious [11, p. 95].

Unpacking

When a packed file is executed, the entry point of the file is located in the unpacking stub. This means that the code in the unpacking stub is executed first when running the file. The unpacking stub first decompresses or decrypts the contained file. The unpacking stub then transfers the execution to the contained file. This involves executing the code at the entry point of the contained file, known as the Original Entry Point (OEP) [12].
This process of decompressing or decrypting a contained file and executing it, is referred to as unpacking. Executing a file in order to make it unpack itself, is known as dynamic or generic unpacking [13].
In certain cases, files can also be unpacked without executing them. By reverse engineering the functionality of a packer, it is possible to develop unpacking software that reverses the actions performed by the packer. This is known as static unpacking and is both safer and faster than executing potentially malicious files. The issue of static unpacking is that there are many more packers available than static unpackers, since reverse engineering is time-consuming and requires manual labour [13].

2.1.5 Clustering PE-files

As mentioned in the introduction, clustering involves grouping similar files. One option is to compare many of the individual properties present in the PE header or PE-sections, and another option is to use hashes based on larger portions of a file.
The challenge of grouping similar files is not simply finding files that have features in common, but rather to only group files that in fact are related to each other.
When clustering malicious files, an ideal cluster contains all the files belonging to a specific family of malware, and no other files. Bad clusters would contain malware belonging to different families, or possibly even a mix of malicious and benign files.

2.1.6 Finding similar files based on distance

To identify similar files based on distance, one must iterate over all files, and for each file calculate a distance to other files. If the distance is lower than a certain threshold, one can conclude that files are similar. Clustering based on similarity score is very similar to clustering on distance. If the similarity score between two files is high, this corresponds to a low distance.
Traditional centroid-based clustering methods such as K-means cannot be used for triage since this would not allow detection of new classes of malware. A threshold must be used to decide if files are sufficiently similar to each other. If unknown

files simply were added to the closest cluster, new classes of malware would be incorrectly labelled. Files should therefore not be added to a cluster unless the distance is lower than a given threshold. This is typically referred to as constrained / constraint-based clustering, which is a type of semi-supervised learning [14]. Finding suitable thresholds for identifying if files are similar enough, is a research topic by itself. In this thesis, thresholds proposed in previously published papers will therefore be used.
Since one of goals is to cluster quickly, and it is not vital that all items belonging to the same class are added to a single cluster, the quality of clusters does not have to be as high as if true clustering was performed. Some optimisations can therefore be implemented to reduce the time-complexity. By only comparing new files to the centroids of clusters and previously clustered files that are not part of any clusters, fewer than n comparisons are required for each new file. The exact number of comparisons required, will depend on how many clusters there are, and how many files have not been added to any cluster. If most files are added to clusters and clusters typically contain many files, the time-complexity of clustering can be significantly lower than $O(n^2)$.
With distance/similarity based fuzzy hashes, it can be difficult to identify a centroid as an arbitrary point that is in the true centre of the cluster. An option is to use the most central item in the cluster. One method of identifying the most central item is to use closeness centrality, commonly used in graphs to find the most important vertices. The closeness centrality of an item is equal to the mean distance from the item to all other items; $C(x) = \frac{1}{\sum_y d(x,y)}$ [15]. The item with the highest closeness centrality in a cluster, is suitable to use as centroid.
A potential trade-off that improves speed but likely degrades cluster quality, is to use the first element in a cluster as the centroid, and not update the centroid after adding new elements. This likely results in more clusters, but the clustering speed is increased.

2.1.7 Identifying similar files based on identical values

Finding similar files based on identical values is much faster than calculating similarity scores. This is not simply because comparing if two values are identical is faster than calculating a similarity score, but because it completely removes the need for iterating over files and checking if the values are equal.
To identify other files based on identical feature values, a two-dimensional array can be used to avoid having to compare all files. The first dimension should be an array that contains the feature values, for instance a hash digest, that are used as keys. The second dimension contains a second array. This array should contain references to files that are related to the given value/key. Figure 2.1 illustrates how this could look like for finding files with identical imphash digests, a feature described further in section 2.2.2.
Iterating over the whole array to find the correct imphash, would be inefficient when processing large amounts of data. Using a more suitable data structure, can

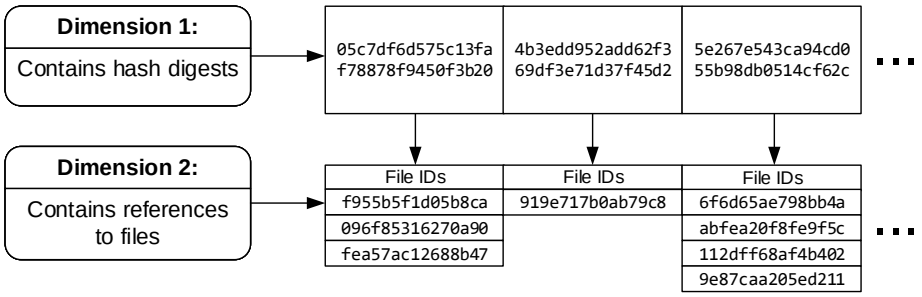


Figure 2.1: Two-dimensional array for finding files with identical hash digests.

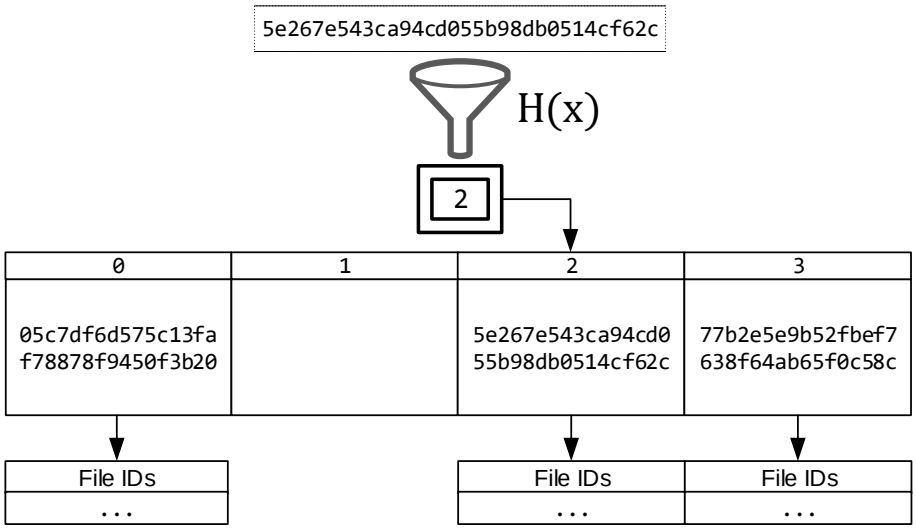


Figure 2.2: Using a hash table to quickly find files with a given "key".

increase the speed greatly. Two basic data structures that are suitable for such comparison, are hash tables and binary search trees.

A hash table is a data structure that allows keys to be mapped to a value. To find the correct index (offset in the array) for a specific key, a hash function is used as illustrated in figure 2.2. This figure illustrates the best-case scenario where a collision does not occur. The correct index is found immediately by using a hash function that converts the key into an index. This corresponds to a time-complexity of $O(1)$ for searching. The best-case time-complexity for inserting an item is also $O(1)$ [16].

The performance of using hash tables can be reduced if many collisions occur, or the hash table must be dynamically resized. Since it is very unlikely that a perfect, collision-free hash is identified, extra memory should be allocated to reduce the

likelihood of collisions.
Using binary search trees is an alternative to using hash tables. In binary search trees, all elements are sorted in a tree structure. Finding the correct element is therefore quick compared to iterating over all values in an array. Unlike hash-tables, there is no risk of collisions causing degraded performance, but the best-case time-complexity of performing a search is $O(\log n)$ [16].

2.1.8 Time complexity of clustering with different types of features

As previously mentioned, some features require a distance/similarity score to be calculated between all files, to identify if files are similar. These features will hence be referred to as "slow features", since the clustering requires comparison between many files and thus will be slow in a large data set.
Other features allow identification of similar files by finding identical values. These features will hence be referred to as "fast features", since the comparison will be fast even with very large data sets.
There are also some features with a time-complexity somewhere between these types of features, such as when calculating distances between icons. When trying to identify similar icons, a distance must be calculated between all the different icons. Preliminary research indicates that the number of unique icons is a lot smaller than the number of unique files; In the small data set of 9 823 files, 5 818 files (59.2%) had an icon, but there were only 1 628 unique icons. This means that only 16.6% of the files had a unique icon. The time-complexity of finding files with similar icons would therefore be much lower than $O(n^2)$. Such features will hence be referred to as "medium speed features".

2.1.9 Performing triage through clustering

Clustering of PE-files can be applied in triage. Triage is typically used in a setting where large amounts of previously analysed PE-files are available, and a continuous stream of new, unknown PE-files is coming in. The goal of the triage is to identify which PE-files should be prioritised for in-depth analysis or not. In-depth analysis could for instance involve dynamic analysis or manual analysis by a malware analyst. The files being prioritised, are not necessarily novel malware variants, but simply files where a label could not be determined through triage based on static analysis.
Figure 2.3 illustrates how the use of clustering can help reduce the number of files in need of detailed analysis: An unknown file arrives from a feed of unknown PE-files. An attempt is made to cluster the new file with the files that have been previously analysed. A file was matched, and they are grouped together in a new cluster. Since the previously analysed file had been labelled as "Ransom:Win32/WannaCrypt" and they were identified to be similar, it is assumed that the unknown file also should have the same label.

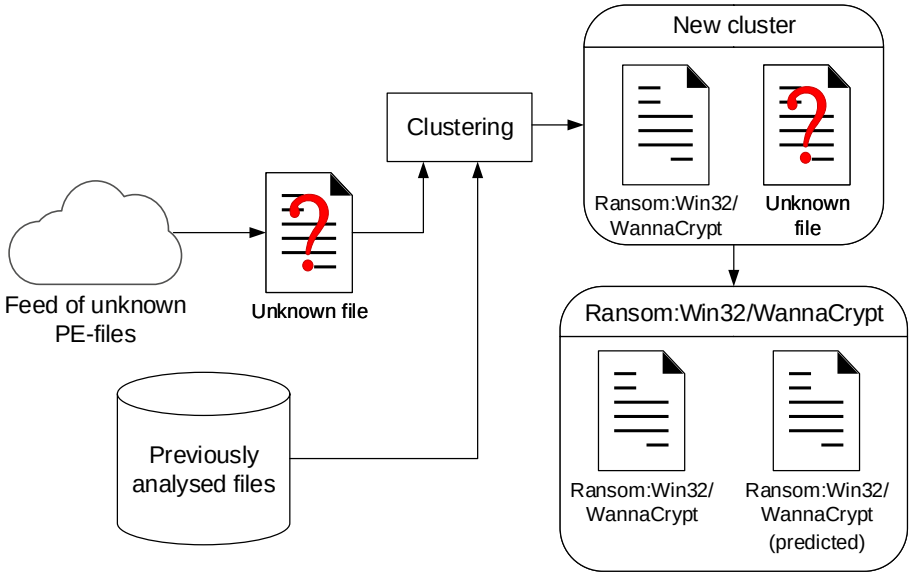


Figure 2.3: Triage of files: The label of an unknown file is predicted with clustering and transduction, and the file should therefore not be prioritised.

Labelling new files can also be done through transduction, similar to inductive reasoning, where files with predicted labels are used to further label new, unknown files. Unlike induction, transduction involves using newly labelled items in further labelling of subsequent unknown files. Figure 2.4 illustrates how this can affect labelling of unknown files. When using induction, one of the files are mislabelled as belonging to class C, even though the file is more similar to files that are classified as belonging to class A. The downside of using transduction is that any errors in labelling will propagate.

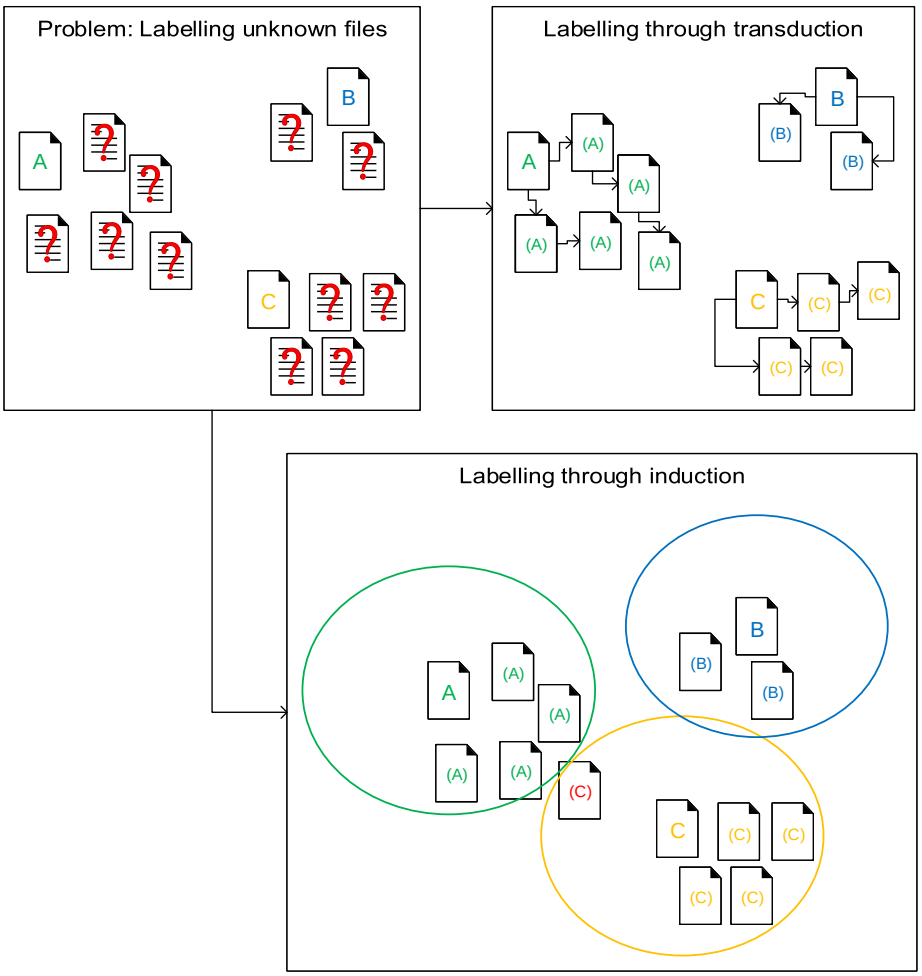


Figure 2.4: Illustration of the difference between using induction and transduction for labelling files. A file in class *A* is incorrectly labelled as belonging to class *C* when using induction.

The triage can further help with reducing the cost of analysing files in-depth, by allowing the analysis of representative files. When a representative file is analysed in-depth, one can assume that the results also apply to similar unknown files, as illustrated in figure 2.5. In this step one of this figure, similar files have been clustered together, but none of the files have been labelled yet. Since this cluster now is the cluster with the most unknown files, a representative file is sent to in-depth analysis, here represented by dynamic analysis in step two. Through dynamic analysis, file A is labelled as "Ransom:Win32/Petya". Since file A was labelled in step two, one can see that the cluster of files A, B and C now contains a labelled file. It is then possible to label tree files, by only analysing a single file in-depth.

The clusters containing the most unlabelled files, should be prioritised for in-depth analysis, since analysing a representative file in a large cluster, allows labelling of more files than analysing a representative file in a small cluster. If a cluster contains 20 unlabelled files, and analysing a single file in-depth is enough to label the other files in the cluster, the cost of labelling files is decreased by 95%, assuming that the cost of clustering files is negligible compared to the cost of in-depth analysis.

For such an approach to be effective, files being clustered together must be related. Malware authors are interested in making their files appear as benign executables, since it could allow malicious files to remain undetected by anti-virus software. The system must therefore verify the quality of clusters in a manner that makes it highly unlikely that files not being related to each other, are clustered together. This issue is discussed further in section 3.1.4 and 5.4.1.

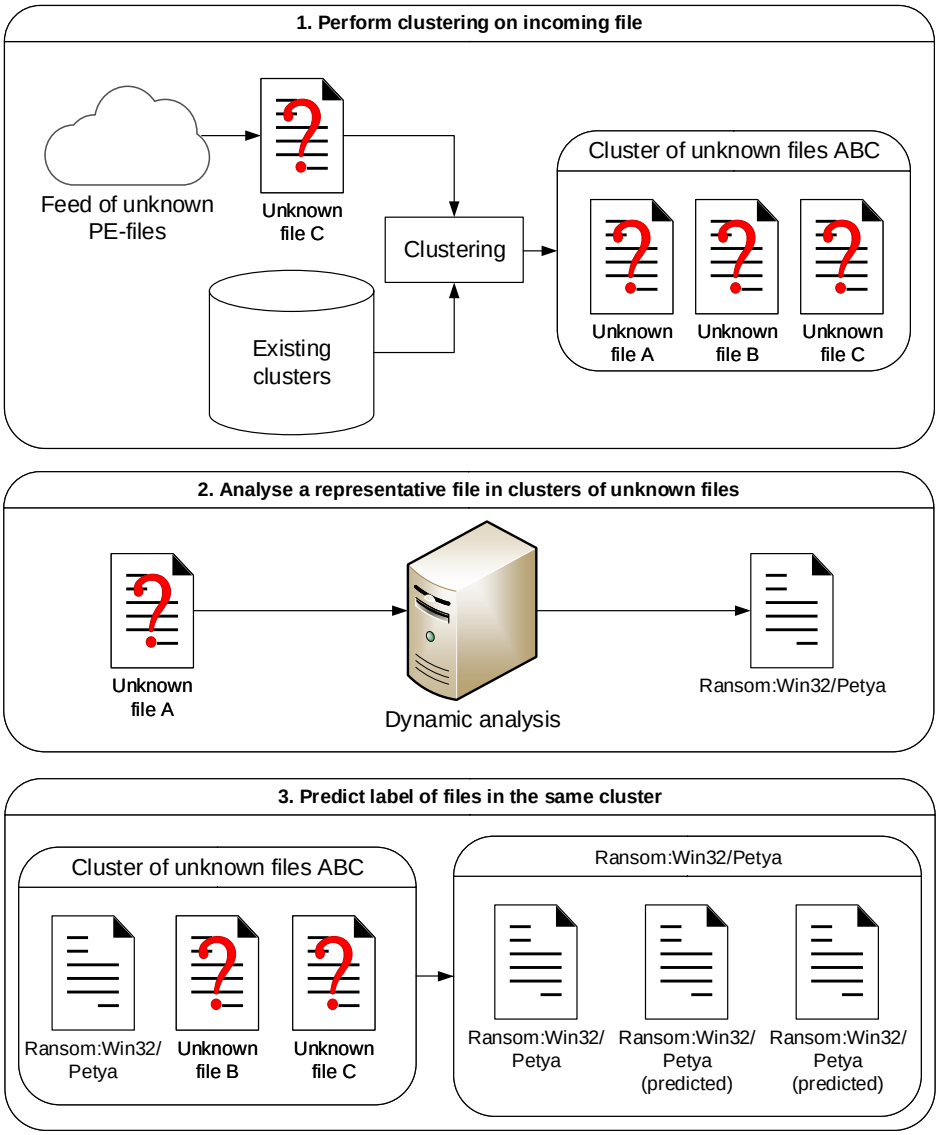


Figure 2.5: If many unknown files have been clustered together, it is possible to analyse a representative file in-depth, and predict the labels of the similar files.

2.2 Previous findings and approaches

Many papers have been published on methods for handling triage of PE-files. The papers are focusing on malicious files only, since most of the files that make triage necessary, seem to be malicious. This section covers some of the findings and approaches to both malware triage and general clustering of malware, that are relevant to the topic of this thesis.

2.2.1 Clustering files using distance/similarity based fuzzy hashes

Fuzzy hashes have been used to cluster malicious files for many years already. In 2007, Shadowserver Foundation published a paper [17] concluding that fuzzy hashes can be used to group similar malicious PE-files. They used *ssdeep* in their experiments and recommended that malware analysts provide fuzzy hashes of newly discovered malicious files together with identifiers such as cryptographic hashes. The only challenge mentioned in the paper was related to the use of packers [17]. This seems to be the first paper suggesting the use of *ssdeep* for identifying similar malicious PE-files. Since then, *ssdeep* has become the de facto industry standard fuzzy hash for PE-files. Popular services such as VirusTotal and JoeSandbox both provide *ssdeep* hashes for files uploaded to the services.
In recent years, papers criticising the findings of [17] have been published. An important paper was published by Pagani *et al.* in 2018 [18]. This paper concluded that *ssdeep* is less suitable for identifying similar binary files than other fuzzy hashes. While it is true that *ssdeep can* be used to group similar malicious PE-files, it is not as effective as the researchers claimed. The experiment in [17] was performed on a fairly small data set, with files belonging to only two different families of malware. Pagani *et al.* compared *ssdeep* to other fuzzy hashes such as *sdhash* and *TLSH*. Their findings suggest that *sdhash* is best suited for finding similarities between files made with different compiling options and TLSH is best suited for finding similarities between files where changes have been made to the source code. A noteworthy advantage of all these fuzzy hashes, was that they can be used for any file type, and not just PE-files [18].
Ssdeep produces a "context triggered piecewise hash" by breaking files up into smaller pieces, creating a small hash for each piece and joining the small hashes together to produce a hash for the whole file. *sdhash* (similarity digest hash) creates a hash that consists of a sequence of bloom filters. The bloom filters are created by identifying 64-byte sequences that have been empirically identified as being unlikely to encounter, and then hashing these unique 64-byte sequences and putting the hashes into bloom filters [19]. TLSH (Trend Micro Locality Sensitive Hash) creates a hash based on N-grams of bytes [18].
N-grams are sequences of N items. The word "hash" does for instance contain the 2-grams (bi-grams) "ha", "as", and "sh". Typically, the number of occurrences for each n-gram is counted, which then can be used to predict future sequences of items. For binary files, n-grams of bytes can be counted and then compared

to other binary files to find files with a similar number of occurrences for each n-gram.
For determining if two files are related, the authors of TLSH found that using a distance threshold of 100 when comparing malicious files, the detection rate was 94.5% while the false positive rate was 6.43% [20]. Independent research has also concluded that a threshold of 100 is suitable for identifying if files are similar [21].
As the number of malicious files have increased, one significant challenge of using *ssdeep* and other hashes based on calculating a similarity score between hashes has become apparent. To cluster files using these hashes, a score must be calculated between all the files. This results in a time complexity of $O(n^2)$ since each file must be compared to all other files in the data set to find files that are similar. With data sets containing many million files, finding similar files using distance-based hashes is time-consuming and becomes even slower as the data set grows due to the quadratic growth in the number of calculations required.
Even though *ssdeep* has been found to be less suitable for finding similar binary files, a potentially major advantage was presented by Wallace in 2015 [22]. The calculations required in order to calculate a distance between all files could be drastically reduced by first performing small calculations that determine if two hashes *could* match (having a similarity score of at least 1). The run time of clustering malicious files was therefore drastically reduced on data sets that were not homogeneous [22]. The average "cost" of comparing two hashes was reduced, but the number of comparisons needed for clustering files would still have quadratic growth.

2.2.2 Clustering files based on identical hash digests

Wicherski proposed a solution to this problem in 2009 [23], that was aimed specifically at clustering PE-files. The proposed hash was based on structural information of the PE header and data section and was named *peHash*. This hash was supposed to remain consistent, even though minor changes were made to PE-files. This allows files to have identical *peHash* values even though the files are not identical. The major benefit of this method was that one could find similar files without needing to calculate a distance between all the file hashes. One could simply group files based on identical hash digests instead [23]. Using hash tables or sorted binary trees, one can quickly find other files with identical hash values as described in section 2.1.7.
Other researchers have also attempted to identify potential features or hashes that remain identical for closely related malicious files. Researchers at Mandiant published a threat report in 2014 [24], describing a new feature that can be used to find closely related malware. They had discovered that the order of the imports in the IAT, often remains identical over multiple variants of the same family. It is also rarely identical for different families of malware, provided that the files do not have very few imports. An "imphash" is simply a hash of the ordered imports

in the IAT, and can be compared to the imphash of other files in order to find related files [24]. Independent research published in 2015 also suggested that imphash had good accuracy on clustering malware. The main downside of imphash was poor performance on packed malware, but this is common for features based on static analysis. [25]. In 2018, Chikapa and Namanya published a paper comparing *peHash* and imphash. The results from the experiments suggested that imphash had better accuracy as well as creating fewer clusters per family [26]. It would have been nice to know if there were specific situations where *peHash* performed better than imphash or vice versa, but the researchers did not investigate the results in detail. Namanya *et al.* published in 2016 a paper [27] where they had combined *peHash*, imphash, *ssdeep* hash of the full PE-files and *ssdeep* hash of file sections in a "Certainity Factor model Combinational Metric". The combined metric achieved even greater accuracy than using only one feature at a time, indicating that some features could complement each other. The researchers did not investigate how the clustering speed was affected, so it is not known how much the improved accuracy cost in terms of processing power.

The hashes and features described so far, have all been ineffective at finding packed files that are related to each other, but one feature is often copied from the contained file onto the resulting packed file – the icon displayed when viewing the file in the Windows file explorer or on the desktop. As mentioned by Silva *et al.* in a paper published in 2018 [28], one can greatly improve the accuracy of clustering PE-files by clustering on icons. Using icons as a feature alone is a bad idea, since malware authors often copy icons from benign executables. When icon is used together with other features however, it helps improve the accuracy of clustering malware greatly, since the same icon often is used for variants of the same malware [28]. File icon is also used by VirusTotal for their "Multi-similarity search" function. VirusTotal generates a *dhash* of the main icon for all PE-files with an icon [29]. This allows identification of other PE-files with identical or nearly identical icons, since *dhash* (difference hash) is a perceptual hash [30].

VirusTotal also uses imphash, *ssdeep* hash and Vhash for their multi-similarity search function [29]. Vhash is a proprietary hash developed by VirusTotal for clustering similar files. The only information that is publicly available is that Vhash is "an in-house similarity clustering algorithm value, based on a simple structural feature hash [that] allows you to find similar files" [31]. The exact performance of Vhash is unknown, but considering the amount of research and development that likely has been involved, it could be a valuable feature for clustering files retrieved from a VirusTotal feed.

Another potentially fast and accurate feature was described by Webster *et al.* in 2017 [32]. In their paper, the authors describe an undocumented part of the PE32 header created by Microsoft's linker "LINK.exe" since Visual Studio 97 SP3. This undocumented header is simply known as the *Rich header*, since it ends with the letters "Rich". The header was found to contain information on the build environment used to build the executable and a checksum used as a key for encrypting the header. Some packers copy the *Rich header* of contained files and use it in the new

header on the packed file, which could allow accurate clustering of packed files. To find related files, one could either find files with identical *Rich header* checksums, or use a distance/similarity based metric for comparing the values [32]. Since the *Rich header* is not vital for the execution of a PE-file, the header can be removed or modified by malware authors – and there are concrete examples of this being exploited to confuse malware analysts [33]. Researchers at ESET have attempted to use this feature for triage of files, and found that about 73% of all PE-files in the wild had a *Rich header*. Although the experiments on performing triage using the *Rich header* had good results, the researchers warned against relying on the *Rich header* alone, since it can be easily removed or modified [34]. Assuming verification is made to check that the *Rich header* is not malformed and the contents of the *Rich header* seems to match the features of the PE-file, the *Rich header* might be a valuable feature to check if two files are related.

2.2.3 Performing malware triage

A solution aimed specifically at triage of malware, was published by Jang *et al.* in 2011 [35]. Instead of focusing on creating a better hash for clustering malware, BitShred was a proposed method for malware triage, describing all the steps needed to perform triage that could scale to handle millions of files. The authors claimed BitShred was over 1000 times faster than existing approaches to malware clustering, but this claim was based on comparisons against approaches using dynamic analysis. The authors used a hash based on N-grams, and Jaccard similarity for calculating similarity scores between files, but stated that other distance/similarity based features could be used. Since the comparison had a time complexity of $O(n^2)$, measures were needed to allow the method to scale. The authors built a scalable solution that could split the feature extraction and comparison between multiple machines using Apache Hadoop and reported having an accuracy of over 90% when testing on non-packed malware. Although the solution could scale well horizontally, the resources required to analyse files would have quadratic growth due to the comparison method having a time complexity of $O(n^2)$. To mitigate the problem of packed malware, the authors stated that one could use "off-the-shelf unpackers", but did not go into detail on which unpackers one could use, nor what the success-rate of unpacking packed files would have been [35]. Similar claims related to unpacking files were stated in [36].
Since the malware found in the wild often is packed, the accuracy presented in [35] is not necessarily representative to the accuracy one would achieve in real malware triage. A method named "MutantX-S" was proposed by Hu *et al.* in 2013 to solve this issue [37]. MutantX-S combines generic unpacking with a hash of N-grams based on the opcodes of executables [37]. Opcodes (operation codes) are the operations that should be carried out by the CPU, e.g. "add". Together with one or more operands, e.g. "eax, 1", they form an instruction. By performing unpacking, it is possible to cluster files with a high level of accuracy even when handling packed malware – although with some limitations. Generic unpacking involves

unpacking files using a generic method that works for files packed with various packers. Specialised unpackers could be used to unpack files that are packed with certain packers as UPX or (Win)Upack, since there exists software that can unpack files packed by these packers. The problem of using specialised unpackers is that developing a new packer, often is easier than creating a corresponding unpacker. Specialised unpackers are therefore only available for some of the most common packers.

The authors of MutantX-S claimed that using specialised unpackers would be ineffective and costly, since it would require manual reverse engineering of all frequently used packers. They therefore proposed, and implemented, a generic unpacker. The unpacking was performed by executing the PE-files, in an environment where the addresses written to and instructions executed are logged. An unpacking stub will usually write to various addresses when the file is being decompressed / decrypted. When it has finished doing so, the file will begin executing code one of the addresses that was written to. The generic unpacking algorithm assumes that the OEP is located at that address and will then make a memory dump.

Although the memory dump does not contain the full, original PE-file (IAT is for instance not restored), the memory dump does contain the instructions of the original PE-file, assuming the unpacking was successful. This was the reason for using a hash based on opcode N-grams. The generic unpacking was tested by packing a malicious file using eight different packers, resulting in eight different files. The packed files were then unpacked using the generic unpacker, with a timeout of 1 minute. Files packed with *Armadillo* could not be unpacked, but files packed with the other seven packers could be successfully unpacked, with only a small difference in the N-grams compared to the original file. As with BitShred, the hashing trick was used to reduce the dimensionality of the feature vectors, which enable faster comparison between two files. The hashing trick resulted in a minor decrease in precision, but also a major decrease in running time and memory usage. When testing the ability of MutantX-S to predict labels of unknown malware, a data set of 40 000 files collected in a time span of 12 months was used. MutantX-S was able to achieve an accuracy between 0.7 and 0.8, when labelling new files using files identified in the last six months as the training set [37]. Unfortunately, the authors did not specify how many packed files there was in the data set or how the accuracy would have been if generic unpacking was not used. If a file had been packed twice, this method would likely be ineffective since it is assumed that the OEP is found when execution is transferred to the second unpacking stub. In this case, the instructions of the original executable would still be compressed / encrypted.

PinDemonium [38] is another attempt at performing generic unpacking. Unlike MutantX-S, PinDemonium could restore a fully functional PE-file when unpacking, but had additional overhead that required the unpacking process to have a timeout of five minutes. PinDemonium was evaluated by attempting to unpack files the authors had packed themselves, and by attempting to unpack files from VirusTotal

that had been identified as being packed. The success rate on unpacking malware found in the wild, was significantly lower than the success rate of unpacking files the authors had packed themselves [38].

Processing malware as an infinite length evolving data stream

The methods mentioned so far, would not have been very efficient to use for anti-virus software vendors. Most clustering techniques mentioned so far have been based on batch clustering. This involves clustering all files, or smaller portions of a data set as batches (groups of items). With a feed that continuously provides new, unknown files, such a solution would be less suitable.
Malware can be seen as an evolving data stream, where new malware constantly emerges, and old files become less relevant or might even experience a new uprising. For MutantX-S, the difference in accuracy between using historic data from the last 12 months and historic data from the last 6 months in the training set was minuscule, but only using data from the last 6 months reduced the space- and time complexity significantly. Several papers have described such evolving methods in a more detailed manner [39–41]. One of them was published by Masud *et al.* in 2008 [39], and described malware streams, such as the feeds processed by anti-virus software vendors, as evolving, infinite-length streams of data. For clustering infinite-length streams, traditional batch clustering methods become ineffective. In their experiments, they found significant concept drift in the data set. This means that the statistical properties of items change over time, making a trained model less accurate over time. It was assumed that this was caused by malware authors adapting to new defences, and the authors therefore suggested that old data is excluded from clustering as new data is added [39].
A paper basing itself on these findings was published by Ouellette *et al.* in 2013 [40]. They proposed a method for detecting malware using semi-supervised learning. The method had an initial step that involved batch clustering on a training set of labelled files to make the system able to identify malicious files. The second step involved real-time clustering of the files in the testing set containing unlabelled files. The files in the testing set were added to clusters iteratively, one at a time, and labelled if it was found to match an existing labelled file. In addition to labelling a file iteratively, the model is iteratively updated with the new files that are clustered. By combining newly labelled files with the labelled training data, the accuracy of the clustering was increased compared to using the training data only [40].

2.2.4 Divide-and-conquer clustering

Since malware authors constantly are adapting to new defences, and often heavily obfuscating malicious files, there is usually great difference between malicious files. How files within a class are similar, can therefore differ between classes. A clustering method designed to handle large data sets where files can be similar in different ways is *divide-and-conquer clustering*.

Khalilian *et al.* presented in 2009 a method that allowed more efficient clustering of vectors. The method combined the divide-and-conquer algorithm with k-means clustering and involved first dividing the data based on the size of the vectors, and then clustering vectors based on similarity. The problems of clustering the vectors was therefore split into several smaller problems related to clustering vectors of equal size. Finding similar vectors when all vectors are of roughly the same size was found to be easier than if vectors of all sizes were compared. The vectors could therefore be clustered with fewer iterations and higher accuracy [42].
To the best of our knowledge, divide-and-conquer clustering has neither been applied in clustering of malware nor triage of PE-files yet.

2.2.5 Evaluating malware triage / clustering

There are many challenges related to evaluating malware clustering, as described by Li *et al.* in a paper published in 2010 [43]. In the paper, the authors indicate that the results of many papers on malware clustering might not be as significant as one would first perceive. A major reason for this, is the use of data sets that do not represent malware found in the wild. An example given by the authors, was a research project that had generated a data set of malware based on labels given by different anti-virus software. By only choosing files where there was a broad agreement between multiple anti-virus software, the authors could be fairly certain that the labels were correct, while also being able to quickly create a data set. The issue was that this method can lead to artificially good results, since malicious files with a broad agreement between many anti-virus software potentially also are easier to cluster. Another issue of majority voting is the use of anti-virus software with a third-party anti-virus engine. For the majority voting to be fair, one must therefore make sure to only use independent anti-virus engines. To assess how significant the results of various research projects were, Li *et al.* performed an experiment similar to other methods for large scale malware clustering. The main difference was that Li *et al.* used plagiarism detectors to perform the clustering. Plagiarism detectors for software are made to detect how similar two executable files are and thus identify if plagiarism has occurred. Using plagiarism detectors, the researchers were able to achieve close to identical results as other research projects experimenting with algorithms custom made to cluster malware. Since the plagiarism detectors were not specifically made for clustering malware, the authors noted that the malware samples were not difficult to cluster. When investigating the results further, the researchers identified a likely reason for why the results are so good for both malware clustering algorithms and plagiarism detectors. The data sets had classes containing many files, which led to large clusters with low intra-cluster distances and higher inter-cluster distances. By reducing the number of elements in clusters with a high number of elements, the accuracy of all clustering algorithms dropped significantly. To ensure accurate results in research projects involving clustering of malware, it is suggested that data sets should be more representative of malware found in the wild. While there are several pub-

lic data sets with malicious files, there are no data sets that researchers agree on using. Using a standardised data set might not be a good idea anyway, since the malicious files in the wild constantly change, to combat new defences. A research project having great results on a standardised data set could therefore have poor results on malware found in the wild [43].

Measures for evaluating triage / clustering quality

As for how to evaluate the clustering of malware, there are no perfect solutions. There are however some measures that are common to use since they provide a fairly accurate representation of the results. Clusters can be evaluated using internal or external evaluation. Internal evaluation involves evaluating the intra- and inter-cluster distances, to assess the quality of the clusters [41]. The labelling of items by an expert, is as an alternative clustering method that represents a golden standard for labelling items. Comparing labels predicted by a model to labels given by an expert, is called external evaluation [44]. External evaluation is more suitable for evaluating clustering used for malware triage, since the goal often is to replicate the labelling performed by humans. When clustering based on identical hashes (for instance using *peHash* and imphash), it is not possible to calculate intra- and inter-cluster distances, which would make internal evaluation difficult.
When comparing labels during external evaluation, *true positives*, *true negatives*, *false positives* and *false negatives* can be calculated. These measures can be used to calculate the precision and recall, which seem to be the most common measures for evaluating the quality of clusters created for triage of malware [35, 37, 45, 46].
Figure 2.6 illustrates how the precision and recall is calculated. Precision is equal to the true positives divided by the sum of true positives and false positives. For labelling malware, this corresponds to dividing the correctly labelled files, by the sum of the correctly and incorrectly labelled files. The term precision is used, since the measure indicates how *precise* the method is when labelling files. The precision could be used to answer the question “how likely is it that a label is correct when using this method?”. Recall is calculated by dividing the true positives by the number of relevant items. For labelling malware, this corresponds to dividing the correctly labelled files by the number of files, assuming that all files should be given a label. The recall could be used to answer the question “how likely is it that a file is labelled, and the label is correct, when using this method?”.
Other measures used for measuring cluster quality are F_1-score and purity [21, 47].
F_1-score is a measure of accuracy that considers both the precision and recall, but with the possibility of weighting the recall value as more important than precision.
Purity is the ratio of how “pure” a cluster is, measured from 0 to 1 (0% to 100%). Within the field of clustering, purity is calculated by counting the number of items in the most common class in a cluster, and then dividing by the total number of

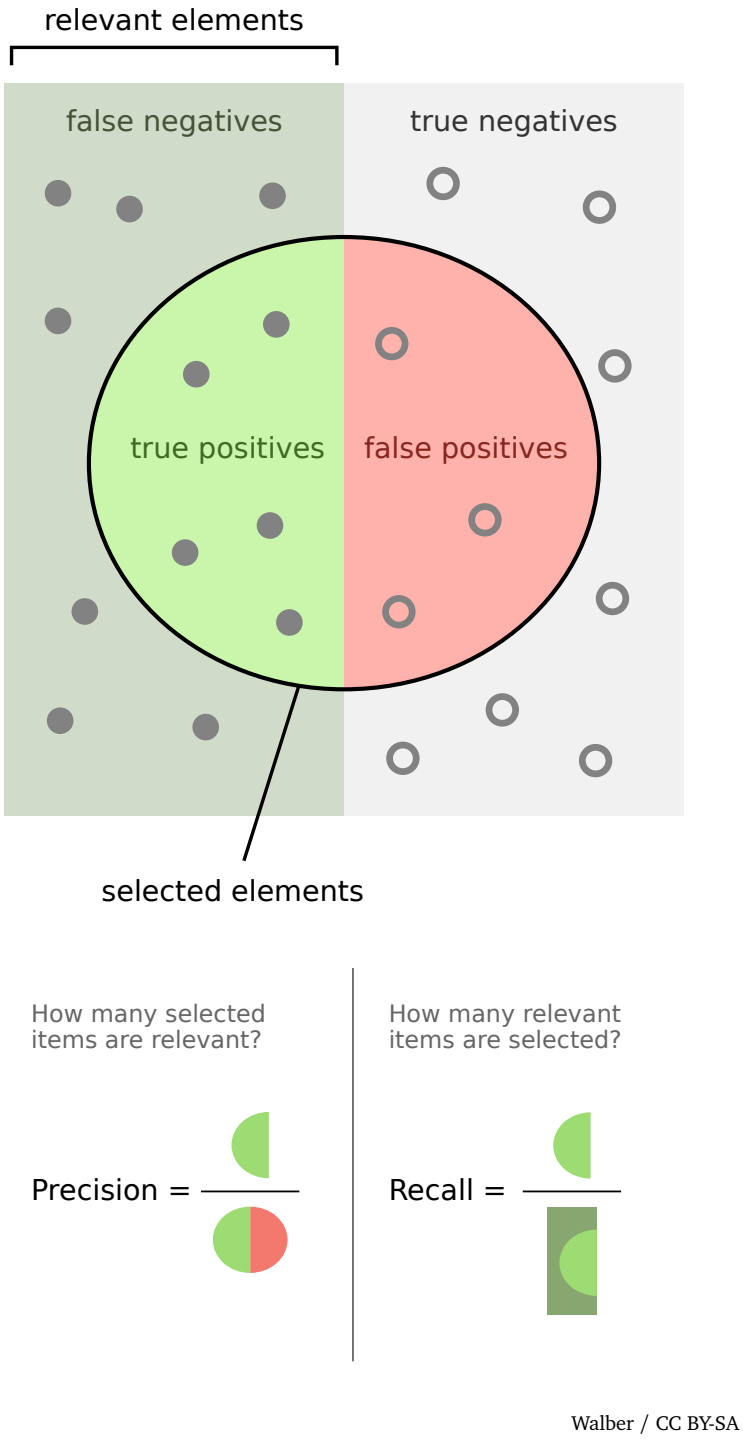


Figure 2.6: Illustration of how precision and recall is calculated

items in the cluster. One issue of using purity for evaluating clustering is that the purity is 100% if all clusters have a size of 1 – which is an issue since the point of clustering is to group items. This issue can be mitigated by also evaluating the size of clusters.
When using clustering for triage of files, the focus is in general slightly different from true clustering:

- When performing triage, it is not as common to label all the files in the testing set, since mislabelling files can lead to severe consequences. Unless one can label a file with a high degree of certainty, it is likely better to not give the file a label.
- It is fine to have a much greater number of clusters than the number of classes.
- The classes are usually imbalanced, since there are much more files belonging to certain families than others.

Due to these reasons, the recall is typically lower than when using true clustering methods such as k-means [37].

Validating results

When a data set is split into a training set and a testing set, it is possible that selection bias can occur; If the selection of items is not sufficiently random, the results could be biased due to how the data set was split. To avoid selection bias, it is common to use cross-validation, and one of the basic methods for performing cross-validation, is to use k-fold cross-validation [48]. Leave-One-Out Cross-Validation (LOOCV) is another basic method for performing cross-validation, but LOOCV is not feasible to use with very large data sets. K-fold cross validation involves randomly splitting the data set into K folds (splits) of roughly equal size and performing experiments K times. For each iteration, one fold is chosen as the testing set and the remaining folds are used as training set, as illustrated in figure 2.7. Typical values for K are 10 or 5 [49, pp. 181–182]. When performing K-fold cross-validation, experimental results are produced for each split. The final results are then typically given as the mean of these results and the 95% confidence interval [48]. The confidence interval is calculated from the standard deviation and is typically given with a value of 95% or 99%. A 95% confidence interval indicates that the “true” value is within the given interval with a probability of 95% [49].

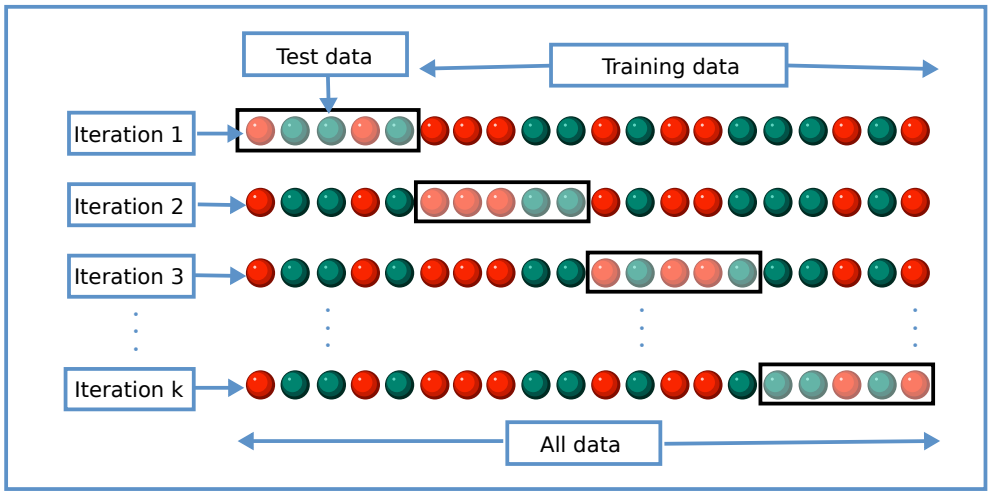


Figure 2.7: Splitting data for multiple iterations with k-fold cross-validation

Chapter 3

The proposed method

This chapter first describes the proposed method for how to apply the DAC algorithm in triage of PE-files. This is followed by a description of the proof of concept that was implemented in order to evaluate how the performance of triage is affected by applying the DAC algorithm.

3.1 Applying the DAC algorithm to triage of PE-files

Application of the DAC algorithm involves splitting the task of prioritising PE-files into smaller tasks. Under the assumption that different files exhibit different features, the features that will be ideal to cluster a file by with will vary.
The overall procedure is structured as a pipeline, where unknown files are coming in at the start of the pipeline. Progressing through the pipeline, a file is processed in the various steps required to perform the triage. Within each step in the pipeline, an attempt is made at processing the file in the most optimal way. Usage of the DAC algorithm is however mainly applied to the clustering step.

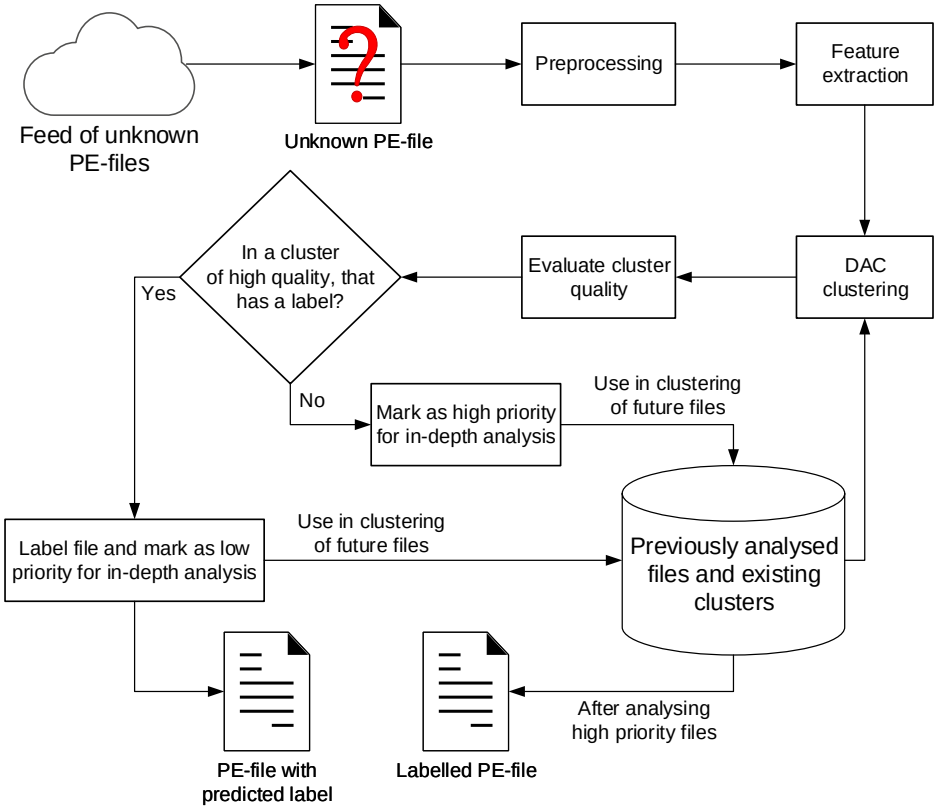


Figure 3.1: Illustration of the triage pipeline. Files are processed in the steps described in section 3.1.

The steps involved in the pipeline are illustrated in figure 3.1. These steps are described in more detailed under the subsequent sections, but the steps can be summarised with:

1. Preprocessing of files: De-obfuscation to allow extraction of correct features.
2. Feature extraction: Extract features that will be used to find similar files.
3. Evaluate cluster quality: Evaluate quality to avoid concluding that files are similar if multiple classes share a certain feature.
4. DAC clustering: Cluster each file with the features that are most suitable for the specific file.
5. Evaluate cluster quality: If the clusters were of poor quality or otherwise does not have a label, go back to DAC clustering and attempt to cluster the file with less suitable features.
6. Mark as low/high priority: Label the file if it is in a cluster was of high quality and the cluster has a label assigned to it. If multiple clusters are of high quality and have labels, use the label of the cluster that is of highest quality. A file that is given a label with this method can be considered as

being of low priority for in-depth analysis. A file that could not be labelled is marked as high priority.

Regardless of whether a file was labelled or not, it will be added to the database of previously analysed files and existing clusters. In-depth analysis will be performed when resources are available. A file in a cluster containing as many high-priority files as possible should then be prioritised.

3.1.1 Preprocessing

In the reviewed literature mentioned in section 2.2.3, there seems to be wide agreement that one should perform unpacking on files that are identified as being packed. This should be done to achieve better performance, since clustering packed files based on static analysis can be challenging.
The authors of [35] and [36] simply used a data set of non-packed files and argued that the files can be unpacked using "off-the-shelf unpackers", while [37, 38] proposed methods for how to unpack files.
Figure 3.2 illustrates the steps of unpacking a file. The first step of involves attempting to detect if a file is packed and which packer the file has been packed with. If the file seems packed, and it is likely that one will be able to unpack the file, attempt to unpack the file. If the file was successfully unpacked, send the contained file through the triage pipeline. The original file can then be labelled, based on any labels given to the contained file.

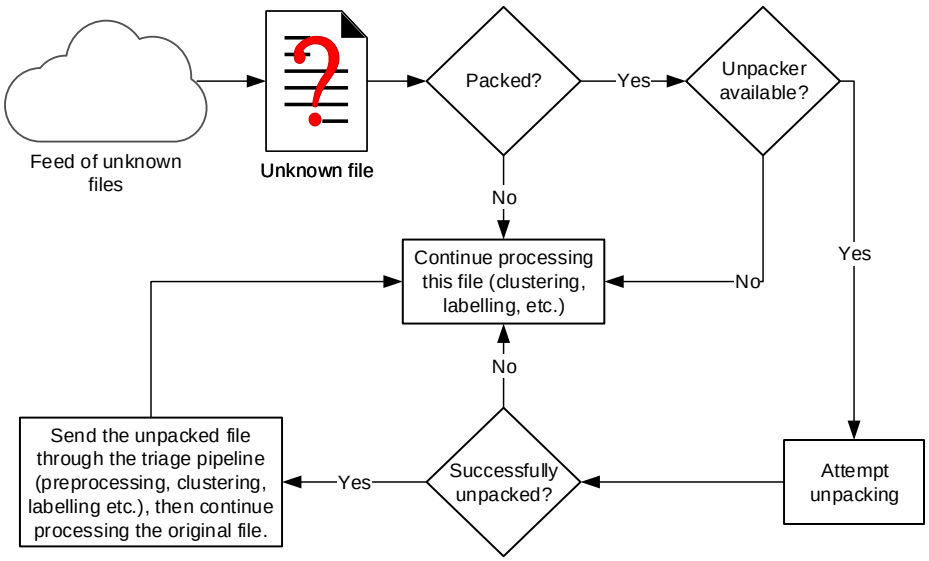


Figure 3.2: Preprocessing of PE-files. If a file is detected as being packed, an attempt is made at unpacking the file. If the file is successfully unpacked, the resulting unpacked file will be clustered first. The original file can then potentially be labelled based on any label given to the contained file.

3.1.2 Feature extraction

If new files were only supposed to be clustered together with previously analysed files that had been loaded from a database, it would be possible to minimise the computational cost of feature extraction, by only extracting enough features to cluster and label the new file. Processing evolving data streams however, this could become an issue. If only the bare minimum of features are extracted, it would be less likely that similar files will be matched when clustering future files.
An example of this would be if two files were retrieved from a feed of PE-file: *File A* and *File B*. *File A* is retrieved first and the imphash of the file is extracted. The file is then clustered by the imphash, a match is found and *File A* is given a label. *File B* is then retrieved and the imphash of the file is extracted. When attempting to cluster *File B* based on imphash, no match is found. The TLSH hash of *File B* is then extracted and an attempt is made to cluster the file. Since no TLSH hash was extracted from *File A*, it would not be possible to identify if *File A* and *File B* were similar, based on any features other than *imphash*. Since the hypothesis of this paper is based on the idea that there is no single feature that is perfect, and that some features can complement each other, all relevant features must therefore be extracted from all files.
This results in additional consumption of storage, memory, and processing power. The time-complexity of extracting features and the space-complexity of storing them is assumed to be $O(n)$, and the feature extraction is therefore assumed to scale well.

3.1.3 DAC clustering

This section describes how the DAC algorithm has been applied in clustering of PE-files. The resulting clustering technique is referred to Divide-And-Conquer clustering (DAC clustering), although the DAC algorithm most likely could have been applied differently to split the clustering into smaller tasks.
There is no definite solution for how the task of clustering PE-files ideally should be split into smaller tasks, but based on existing knowledge, some assumptions can be made. First, different files exhibit different features. Some features are common in all files and must be present, for a file to be considered as a PE-file. Other features however, are optional and are not always available, such as icon or *Rich header*. Secondly, the goal of clustering files for the purpose of triage, is to quickly cluster files in a manner that is as accurate as possible.
The first method used for dividing the clustering into smaller tasks, is therefore to cluster each file as accurately as possible based on the features that are available on each specific file. As [27] found, combining multiple features improves accuracy. If imphash, icon hash, *Rich hash* and TLSH hash can be extracted from a PE-file, the best accuracy would likely be achieved by using all of these features. If only a TLSH hash could be extracted however, the file should be clustered using this feature only. In this manner, all files will be clustered in a manner that ensures that the files will be clustered as accurately as possible.

Using all available features for clustering a file can increase accuracy, but will also require additional computation. A potential goal is therefore to achieve satisfactory accuracy, by clustering files with as many available features as necessary until such accuracy is reached. If satisfactory accuracy is achieved without clustering by all the features, the computational cost of clustering most files can be reduced. The reduction is most significant if the most expensive features are the ones not being used. This can allow computationally expensive features to be used when necessary to achieve sufficient accuracy of clustering, and not being used when they are not needed.
Using these optimisations based on the DAC algorithm, DAC clustering is proposed to perform the following when clustering a file:

1. Attempt to cluster the file with all fast features that could be extracted from the file.
2. Evaluate the quality of clusters with labels the file was added to, as described in section 3.1.4.
3. If no cluster with a label and sufficiently high quality was identified, repeat step 1. and 2. with the feature of highest priority, that has not yet been used.

The reasons why the clustering method uses all available fast features at once, is because it is necessary to allow future files to match with the current file by means of fast features. The cost of clustering with all the fast features is also negligible, provided that the number of fast features being used is not very high.
If a fast feature is present on a file, there is no way of knowing if the file can be matched with other files based on the feature, without attempting to cluster the file. Because of this as well as the tiny cost of clustering with fast features, any unsuccessful attempts at clustering a file with a fast feature can be seemed as a check for whether the fast feature can be used for clustering the file.
For files that are clustered with a slow feature, the overall cost of clustering the file will be higher than if the slow feature was the only feature used, since resources were used in attempting to cluster the file based on fast features. For divide-and-conquer clustering to have better performance than clustering based on slow features, most of the files must be successfully clustered using fast features. When most files are successfully clustered using fast features, the average cost of clustering files will be low.
Since it is of importance that most files can be successfully clustered with fast features only, it is important that the individual performance of features is measured, as well as the performance gained from combining different features. By measuring this, one can apply fast and accurate features first, and subsequently apply slower features that complement the fastest features, as shown in figure 3.3. In this figure one can see that most of the files are clustered with fast features. Some files were not successfully clustered, and a new attempt was made with medium speed features. It was then possible to cluster many of the remaining files. A similar procedure was repeated, until all files were clustered or there were no other features to cluster the files on.

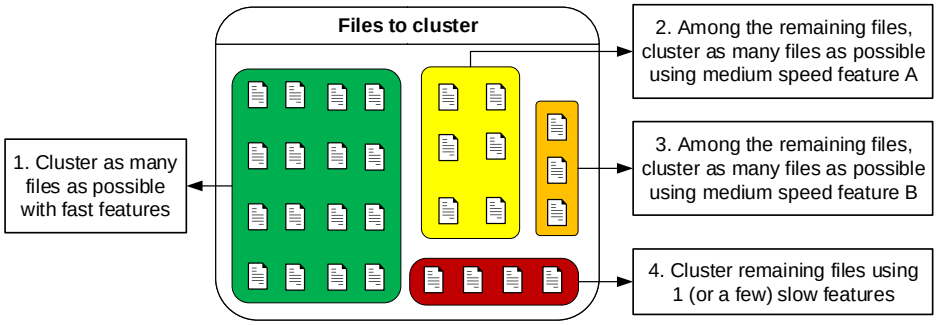


Figure 3.3: Clustering most files using fast features and applying slower features that complement the fast features on remaining files.

After a file is clustered, triage can be performed as described in section 2.1.9.

3.1.4 Evaluating cluster quality

Some features can be shared by files belonging to multiple classes. Files from multiple classes can for instance exhibit the same icon or imphash. An evaluation of the quality of clusters is therefore necessary if such features are used. If a cluster is evaluated to be of poor quality, one should not use the cluster to label any files added to the cluster.
A measure of cluster quality should typically represent how likely it is that a certain cluster only will contain items belonging to a single class. The following cases are given as examples of clusters exhibiting good / poor quality:

- If a cluster contains 20 files that all belong to the same class, a file that is added to this cluster in the future will most likely also belong to the same class. The quality is therefore assumed to be high.
- If a cluster contains 10 files belonging to class A and 9 files belonging to class B, it would be difficult to ascertain if an unknown file added to this cluster belongs to class A or class B. The quality of this cluster is therefore assumed to be low.

Purity, described in section 2.2.5, is a measure that can be suitable to evaluate the quality of clusters. The issue of using purity, is that the labels of the files have to be known in order to calculate the purity. When performing triage, one will most likely discover new clusters that do not contain any labelled files. A measure of cluster quality that uses internal evaluation, described in section 2.2.5, would therefore be required to evaluate the quality of all clusters.

3.1.5 Labelling files

If a file has been added to a cluster of high quality that also has a label assigned to it, the file can be labelled as described in section 2.1.9. If a file has been added to

multiple clusters with labels assigned to them, the label of the cluster exhibiting the highest quality should be used.

3.2 Proof of Concept

To evaluate the performance of the described method, comparisons of speed and accuracy between DAC clustering and naive clustering methods is needed. A proof of concept has therefore been implemented to perform these comparisons. DAC clustering could use an arbitrary number of features, and the proof of concept is only using a small subset of features. The findings can therefore not be that "DAC clustering performs better / worse than the other clustering methods". The findings would rather be that "DAC clustering *can* perform better / worse than the other clustering methods".
The proof of concept was implemented in Python since this allowed fast development. There are many valuable open-source Python modules / libraries available. These modules help with parsing PE-files and extracting relevant features / hashes. Software written in Python, is generally slower than software written in lower level programming languages such as C or C++, due to the extra overhead that is implicit in dynamic programming languages. Since this implementation was only intended to be a proof of concept, the simplified implementation and the availability of libraries was prioritised over raw performance. In production environments handling very large amounts of files, it would be wise to rather use lower level programming languages.
This section further describes how the proof of concept handles the steps involved in the triage. Unlike section 3.1, this section does not describe the theoretical method, but rather how the triage environment is simulated, and how the processing at each step in the pipeline has been implemented.

3.2.1 Simulating a malware triage environment with a feed of unknown PE-files

The proof of concept was implemented such that it can simulate the environment of an anti-virus software vendor, that needs to perform triage. In this environment, a large amount of previously analysed and labelled files is available. There is also a feed where new, unknown files continuously arrive. As stated in [39], this feed is likely to be evolving over time.
A common training set is fairly similar to a database of previously analysed files. An anti-virus software vendor will likely have analysed a huge number of files previously. To limit the complexity of clustering, it would likely be wise to reduce the number of files used as a training set. By using files that have been seen in various feeds within the last month, one can create a set that is likely to contain relevant files, while limiting the number of files to process. Since all these files are available at once, the proof of concept uses batch clustering on the training set. After files have been clustered, the quality of clusters are evaluated as described

in section 3.1.4 and clusters exhibiting high quality are labelled based on majority vote. This is possible since all files in the training set are supposed to represent previously analysed files, and therefore have a label assigned to them.
The feed of unknown files corresponds to a testing set where the labels of files are unknown, and the implementation should predict the labels of the files. As if the files were arriving continuously, the proof of concept will attempt to cluster and label the files in real-time, one file at a time. To allow the performance to increase as new files are added to clusters, the quality of clusters is re-evaluated and the labels are updated once a file in a cluster is given a label.

3.2.2 Preprocessing

The first step in the pipeline involves preprocessing. In the proof of concept, this is limited to unpacking only. In the preprocessing step, an attempt at unpacking the file will be made if a file is detected as being packed. After experimenting with many potential unpacking methods, it was concluded that static unpacking with ClamAV was the fastest method that could unpack a large portion of the packed PE-files. ClamAV is an open-source anti-virus engine that is made to detect malicious files, but it can also be used for unpacking PE-files. Developers of ClamAV have likely spent considerable time in implementing this functionality, and as of now it handles unpacking files that have been compressed/obfuscated by the following packers [50]:

- Aspack (2.12)
- UPX (all versions)
- FSG (1.3, 1.31, 1.33, 2.0)
- Petite (2.x)
- PeSpin (1.1)
- NsPack
- wwpack32 (1.20)
- MEW
- Upack
- Y0da Cryptor (1.3)

It would normally take considerable time to run ClamAV on a PE-file, but by replacing the signature database of ClamAV with an empty signature database, the processing time was reduced greatly. The processing time was reduced from approximately 15 seconds down to approximately 60-100 millisecond when unpacking files. The service updating the signature database also had to be disabled, to prevent the service from restoring the signature database.
When running the program *clamscan* on a packed PE-file with the arguments "–debug", "–leave-temps" and "–tempdir *destination_directory*", ClamAV will detect if a specified file is packed, attempt to unpack the file and leave any unpacked files in *destination_directory*. The unpacked files are not always PE-files, since the real payload might be a script or another type of file.
An example of this, is PE-files that simply execute the commands in a contained

script. AutoIt is an example of software that enables this, and has previously been used in creation of malicious PE-files [51]. The software is made to allow users create scripts for automating tasks. To execute the commands in an AutoIt script, the script must normally be executed by the AutoIt software. As an alternative to this, AutoIt provides the tool Aut2Exe, which can be used to create executable PE-files. Such a PE-file would then contain the script and all dependencies required to run the commands in the script when the PE-file is executed. The default behaviour of Aut2Exe is to pack the file with UPX after compilation [52].
After performing unpacking, the proof of concept will iterate over all the unpacked files. Unpacked PE-files will be sent through the triage pipeline and will therefore be recursively unpacked and analysed in the same manner as the original file. Unpacked files that are not PE-files, are instead treated as unpacked resources, a feature described in section 3.2.3. When all unpacked PE-files have been sent through the triage pipeline, the original file will move to the next step.
Appendix A describes other unpacking that was tested, but had to be abandoned due to poor performance.

3.2.3 Feature extraction

The implementation uses a limited set of features and has been limited to two steps. The first step combines multiple fast features, and the last step uses a single slow feature.
The fast features that were implemented in the proof of concept were:

- imphash (hash of import table)
- icon hash
- contained resource hash (hash of unpacked resources)
- Vhash (proprietary hash from VirusTotal, only used in the large data set)

Imphash, icon hash and Vhash is described in section 2.2.2. Contained resource hash was however not used based on previous works. As mentioned in section 3.2.2, some files that are unpacked are not PE-files. An attempt was therefore made at clustering files based on whether they contain similar resources (files that are not PE-files). For all contained resources, the SHA-256 digest of the file is computed and then used as a feature for clustering the file.
Clustering files by icon is also based on a SHA-256 digest of the data. By storing a digest instead of the raw icon, the storage/memory consumption is reduced. Comparing two hashes to identify if they are identical, also requires less computation than comparing two files to identify if they are equal.
The slow feature that was implemented in the proof of concept is TLSH, described in section 2.2.1. This is a catch-all derived feature, that can be extracted from all PE-files – even those that are corrupted and cannot be executed or parsed by the Python module *pefile*. *ssdeep* and *sdhash* could also have been used, but previous work suggests that TLSH can be more useful for finding similar PE-files.
As with unpacking methods, the use of certain features had to be abandoned. Appendix A describes the features that had to be abandoned due to poor performance

or proposed as future work due to the limited time available.

3.2.4 Improving performance through parallelization

Performing unpacking and feature extraction on files does not have to be done in the same order as the files arrived. Performing these actions on PE-files can be done completely independent and can therefore be parallelized fairly easily.
The proof of concept therefore uses Python's multiprocessing module to distribute the preprocessing and feature extraction. Multiple workers retrieve files that need unpacking and feature extraction from a "queue" of files. After a worker has finished processing file, the features are placed into a clustering queue. The clustering process will then retrieve the features of one file at a time from this queue, and cluster the file based on the extracted features.
This allows the unpacking and feature extraction to scale well across multiple machines. The current implementation only supports scaling across multiple cores on a single host, but can quite easily be modified to scale across multiple machines. Unpacking and feature extraction is easy to distribute, but as mentioned by the authors of BitShred [35], performing clustering in a distributed manner is more challenging. If the clustering was performed in a distributed manner, it might have been more difficult to measure the performance of different clustering methods also. The clustering is therefore performed by a single thread.

3.2.5 DAC clustering

After the features of a file has been extracted, the DAC clustering method described in section 3.1.3 is used to cluster the file. Unlike the figure illustrated in that section, the proof of concept does not attempt to cluster files with any medium speed features. Only a few fast features and a single slow feature is used. If the file is not added to a labelled cluster of high quality when attempting to cluster it with the fast features, an attempt is made at clustering the file with TLSH.
Details of how the proof of concept handles clustering files based on fast features and slow features are described in the next two sections.

Clustering using fast features

Clustering files based on identical values, could easily be implemented in Python. In Python, a "dictionary" is a data structure that can be used to map keys to their corresponding values. In other programming languages, this type of data structure is often referred to as associative arrays. The underlying implementation of Python dictionaries shares some similarities with hash tables, since a hash function is used to find the values based on the key [53].
The Python function defined in listing 3.1 is a slightly simplified version of the function used to cluster files on equal values in the proof of concept. In addition to being easy to implement, the clustering method also allows very fast retrieval of files with identical imphash.

Code listing 3.1: Clustering files based on equal imphash in Python. As one can see in the code below, clustering files based on equal values is simple to implement as well as fast to execute.

```
def cluster_with_imphash(sha256, imphash, imphash_clusters):
  """
  Cluster a file based on the imphash of the file

  Parameters:
  - sha256:     The SHA-256 digest of the file
  - imphash:          The imphash of the file
  - imhash_clusters:  A dictionary of imphash clusters,
                where the imphash values are used as keys
  """
  if imphash in imphash_clusters.keys():            # If cluster exists,
    imphash_clusters[imphash]['items'].add(sha256)  # add file to cluster.
  else:                                             # If cluster doesn't exist,
    imphash_clusters[imphash] = {                   # create new cluster.
      'label': None,
      'purity': 0,
      'items': set([sha256])                        # Add file to new cluster
    }
```

Clustering using slow features

As mentioned in section 2.1.6, clustering files based on distance, is slightly more difficult when many previously analysed files, have not been clustered using all features. A file that is being clustered with TLSH, is first compared to the centroid of all existing clusters. If the file was sufficiently close to any centroid, the file is added to the cluster with the closest centroid.
If the file is not sufficiently close to an existing centroid, a new cluster is created with the file as centroid. An iteration is then made over all files that have not been added to any TLSH cluster yet. If any other file is sufficiently close to the centroid of the new cluster, the file is added to the newly created cluster. Since some of these files might have a label, an attempt is made to label the new cluster.
As described in section 2.1.6, a method for reducing the time-complexity of clustering is to not update centroids after adding files to clusters. Preliminary research suggested that the benefit gained from updating the centroid after adding an item was low, and the default behaviour of the proof of concept is therefore to not update the centroids. An option to enable updating of centroids is however included in the implementation.
The Python function in appendix B illustrates how much more complex this is than clustering files based on equal values, and how many iterations have to be made, in order to cluster files with distance based fuzzy hashes.

3.2.6 Evaluating clusters

To evaluate the quality of clusters, the proof of concept uses the measure purity, described in section 2.2.5. It is assumed that a cluster is of poor quality if the purity is less than 0.5, or if the cluster contains five files or more and the purity is less than

0.8. These values were selected by manually inspecting a few clusters created with each of the features. A thorough scientific analysis of what the optimal thresholds are would be preferable. This would however require extensive testing and could therefore not be performed within the limited time available for the project.
As described in section 3.1.4, the purity of a cluster can only be determined if it contains labelled items. This measure is therefore mainly used at the end of the training, after all files in the training set have been clustered. Any clusters exhibiting low purity after training, will be marked as a cluster of poor quality.
After clustering unknown files, some files will be clustered together in new clusters with unknown quality. In lack of a suitable method for evaluating these clusters, the proof of concept assumes that all these clusters are of high quality. The consequences of doing so, are discussed under section 5.4.

3.2.7 Labelling clusters and files

At the end of the training phase clusters are labelled. The mentioned evaluation of cluster quality is used to identify if a cluster should be given a label, and majority vote is used to identify which label should be used. If the quality of a cluster is evaluated to be good, the label of the most common family in the cluster is used to label the cluster. If a cluster is of poor quality, no label is given to the cluster.
For labelling a file, all clusters the file is part of, are inspected. Among clusters that have a label, the cluster with the highest purity is chosen. The label of this chosen cluster is then used to label the file. If the file cannot be labelled, but it has been unpacked to another PE-file which has a label, the label of the unpacked PE-file is used to label the original file.

3.2.8 Prioritising files for in-depth analysis

The triage is not only supposed to filter out files that are similar to previously seen files, but also help maximising the benefit gained from performing in-depth analysis. It should therefore be possible to identify files that should be prioritised for in-depth analysis. Analysing a file in a cluster that contains many unlabelled files should be prioritised, since this allows prediction of the labels for as many files as possible.
For clusters of good quality, the first item in the cluster is sent to a simulated in-depth analysis. The simulated in-depth analysis is simply a function that returns the correct label of the file. The label of the file is then given to the cluster, as well as all other unlabelled items in the cluster.

3.3 Experiments

The proof of concept has been implemented to allow several different clustering methods to be used. The results of clustering with different methods can then be compared against each other. The compared methods are:

- Using a single feature for clustering all files
- Using all fast features
- Using all features
- Divide-And-Conquer

The main goal of the experiments is to compare the speed and accuracy of the DAC clustering described in chapter 3 to more naive methods, and evaluate how the application of the DAC algorithm affects the performance.
Although using all features or only fast features are seen as more naive methods, the divide-and-conquer algorithm has been applied to some extent; Some features are only used on some files, which results in some files being clustered differently than others. The truly naive methods are the ones where a single feature is used to cluster all files. The difference between clustering with "all features" and DAC clustering, is therefore that the proposed DAC clustering method will not cluster a file with slower features, if the file has been successfully clustered with fast features.
The features used in the experiments are mentioned in section 3.2.3.

3.3.1 Data sets

Two data sets were used for the experiments. The first data set was a rather small data set, consisting of 10 families with a total of 9 823 files. The second data set is a quite large data set of 93 families with a total of 232 301 files.
The reason for using two data sets of different sizes, is to test how the size of the data set affects the results. The number of computations required for clustering files with slow features grows faster than the number of items in a data set, and the findings can therefore be different for data sets of different sizes.
The small set is a subset of a data set used in previous research projects, at the Testimon Research group at NTNU. It consists of both packed and non-packed PE32 files (PE-files compiled for 32-bit processors). It was collected in Q3 2015 from maltrieve, VirusShare, VxHeaven and files shared by students. According to the paper describing the data set, the files were labelled using anti-virus reports retrieved from VirusTotal, but it is not mentioned whether majority voting was used or not [54, 55]. This data set will hence be referred to as the *small data set*, and details of this set is listed in table 3.1.
The large data set was collected in March 2020 by subscribing to a VirusTotal feed. Files were labelled based on majority votes among the anti-virus results in VirusTotal, and unfortunately it was not considered that some anti-virus software vendors, use the same anti-virus engines. Files were collected based on family name and Vhash, but only files where family name could be determined, were used in experiments, since validation otherwise would be difficult to perform. More specifically, the method was:

1. Collect all PE-files in the feed that have not already been collected.
2. Determine family name based on anti-virus results. Store family name and Vhash in a database.

3. If there are more than 10 000 files of a specific family or with a specific Vhash, do not collect more files of that family or with that Vhash.
4. After finishing collection, group by family and Vhash. Store files that are in clusters of at least 100 files.
5. Exclude any files without a family name.

This data set will hence be referred to as the *large data set*, and details of this set is listed in table 3.2. Notice that there are 13 438 files in the *agent* family. This means that some files were included in the data set only due to having identical Vhash as at least 100 other files. The results of clustering files based on Vhash are therefore likely not fully representative of how Vhash would perform in clustering of other PE-files.
In an ideal situation, the large data set should have been collected over a longer period. It should nevertheless be more representative for typical files found in the wild today, than the files in the small data set. The data set exhibits more realistic imbalance related to the number of files belonging to each family, and since it is much bigger, the challenges related to scalability becomes more evident.
The labelling process is not perfect, and a few files have been verified to carry the wrong label. Due to the amount of data required for testing large scale triage, no better solution was found for labelling the files. Most files seem to be labelled correctly, and the results should therefore not be noticeably affected.
In addition to showing the families of malware and the number of samples in each family, the tables show the number of packed samples within each family. This was identified by using Detect It Easy ver. 2.05. This tool only detects files that are packed with known packers, and the tool might report false positives. The given numbers are therefore only approximate values.

Table 3.1: Details of the small data set

Family	Non-packed files	Packed files	Total
agent	478	522	1000
hupigon	97	903	1000
obfuscator	113	887	1000
onlinegames	921	79	1000
renos	657	343	1000
small	139	861	1000
vb	482	581	1000
vbinject	635	365	1000
vundo	609	214	823
zlob	821	179	1000
Sum	**4952 (50.4%)**	**4934 (49.6%)**	**9823**

Table 3.2: Details of the large data set

Family	Non-packed files	Packed files	Total
adposhel	1656	0	1656
agent	9121	4317	13438
allaple	5021	0	5021
alman	166	14	180
bancteian	912	0	912
banker	281	29	310
benjamin	2	8751	8753
bifrose	141	55	196
bladabindi	1313	55	1368
blocker	134	14	148
brontok	36	165	201
coinminer	10651	6	10657
cosmu	15	236	251
crypt	268	5	273
darkkomet	198	34	232
detroie	107	75	182
dialer	39	568	607
dinwod	73	3672	3745
eggnog	279	378	657
emotet	2709	87	2796
expiro	1488	1	1489
farfli	113	51	164
fasong	280	327	607
fearso	141	203	344
floxif	2472	286	2758
fsysna	3299	0	3299
fugrafa	170	0	170
gandcrab	6333	0	6333
glupteba	404	0	404
hematite	9989	0	9989
hupigon	140	231	371
ipamor	2	1281	1283
ircbot	247	409	656
jeefo	410	0	410
juched	131	2	133
keylogger	109	13	122
koutodoor	109	45	154
lamer	197	4861	5058
locky	105	2	107

Table 3.2 continued from previous page

Family	Non-Packed files	Packed files	Total
lolbot	1150	0	1150
lunam	2402	3244	5646
mabezat	172	4	176
madangel	104	5	109
mansabo	4910	0	4910
mydoom	237	3601	3838
neshta	4404	12	4416
nimnul	228	43	271
nitol	1255	0	1255
nymaim	216	0	216
padodor	9137	0	9137
pakes	612	1	613
parite	996	90	1086
picsys	1937	2551	4488
pluto	1149	605	1754
qqpass	37	83	120
qukart	4279	0	4279
ramnit	2127	322	2449
renamer	434	0	434
ribaj	1174	0	1174
rozena	129	0	129
runouce	1120	78	1198
sality	4998	680	5678
shipup	1533	384	1917
shodi	25	546	571
sivis	4665	0	4665
skybag	100	0	100
small	6101	426	6527
softcnapp	4492	0	4492
soltern	5250	0	5250
starter	224	0	224
startpage	4558	21	4579
stormattack	1330	0	1330
swisyn	658	111	769
swrort	387	9	396
sytro	7328	1854	9182
tinba	91	63	154
toffus	256	0	256
tofsee	178	0	178
unruy	9353	102	9455

Table 3.2 continued from previous page

Family	Non-Packed files	Packed files	Total
upatre	3290	1506	4796
urelas	2316	673	2989
ursnif	316	1	317
vbkrypt	133	29	162
viking	72	441	513
vilsel	406	232	638
virlock	9986	0	9986
virut	8868	1120	9988
vobfus	570	16	586
vundo	1921	0	1921
wabot	9832	73	9905
warezov	247	1	248
xorist	113	16	129
zegost	76	42	118
Sum	**187143 (80.6%)**	**45158 (19.4%)**	**232301**

3.3.2 Hardware setup

The experiments with the proof of concept, were performed on a virtual machine in a local OpenStack cloud computing platform. In shared cloud platform environments, overcommitting is often used. This allows multiple virtual CPU-cores to be created per physical CPU-core, since it is unlikely that all users will be using all available processing power at all times. Ideally the experiments should have been performed on a dedicated system, but since cross-validation is used, it is unlikely that disturbance caused by other users will occur, without such disturbances being clearly visible in the results.
The virtual machine had 16 virtual CPU cores with the model name “Westmere E56xx/L56xx/X56xx (Nehalem-C)” and 32 GB of RAM, but the exact hardware should not matter. It is rarely possible to correctly compare the speed of different research projects, as even the same CPU can perform differently if the microcode is updated. To allow comparison of the method against more naive methods, the comparisons are therefore made with the same software, on the same hardware. Additionally, the results can be validated by downloading the open-source proof of concept and performing the same comparisons on new hardware. Although the exact timings likely will differ, how fast the different methods are relative to each other, should not change.

Chapter 4

Results

This chapter presents the results from experiments performed with the proof of concept. Section 4.1 first presents the results from performing the experiments on the files in the small data set. This is followed by results from experiments on the large data set in section 4.2.
Results from experiments performed with k-fold cross-validation are given as the mean value, together with the 95% confidence interval. For all experiments where k-fold cross-validation was used, k was equal to 5, meaning that the experiments were conducted five times.

4.1 Small data set

The small data set containing 9 823 files was used in all experiments except those that involved Vhash. The results are presented according to the steps in the triage pipeline described in section 3.1. In each of the steps, the performance of different clustering methods is compared. A final comparison that measures the overall effectiveness of the triage when using the different clustering methods is included at the end.

4.1.1 Training

During training, the feature extraction and clustering is performed in separate steps. This ensures that the speed of extraction- and clustering speed can be investigated independently.

Preprocessing (Unpacking)

By unpacking files in the small data set with 9 823 files, of which 4 934 files (49.6%) have been identified as being packed, the proof of concept was able to achieve the following:

- 1 350 files (27.4% of the packed files) could be unpacked to at least one file.

- 1 204 files (24.4% of the packed files) could be unpacked to at least one PE-file that was not identified as being packed. This includes both files that were unpacked directly to a non-packed file and files that were indirectly unpacked to a non-packed file, through recursive unpacking.
- 4 208 PE-files were unpacked from the files in the data set. Due to this, the proof of concept had to process 14 031 files instead of 9 823. This is an increase of 42.8%.
- Performing unpacking on files, did not lead to a significant increase in processing time by itself. Attempting to unpack the 4 208 files that had been identified as being packed, resulted in an 11.5% increase in processing time. A large increase in processing time was however caused by parsing and recursively unpacking the unpacked files.

Feature extraction

The mean time required for parsing and extracting features from PE-files is shown in table 4.1. The execution time is given in milliseconds per file, on a single CPU core. When denoting the execution time per file as t, the number of files as n and the number of CPU-cores as c, the total execution time required to extract features from a set of files, is approximately $\frac{t \cdot n}{c}$.

Table 4.1: Mean execution time of parsing and extracting features from a single file.

Feature	Unpacking performed?	Execution time per file (ms)
No features	No	258 (+/- 1)
	Yes	445 (+/- 4)
imphash	No	259 (+/- 1)
	Yes	445 (+/- 4)
icon hash	No	261 (+/- 1)
	Yes	445 (+/- 5)
contained resource hash	No	N/A
	Yes	447 (+/- 5)
TLSH	No	260 (+/- 1)
	Yes	457 (+/- 7)
All features	No	271 (+/- 2)
	Yes	459 (+/- 5)

Most of the time spent per file is not in fact spent on extracting the aforementioned features. Before extracting features from a file, the SHA-256 digest of a file is calculated, the file is potentially unpacked, and the file is parsed with the *pefile* module. Parsing files with *pefile* seems to be quite time-consuming, but allows fast retrieval of certain features after parsing. Extracting icon or imphash is for instance much faster, but extracting TLSH hash is not faster since this feature is

extracted from the raw bytes of the file. “no features extracted” is therefore given as a baseline.
It is evident that the processing time increases significantly when parsing and recursively unpacking all unpacked files. Simply parsing the files without extracting any features, took 72% longer time, compared to only parsing files when unpacking was not performed.
For divide-and-conquer clustering, all features must be extracted. The additional cost of extracting multiple features, compared to extracting a single feature only, seems negligible, provided that the file has been parsed with PE-file.

Clustering

The results of clustering with the different features are evaluated in various manners. The results presented in table 4.2 shows a significant difference the number of files that can be clustered with the different methods. When unpacking was not performed, 5 750.8 files (73%) could be clustered with imphash, 3 430.4 files (44%) with icon, and 2 626.8 (33%) with TLSH at a threshold of 100. Although more files are considered as “being in a cluster” by the underlying implementation, the files are not counted as being clustered if they are the only file in a “cluster”, when statistics are gathered.
TLSH was the only feature where the number of files clustered was affected by performing unpacking. This is due to TLSH clustering files with the “most similar” files (within a threshold), as opposed to the strict method used by other features. The current implementation compares the TLSH hash of each file to the TLSH hash of all other files, including the unpacked files. This seems to slightly decrease the number of files being successfully clustered, since a few files then will match better with an unpacked file.

Table 4.2: Number of files clustered, mean size and mean purity of clusters, when clustering the 7 858 files in the training set with individual features.

Feature	Unpacking performed?	Files clustered	Mean size of clusters	Mean purity of clusters
imphash	No	5750.8 (+/- 20.3)	10.4 (+/- 0.1)	93.2% (+/- 0.2%)
	Yes	5750.8 (+/- 20.3)	10.4 (+/- 0.1)	93.2% (+/- 0.2%)
icon hash	No	3430.4 (+/- 21)	8.7 (+/- 0.1)	89.3% (+/- 0.5%)
	Yes	3430.4 (+/- 21)	8.7 (+/- 0.1)	89.2% (+/- 0.5%)
contained resource hash	No	N/A		
	Yes	291.2 (+/- 30)	4.8 (+/- 0.4)	52.9% (+/- 4.4%)
TLSH	No	2626.8 (+/- 20.4)	4.5 (+/- 0)	95.7% (+/- 0.3%)
	Yes	2617.2 (+/- 21)	4.5 (+/- 0)	95.6% (+/- 0.3%)

In addition to being able to cluster many files in the data set, the created clusters should preferably be as large and pure as possible. The mean size and purity of the clusters are therefore also shown in table 4.2. Imphash seemed to be the individual feature that performed the best in terms of number of files clustered, mean size

and speed, but TLSH clusters had a higher purity, when using a distance threshold of 100.
Even though imphash indeed performed well, it certainly was not able to cluster all the files in the training set. If the features can complement each other, combining features can allow clustering of more files.
Another aspect of clustering that is highly interesting, is the execution time required to cluster the files. Table 4.3 shows the execution time of clustering the files with the individual features, as well as different combinations of features.
Clustering files with fast features is very fast, both when features are used individually and when they are combined. DAC clustering is slower than using fast features only, yet it is much faster than using TLSH or all features on all files.

Table 4.3: Total execution time for clustering the 7 858 files in the training set, using individual features and combinations of features.

Feature	Unpacking performed?	Time to cluster (in seconds)
imphash	No	0.559 (+/- 0.037)
	Yes	0.899 (+/- 0.058)
icon hash	No	0.528 (+/- 0.021)
	Yes	0.821 (+/- 0.017)
contained resource hash	No	N/A
	Yes	0.814 (+/- 0.034)
TLSH	No	61.284 (+/- 2.714)
	Yes	94.92 (+/- 3.944)
All fast features	No	0.642 (+/- 0.013)
	Yes	1.019 (+/- 0.109)
All features	No	64.427 (+/- 2.367)
	Yes	98.89 (+/- 3.498)
Divide-and-conquer	No	19.295 (+/- 0.49)
	Yes	41.343 (+/- 1.188)

The clustering performed as part of the training only gives an impression of how fast files can be clustered together and how pure the average cluster is. The effects of combining features to perform real-time clustering of files is presented in section 4.1.2.

4.1.2 Testing

During testing, files are clustered one by one, and labelled if possible. In this step, the ability of clustering and labelling unknown files is tested. Table 4.4 shows how well the features and combinations of features performed in terms of labelling the unknown files. Using no features, the files are simply parsed. The time spent on clustering files was small compared to the time spent parsing the files regardless of how the files were clustered, but there is still a difference between the cluster-

ing methods. The increased recall achieved by combining all fast features or all features, indicate that the chosen features can complement each other.

Table 4.4: Precision and recall achieved, as well as the execution time of parsing, clustering and labelling the 1 965 files in the testing set.

Feature	Unpacking performed?	Precision	Recall	Total execution time (in seconds)
No features	No	N/A		56.09 (+/- 0.742)
	Yes			77.666 (+/- 6.997)
imphash	No	94.7% (+/- 0.7%)	34.7% (+/- 0.5%)	55.197 (+/- 0.333)
	Yes	94.6% (+/- 0.7%)	35% (+/- 0.4%)	82.289 (+/- 3.443)
icon hash	No	91.8% (+/- 1.4%)	20.9% (+/- 0.7%)	55.922 (+/- 0.57)
	Yes	91.3% (+/- 1.5%)	21.2% (+/- 0.7%)	82.701 (+/- 3.886)
contained resource hash	No	N/A		
	Yes	83.5% (+/- 8.8%)	0.4% (+/- 0.1%)	80.971 (+/- 3.533)
TLSH	No	96% (+/- 0.8%)	30.4% (+/- 1.3%)	58.738 (+/- 0.792)
	Yes	94.7% (+/- 1%)	30.8% (+/- 1.4%)	79.883 (+/- 6.449)
Fast features	No	92.2% (+/- 1%)	42.4% (+/- 0.2%)	56.873 (+/- 0.444)
	Yes	91.9% (+/- 1.1%)	42.8% (+/- 0.2%)	83.011 (+/- 3.632)
All features	No	91.7% (+/- 1%)	49.2% (+/- 0.8%)	58.174 (+/- 0.386)
	Yes	91.1% (+/- 1.1%)	49.1% (+/- 0.8%)	85.557 (+/- 3.394)
Divide-and-conquer	No	92.3% (+/- 1%)	48.1% (+/- 0.8%)	57.508 (+/- 0.332)
	Yes	91.6% (+/- 1.3%)	47.9% (+/- 0.9%)	84.434 (+/- 4.343)

The most interesting comparisons are between clustering with fast features, all features and divide-and-conquer clustering. Although the difference is small, clustering files with DAC clustering is slightly faster than clustering all files with all features. The improved speed does come at a cost; the recall of DAC clustering is slightly lower than the recall of clustering with all features.
By using unpacked files to label the original files, the precision is reduced slightly. The recall is in some cases increased slightly, but in other cases the recall is decreased slightly when unpacking is performed. Despite the unpacking process causing the execution time to increase significantly, the precision or recall did not seem to increase significantly.

4.1.3 Triage

By labelling files that are identified as being related to a previously analysed file, the proof of concept has already performed some form of triage. The files that have been given labels through clustering should not be prioritised for in-depth analysis. The second task of the triage is to identify files that should be prioritised for in-depth analysis, which in the proof of concept is performed after the testing phase is done.
After clustering files with DAC clustering or all features, approximately 50% of the files in the testing set had been given a label. The remaining files might have been added to a cluster of poor quality, clusters that did not contain any labelled items or they might not have been identified as being similar to other files. An attempt

is therefore made analysing representative files in-depth and using the analysis result to predict labels of other files.

Table 4.5: Precision and recall achieved when attempting to label unknown files after analysing a representative file in the same cluster. The recall indicates that very few files could be labelled in this manner, while the precision indicates that approximately 20% of files labelled in this manner were given an incorrect label.

Feature	Unpacking performed?	Precision	Recall
imphash	No	86.7% (+/- 6.7%)	1% (+/- 0.3%)
	Yes	89.7% (+/- 6.7%)	1% (+/- 0.3%)
icon hash	No	76.1% (+/- 12.7%)	0.5% (+/- 0.1%)
	Yes	78.1% (+/- 11%)	0.6% (+/- 0.1%)
contained resource hash	No	N/A	
	Yes	0%	0%
TLSH	No	81.7% (+/- 3%)	1.3% (+/- 0.3%)
	Yes	82.5% (+/- 3%)	1.3% (+/- 0.3%)
Fast features	No	83.2% (+/- 4.6%)	1.1% (+/- 0.2%)
	Yes	88.1% (+/- 9.8%)	1.1% (+/- 0.2%)
All features	No	83% (+/- 3.8%)	1.5% (+/- 0.2%)
	Yes	82.3% (+/- 4%)	1.3% (+/- 0.3%)
Divide-and-conquer	No	78.9% (+/- 4.8%)	1.9% (+/- 0.2%)
	Yes	80.3% (+/- 6.4%)	1.8% (+/- 0.3%)

Table 4.5 shows the precision and recall achieved when attempting to use the results of in-depth analysis to predict labels of other files. Both the precision and recall is poor. The low recall suggests that there were few clusters left; most of the remaining files had not been found to be similar to any other files. The low precision suggests that some of the remaining clusters were of poor quality, containing files belonging to multiple classes.

Since few files could be labelled by performing in-depth analysis of representative files, there were many files left that needed in-depth analysis. When a file is analysed in-depth, it allows accurate labelling of a file, but would be very resource demanding. The precision and recall achieved through labelling files by means of clustering, must therefore be examined in relation to the cost of in-depth analysis. The results used to measure the performance of the triage, is therefore precision/recall and the share of files sent to in-depth analysis. The precision and recall shown in table 4.6 is much higher than the previously presented results, but carries a high "cost" in terms of how many files would have to be analysed in-depth. A precision of 100% would for instance be very bad if all the files had to be sent to in-depth analysis, such as when no triage is performed.

The number of mislabelled files can also be considered as a "cost" of performing triage, since there is a trade-off where the number of files in need of in-depth analysis is reduced, but the number of mislabelled files increases. Since all files

Table 4.6: The final results from performing triage, in terms of how many files had to be analysed in-depth and how precise the labelling was.

Feature	Unpacking performed?	Files sent to in-depth analysis (lower is better)	Precision / Recall
No features	No	100%	100%
	Yes	100%	100%
imphash	No	62.6% (+/- 0.7%)	98% (+/- 0.3%)
	Yes	63.1% (+/- 0.6%)	97.9% (+/- 0.3%)
icon hash	No	76.7% (+/- 1.1%)	98% (+/- 0.4%)
	Yes	77% (+/- 1.2%)	97.9% (+/- 0.5%)
contained resource hash	No	N/A	
	Yes	99.5% (+/- 0.1%)	99.9% (+/- 0%)
TLSH	No	67.2% (+/- 1.3%)	98.5% (+/- 0.3%)
	Yes	67.3% (+/- 1.1%)	98.1% (+/- 0.3%)
Fast features	No	53.2% (+/- 0.4%)	96.3% (+/- 0.4%)
	Yes	53.6% (+/- 0.4%)	96.2% (+/- 0.5%)
All features	No	45.5% (+/- 0.6%)	95.4% (+/- 0.5%)
	Yes	46.2% (+/- 0.5%)	95.1% (+/- 0.6%)
Divide-and-conquer	No	46.8% (+/- 0.7%)	95.8% (+/- 0.5%)
	Yes	46.6% (+/- 0.5%)	95.4% (+/- 0.6%)

are labelled, the number of false positives is equal to false negatives, and the recall is therefore equal to the precision.
Since the difference in execution time was negligible between clustering with all features and DAC clustering, and performing unpacking only led to an increase in execution time, the best results were achieved when using all features for clustering, without performing unpacking. By clustering using this method, less than half of the unknown files had to be analysed in-depth.

4.2 Large data set

The amount of data handled by a system performing malware triage, would likely be much greater than the data set used for evaluating the different features and clustering methods described in the previous sections. For a more realistic test of the method, the experiments that are likely to be affected by the size of the data set had to be repeated on a much larger data set.
As in the previous sections, the results are presented according to the steps in the triage pipeline described in section 3.1, but the preprocessing step is skipped since the results of performing unpacking was assumed to not depend on the size of the data set.

4.2.1 Training

The training step was again divided into a feature extraction and clustering step.

Feature extraction

The processing time required to parse a file and extract features was roughly equal to the processing time per file in the small data set, as seen in table 4.7. The feature extraction was therefore confirmed to have a time-complexity of $O(n)$.

Table 4.7: Mean execution time required to parse and extract features from a file in the large data set.

Feature	Execution time per file (ms)
No features	367 (+/- 2)
imphash	372 (+/- 2)
icon hash	370 (+/- 2)
TLSH	381 (+/- 1)
All features	387 (+/- 1)

The small difference was likely caused by the properties of the files in the data sets (e.g. file size), since the execution time per file also was increased when only parsing the files without extracting any features. Vhash is not included in these results since it was retrieved from VirusTotal metadata and could not be extracted from the files.

Clustering

When clustering the files, findings were mostly the same as for the small data set. The difference in mean size of clusters is most likely caused by the classes being bigger in this data set.

Table 4.8: Number of files clustered, mean size and mean purity of clusters created when clustering the 185 841 files in the training set using various features.

Feature	Files clustered	Mean size of clusters	Mean purity of clusters
Vhash	172633.4 (+/- 24.35)	26.59 (+/- 0.11)	99.32% (+/- 0.03%)
imphash	162693.6 (+/- 82.41)	56.14 (+/- 0.43)	90.5% (+/- 0.15%)
icon hash	75192.8 (+/- 98.7)	28.04 (+/- 0.08)	88.85% (+/- 0.11%)
TLSH	159983 (+/- 27.23)	22.49 (+/- 0.13)	93.14% (+/- 0.28%)

Even though Vhash was used in the collection of the large data set, only 4 095 additional files (1.76%) seem to have been added to the data set based on Vhash. As shown in the table, clusters created based on Vhash were smaller than clusters based on imphash, but the purity of Vhash clusters was much higher, and allowed

more files to be added to a cluster of at least two files. The performance of Vhash was not expected to have been affected that much by the method used in collecting the data set, and might suggest that Vhash is a feature that performs well in clustering of PE-files.

Table 4.9: Total execution time for clustering the 185 841 files in the training set, using individual features and combinations of features. "No features" corresponds to simply iterating over the files without making any attempt at clustering them.

Feature	Time to cluster (in seconds)
No features	14.447 (+/- 0.314)
Vhash	17.232 (+/- 0.345)
imphash	16.755 (+/- 0.189)
icon hash	15.763 (+/- 0.184)
TLSH	4789.028 (+/- 184.49)
Fast features (w/o Vhash)	17.5 (+/- 0.173)
Fast features (w/ Vhash)	20.776 (+/- 0.553)
All features (w/o Vhash)	4974.026 (+/- 303.186)
All features (w/ Vhash)	4850.354 (+/- 109.633)
DAC (w/o Vhash)	2441.116 (+/- 58.583)
DAC (w/ Vhash)	1667.37 (+/- 26.327)

The findings related to the execution time of clustering, shown in table 4.9, are roughly the same as for clustering the small data set – clustering with fast features is very fast, DAC clustering is slower, and clustering with all features is much slower. By using Vhash in the DAC clustering in addition to imphash, icon hash and TLSH, the execution time of clustering files with DAC clustering was further reduced. This is because it allowed a larger share of the files to be successfully clustered with fast features.

4.2.2 Testing

Although the findings during the training phase were quite similar for both data sets, the findings from the testing phase differed slightly.
Table 4.10 shows that the precision and recall achieved through simply clustering files based on independent features or the various combinations of features is significantly better for experiments performed on the large data set. The massively increased training set, a smaller share of packed files, the combination of these

Table 4.10: Precision and recall achieved, as well as the execution time of parsing, clustering and labelling the 46 461 files in the testing set.

Feature	Precision	Recall	Execution time (in seconds)
No features	N/A		1081.395 (+/- 10.185)
Vhash	99.8% (+/- 0%)	92.5% (+/- 0%)	1135.257 (+/- 14.06)
imphash	98.6% (+/- 0%)	78.7% (+/- 0.2%)	1105.803 (+/- 13.164)
icon hash	97.4% (+/- 0.1%)	36.3% (+/- 0.2%)	1096.736 (+/- 15.62)
TLSH	98% (+/- 0.1%)	78.2% (+/- 0.3%)	2776.76 (+/- 16.458)
Fast features (w/o Vhash)	97.9% (+/- 0.1%)	81% (+/- 0.2%)	1118.617 (+/- 17.728)
Fast features (w/ Vhash)	98.9% (+/- 0%)	94.9% (+/- 0.1%)	1121.405 (+/- 14.785)
All features (w/o Vhash)	97.7% (+/- 0.1%)	91.5% (+/- 0.2%)	3134.075 (+/- 38.865)
All features (w/ Vhash)	98.5% (+/- 0%)	95.5% (+/- 0.1%)	3214.105 (+/- 52.14)
DAC (w/o Vhash)	97.8% (+/- 0.1%)	91.3% (+/- 0.2%)	1326.37 (+/- 25.509)
DAC (w/ Vhash)	98.6% (+/- 0%)	95.4% (+/- 0.1%)	1191.756 (+/- 10.496)

properties, or other properties of the data set seems to allow more effective triage of files.
In the table one can also see that using Vhash in addition to the other fast features made it possible to achieve a higher precision and recall for clustering files. The recall achieved by using “fast features w/ Vhash”, “all features w/ Vhash” and “DAC clustering w/ Vhash” were all approximately 95%. The highest recall was achieved when clustering all files with all features, but the time required to cluster the files was also much higher than when the other clustering methods were used. The execution-time for clustering files with DAC clustering was reduced further when Vhash was used, since it allowed an even greater number of files to be clustered with fast features only. The precision and recall is closer to the results of clustering all files with all features, but the total execution time is much closer to that of clustering files with fast features only, than the execution time of clustering all files with all features.

4.2.3 Triage

Since the real-time clustering performed in the testing could be used to label a large amount of the files, much fewer files had to be sent to in-depth analysis. Table 4.11 shows the precision/recall achieved after analysing files in-depth, as well as the share of files that had to be analysed in-depth.
The difference in precision is quite small among the clustering methods that use

Table 4.11: The final results from performing triage, in terms of how many files had to be analysed in-depth and the precision of labelling files.

Feature	Files sent to in-depth analysis (lower is better)	Precision / Recall
Vhash	7.29% (+/- 0.03%)	99.82% (+/- 0.02%)
imphash	20.13% (+/- 0.26%)	98.85% (+/- 0.03%)
icon hash	62.79% (+/- 0.24%)	99.05% (+/- 0.04%)
TLSH	20.16% (+/- 0.27%)	98.4% (+/- 0.11%)
Fast features (w/o Vhash)	17.23% (+/- 0.24%)	98.26% (+/- 0.04%)
Fast features (w/ Vhash)	4.06% (+/- 0.07%)	98.93% (+/- 0.05%)
All features (w/o Vhash)	6.36% (+/- 0.16%)	97.82% (+/- 0.06%)
All features (w/ Vhash)	3.07% (+/- 0.07%)	98.58% (+/- 0.04%)
DAC (w/o Vhash)	6.69% (+/- 0.19%)	97.96% (+/- 0.06%)
DAC (w/ Vhash)	3.28% (+/- 0.09%)	98.68% (+/- 0.04%)

combinations of features. The difference in the share of files that would have to be sent to in-depth analysis is more significant. The share of files that could be filtered out by the triage was highest when Vhash was used in addition to the other features. Among the clustering methods combining different features, including Vhash, the clustering method that allowed the triage to filter out the most files was using all of the features on all files, with only 3.07% of the files remaining. DAC clustering was not far behind, with 3.28% files remaining, and using fast features only was slightly further behind, with 4.06% files remaining.
These clustering methods are the most interesting to investigate further. Figure 4.1 compares these clustering methods in terms of the total execution time for clustering the files and the number of files in need of in-depth analysis. As one can see in this figure, DAC clustering performed almost as well as clustering with all features in terms of how many files that could be filtered out. DAC clustering also performed almost as well as clustering with fast features only in terms of execution time for clustering files.
Based on these results, DAC clustering seems to perform well in terms of clustering files, but it is not as clear which method that was best for triage. Was decrease in execution time for clustering files worth the increased number of files in need of in-depth analysis? This topic is discussed further in section 5.1.

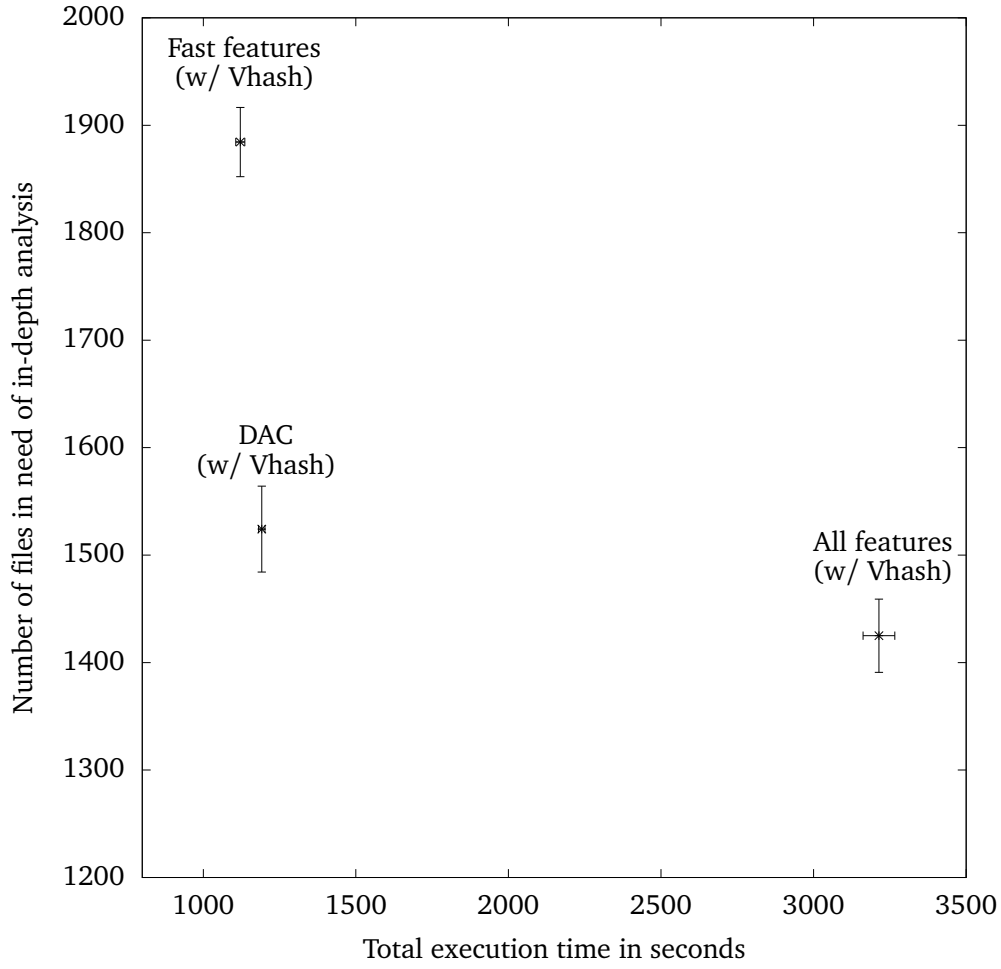


Figure 4.1: Comparison between clustering with Fast features, All features and DAC clustering. DAC clustering is able to filter out almost as many files as clustering with all features, and is only slightly slower than clustering with fast features.

Chapter 5

Discussion

This chapter discusses the results presented in chapter 4 and attempts to uncover any weaknesses with the experiments that could affect how significant the results are.

5.1 Analysis of results presented in section 4.2.2 and 4.2.3

The three clustering methods that performed the best, were to cluster with fast features, cluster with all features and DAC clustering – with Vhash being used in all cases. The main differences between these methods are shown in table 5.1. Some elements have been simplified to allow easier comparison of the values.

Table 5.1: Comparison of the best performing clustering methods. Clustering files with all features took much longer time than the other clustering methods, but allowed a greater number of files to be filtered out.

Features	Execution time of parsing and clutering files (minutes and seconds)	Number of files sent to in-depth analysis (lower is better)
No features (no triage)	18m1s	46461
Fast features	18m41s	1884
All features	53m33s	1425
DAC	19m51s	1524

The method that required the least execution time for clustering files, was to cluster files with fast features only. Even though the quickest method only used fast features, the share of files filtered out was significantly worse than when using DAC clustering or clustering with all features.
The reason why clustering with all features and DAC clustering performed better than using fast features only, is because TLSH was able to complement the other features, and thus allowed an increased number of files to be clustered. TLSH

is most likely not the only feature that can complement imphash, icon hash and Vhash. Other slow features, or even other fast features such as *peHash* and *Rich hash*, would likely also be able to complement the features for finding similar files. Ideally, one would be able to filter out all files with fast features only, but this is somewhat unrealistic since the fast features are quite strict in the comparison. Since fuzzy hashes such as TLSH, *ssdeep* or *sdhash* allows more flexibility in the matching, they will likely be able to complement fast features regardless of how many fast features are used.

An important factor in the analysis of which clustering method performed the best of DAC clustering and clustering with all features, is how much processing time would be required to analyse the files not filtered out by DAC clustering. By clustering the files with all features, 99 additional files were filtered out, compared to the files filtered out with DAC clustering. The processing time that is required to analyse a file with in-depth analysis is at least two minutes, and potentially five minutes. Analysing the 99 additional files would therefore require an additional cost in execution time of at least $99 \cdot 2$ minutes $= 198$ minutes. This is much greater than the approximately 34 minutes saved by using DAC clustering, instead of clustering all files with all features, during the triage. If more than two minutes would be required for in-depth analysis, which is not unreasonable when the overhead of launching the virtual environments, the difference would be even greater.

This analysis indicates that even though there was a tiny difference in the share of files being filtered out, and a significant difference in execution time of clustering files, using all features for all files results in a lower total execution time for labelling all files. The reduction in execution time for clustering files, would have to be much more significant, to justify the small reduction in files filtered out with DAC clustering.

Due to the time-complexity of clustering files with slow features, it is expected that the difference in execution time between DAC clustering and clustering with all features, will become more significant as the size of the data set grows. DAC clustering could therefore potentially perform better than clustering with all features if applied to a much larger data set. Identifying if this is the case, would however be difficult with an empirical approach. A theoretical approach could therefore be better suited. Due to the optimisations that reduce the time-complexity of clustering files with distance-based features, described in section 2.1.6, the time-complexity of clustering files is lower than $O(n^2)$ and highly dependent on properties of the data set. Using a theoretical approach would therefore also be difficult.

Based on the findings of these experiments, one can conclude that DAC clustering can be used to filter out a greater number of files than some of the tested clustering methods. An even greater number of files can however be filtered out by clustering all files with all (available) features. Even though additional processing is required to cluster all files with all features, the overall cost in processing power is lower than when using DAC clustering, since one would have to analyse more files in-depth when using DAC clustering. This is at least the case when performing triage

on the files in the data sets described in section 3.3.1, using the features described in section 3.2.3 and the method described in section 3.1.

5.2 Findings related to unpacking

Unpacking files has been proposed as a solution to combat the usage of packers in previously published papers. Performing unpacking and using unpacked files to improve clustering performance, was found to be a greater challenge than it was indicated in these papers.
Unpacking files with generic unpacking, where files are executed to unpack themselves, does not seem to be suitable for large scale malware triage. Existing solutions that can restore a functioning executable and therefore are able to handle recursively packed files, can typically consume the same amount of resources as performing dynamic analysis. Additionally, the solutions will often fail when attempting to unpack files that are found in the wild. In situations where files must be unpacked, generic unpacking can be a viable solution, but in triage of PE-files however, it seems too resource demanding and provides little value.
Using static unpacking, can allow fast unpacking of files, but many such tools cannot be automated. Using ClamAV was viable, since it has a command-line interface and can unpack files quickly. Even though ClamAV only seems to be able to unpack about one in four PE-files identified as being packed, static unpacking with ClamAV is very fast and the cost of unpacking is therefore low. That does however not mean that it is cost-effective to unpack files for the purpose of improving the triage of PE-files. The precision or recall would have to be improved by the unpacking process, for it to be considered as cost-effective.
In the experiments, the precision and recall was reduced slightly when unpacking was performed. This is likely because the unpacked files were not handled optimally. There are likely methods that can be used to handle the unpacked files, but identifying the ideal method is not trivial. MutantX-S, described one method, but was found to have poor performance on malware found in the wild [37]. No research projects have seemed to both describe how unpacked files should be used to improve the precision or recall of the triage and exhibit good performance on clustering malware found in the wild. A study of how the unpacked files should be handled to achieve optimal effect on precision and recall, could therefore be of value. A subsequent cost-benefit analysis of the unpacking process would make it possible to identify whether unpacking really should be attempted or not.
In [35] and [36], data sets that only contained non-packed files were used in evaluation of clustering. The authors of these papers argued that the findings were representative for malware found in the wild, since files can be unpacked using off-the-shelf unpackers. There does not seem to be any evidence supporting these claims. Many files can be unpacked through static unpacking or computationally expensive generic unpacking, but the share of files that can be unpacked without requiring manual labour seems to be much lower than 100%.
Interestingly, most of the files identified as being packed, could be successfully

clustered and labelled even though they were not unpacked. This indicates that many files identified as being packed, either have been falsely detected as being packed, or that features of the contained executables often are exposed in the new header or new sections (such as icon). This further reduces the likelihood of unpacking being cost-effective.

5.3 Potential issues with the data sets used in the experiments

The results from the experiments were reliable, with a small confidence interval. For several, this does not necessarily mean that the same performance would be achieved in a real malware triage environment.
First, the results only apply to triage of files that the data sets are representative for. The large data set was supposed to be as similar as possible to files typically found in the wild, but some limitations were required, to ensure that it was feasible to carry out the experiments.
The data sets only consist of files where a family name could be determined based on VirusTotal results. As mentioned in [43], the files in the data sets could therefore be easier to cluster than files typically found in the wild. The data set does not include any benign PE-files either, since labelling these files would be difficult. The results are therefore potentially not representative for the triage performed by anti-virus software vendors.
During collection of the data sets, files were labelled based on anti-virus reports retrieved from VirusTotal. If no label could be determined for a file, it was not included in the data set. This could for instance be a file that is packed in a manner that makes it difficult to identify a label for and difficult to cluster. If so, unpacking the file could potentially provide more value than what was gained from unpacking the files in the small data set in experiments on this data set. The poor results related to unpacking, are therefore potentially not representative for unpacking of files found in the wild.
Whether the files in the data set were easier to cluster than files typically found in the wild, does however not affect the results related to the performance of DAC clustering to the same degree. The individual performance of features would be lower if files were more difficult to cluster, but this was also the case when clustering the small data set. The individual performance of features was poor when clustering files in the small data set, compared to the performance when clustering files in the large data set. Even so, the difference between DAC clustering and other clustering methods was roughly the same. One can therefore assume that findings based on comparing DAC clustering to other clustering methods, were not affected by how easy the files were to cluster.
The same applies to the usage of Vhash in the experiments on the large data set. As section 3.3.1 described, Vhash was used in the collection of the data set, and the performance of Vhash might not be representative because of this. Since the

findings did not depend on the individual performance of features, using Vhash in the clustering was fine. Using Vhash in clustering of the large data set, also allowed better insight into how well the proof of concept was able to filter out files for the purpose of triage, provided that the features being used performed well.

5.4 Issues and potential improvements for the proposed method and implementation

A major issue that would prevent the triage being performed in these experiments from being viable in a real triage environment, is that new data constantly is added without removing any old data. As described in [40], malware is constantly evolving. It is unlikely that new files will be matched with old files that have not been seen in a very long time. To avoid false positives and ensure that the computational complexity of clustering does not increase indefinitely, old files that have not been seen in a long time, should be removed from the existing clusters. Another issue, is that any errors made in the labelling is not detected. Errors will likely propagate to new unknown files that are similar to the mislabelled files. Solutions to this was proposed in [40], but would result in a large computational overhead.
Although the individual performance of features can be blamed for poor performance in the experiments on the small data set, the proof of concept did not utilise the availability of multiple features as well as it probably could. It is an potential weakness that files are clustered with multiple fast features, but only the label of one cluster is considered.
If divide-and-conquer clustering had not been used, and all features could be extracted from all files, a combined metric as the one proposed in [27] could have been used. It is likely more difficult to create a combined metric when certain features are missing on some files, but it should be possible to create a similar metric. By using this approach, it might be possible to achieve greater precision and recall for clustering, resulting in better triage.
The last issue in the implementation, is related to the basic measure of cluster quality. Using external evaluation seems to be effective in evaluation of quality for clusters that contained labelled files. The lack of any measures for internal evaluations, did however result in poor precision for labelling files in clusters that did not contain any labelled files. A good measure for internal evaluation would be necessary to prioritise files for in-depth analysis correctly. An alternative to implementing a measure of cluster quality would be to not use features that occasionally are shared between files belonging to different classes. Vhash clusters did for instance rarely contain files belonging to multiple classes. If the distance threshold used for clustering with TLSH had been lower, TLSH clusters would likely also have been very pure, though the size of clusters and share of files clustered likely would have been lower.

5.4.1 Algorithmic attacks

The proof of concept is susceptible to algorithmic attacks since the evaluation of cluster quality is simple and does not verify if files are similar based on more than one feature. By creating a malicious file with the same imports in the same order as a benign file, the files will have identical imphash, and the proof of concept would assume that the malicious file is related to the benign file and possibly label it as benign as well. The proof of concept was implemented to evaluate if divide-and-conquer clustering has merit as a method for improving the speed or accuracy of clustering, and protection against algorithmic attacks was therefore not implemented. If the clustering method is implemented in a real malware triage environment, it is vital that the risk of algorithmic attacks is assessed, and measures are implemented. Using a combined metric as the one described in [27] would likely reduce the risk of algorithmic attacks.

5.5 Future work

Due to the limited timeframe of this project, some issues and questions have not been resolved. This section describes the topics that would likely be of interest to investigate further.

A good measure for internal evaluation would be required to prioritise files for in-depth analysis correctly. The reason why this type of evaluation is needed, is that some clusters based on imphash and icon hash, contain files belonging to different classes. An alternative would be to only cluster files with features that allow clustering with high precision, but this would likely lead to a large decrease in files being clustered. Identifying a good method for performing internal evaluation is therefore likely a better solution. To perform prioritisation of files properly, using the method described in section 3.1, a solution to this problem is necessary. It is therefore proposed that this issue is investigated further.

There is also a need for a method that describes how PE-files should be unpacked and how the unpacked files should be used, to minimise the cost of unpacking and maximising the benefit of the unpacking process. Once a good solution is identified for doing so, an evaluation should be made whether unpacking really should be described as the solution to performing triage on a data set containing packed PE-files. Unless major improvements are made within the field of generic unpacking, it seems unlikely that generic unpacking will be a viable option for triage.

Investigation should be made on whether the challenge of labelling files based on VirusTotal reports, described in [43] could have been the reason for why unpacking files did not seem to improve the precision or recall of clustering. If this is the case, the results would have little significance, since unpacking then could have had more effect when processing files found in the wild.

No studies documenting the precision and recall of clustering with Vhash, seems to have been published. It is therefore difficult to assess if the method used for

collecting the data set, is the reason for the good performance of Vhash or not. Research should therefore be performed on the precision and recall one can achieve through clustering and labelling files based on Vhash. If the results indeed were representative, Vhash is a highly attractive feature. More attempts should also be made at combining Vhash with other features, to identify if any other features can complement Vhash and help increase the precision and recall further.
Finally, the idea behind *peHash* seems to have been very good. Identification of similar files by comparing if hashes are equal, allows fast and extremely scalable clustering of PE-files. It seems like the design and choice of features might not have been ideal, since imphash, a hash solely based on the import address table, allegedly performs better than *peHash*. Using machine learning or possibly even deep learning, it might be possible to create a hash function for clustering PE-files that allow clustering of files with even greater precision and recall.

Chapter 6

Conclusion

This final chapter presents the findings of the thesis. In addition to answering the research questions of the thesis, additional findings and contributions that arose are also presented.
The research questions of the thesis were related to how DAC clustering performs in comparison to more naive clustering methods, how feasible it is to unpack files for the purpose of triage, and to which extent unpacking helps improve performance in triage of PE-files.
By using DAC clustering, it is possible to cluster files much faster than if all files are clustered with all features. There is however a small reduction in the number of files that can be labelled when clustering files, and the number of files that is filtered out in the triage is therefore reduced.
Even though the difference in the number of files being filtered out is small, the cost of analysing a file with in-depth analysis is high. Assuming a processing time of 2 to 5 minutes per file in need of in-depth analysis, the overall cost in processing time would be significantly greater when using DAC clustering, than when clustering all files with all features. DAC clustering did however perform better than the other clustering methods it was compared against.
The results indicate that a difference in the share of files being filtered out, is much more significant than a difference in the execution time required to cluster files. This is the reason why DAC clustering performs worse than using all features for all files, when applied in triage of PE-files. Despite exhibiting worse performance in triage of PE-files, there could be other situations where a slight decrease in recall would be more acceptable, and DAC clustering therefore would be more suitable.
Performing unpacking of files, was found to be more challenging than previously published papers had suggested. Generic unpacking was found to be very resource demanding and cannot be used to unpack most files that are found in the wild. Using generic unpacking for large-scale triage of PE-files is therefore not feasible with the currently available methods for generic unpacking. Static unpacking could be used to unpack files quickly, but could only be used to unpack approximately one in four files that had been identified as being packed. One can therefore

conclude that it is feasible to unpack files to a *small* extent, but not to an extent that would justify the use of data sets with non-packed files only. Since most files cannot be unpacked in a manner that would be fast enough for the purpose of triage, it seems misleading to claim that a data set containing only non-packed files is representative of files found in the wild.

In addition to not seeming very feasible, unpacking files did not seem to improve the precision or recall of clustering. Even though the experiments indicated that unpacking files resulted in slightly degraded performance, the method of treating the unpacked files might not have been optimal. Since most of the packed files could be clustered successfully without performing unpacking, and the data sets purely consisted of files where labels could be determined based on VirusTotal reports, the data might have consisted of files that were easier to cluster than files typically found in the wild. Whether unpacking files improves performance in triage of PE-files, is therefore inconclusive.

Even though the proposed clustering method was not able perform better than a more naive clustering method, the project contributed to increased domain knowledge. The results showed how a fairly big increase in execution time of clustering files can be worth it, provided that the share of files being filtered out is increased. Better insight into how various features perform and how they complement each other, can be of value in future research on clustering and triage of PE-files. So can the obtained knowledge related to clustering of packed PE-files, how unpacking can be performed, and the complexity of unpacking files.

Bibliography

[1] VirusTotal, *Statistics - VirusTotal*, Jan. 2020. [Online]. Available: https://www.virustotal.com/en/statistics/ (visited on 21/01/2020).

[2] AV-TEST, *Malware Statistics & Trends Report*, Apr. 2020. [Online]. Available: https://www.av-test.org/en/statistics/malware/ (visited on 23/04/2020).

[3] Kaspersky, *Kaspersky Lab detects 360,000 new malicious files daily – up 11.5% from 2016*, Dec. 2017. [Online]. Available: https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily (visited on 23/04/2020).

[4] L. Nataraj, V. Yegneswaran, P. Porras and J. Zhang, 'A Comparative Assessment of Malware Classification Using Binary Texture Analysis and Dynamic Analysis', in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '11, Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 21–30, ISBN: 9781450310031. DOI: 10.1145/2046684.2046689.

[5] C. Hirsch, *Collateral damage outcomes are prominent in cyber warfare, despite targeting*, English, 2018. [Online]. Available: https://search.proquest.com/docview/2018926694 (visited on 30/04/2020).

[6] M. Satran, M. LeBLanc, C. Robertson, K. Bridge, M. Kayser, Y. Zhu, J. Kennedy and C. Warrington, *PE Format*, Aug. 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 20/05/2020).

[7] dzzie, *Understanding the Import Address Table*, 2013. [Online]. Available: http://sandsprite.com/CodeStuff/Understanding_imports.html (visited on 25/05/2020).

[8] J. Yonts, 'Attributes of Malicious Files', *SANS Institute Information Security Reading Room*, Jun. 2012. [Online]. Available: https://www.sans.org/reading-room/whitepapers/malicious/paper/33979 (visited on 22/01/2020).

[9] F. Khelifi and A. Bouridane, 'Perceptual Video Hashing for Content Identification and Authentication', *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 1, pp. 50–67, Jan. 2019. DOI: 10.1109/tcsvt.2017.2776159.

[10] A. Swinnen and A. Mesbahi, *One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques*, Aug. 2014. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Mesbahi-One-Packer-To-Rule-Them-All-WP.pdf (visited on 25/05/2020).

[11] A. T. Alexey Kleymenov, *Mastering Malware Analysis: the complete malware analyst's guide to combating malicious software, APT, cybercrime, and IoT attacks*. Packt Publishing, 6th Jun. 2019, 562 pp., ISBN: 1789610788.

[12] Y. Nakatsuru, *Recommendation of Perfect Unpacking*, Apr. 2014. [Online]. Available: https://www.jpcert.or.jp/present/2014/20140424ssmjp.pdf (visited on 25/05/2020).

[13] N. M. Hai, M. Ogawa and Q. T. Tho, 'Packer identification based on metadata signature', in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*, ACM, Dec. 2017. DOI: 10.1145/3151137.3160687.

[14] S. Basu, A. Banerjee and R. J. Mooney, *Active semi-supervision for pairwise constrained clustering*, English, 2004. [Online]. Available: https://search.proquest.com/docview/940861659 (visited on 24/05/2020).

[15] R. Jacob, D. Koschützki, K. A. Lehmann, L. Peeters and D. Tenfelde-Podehl, 'Algorithms for Centrality Indices', in *Network Analysis*, Springer Berlin Heidelberg, 2005, pp. 62–82. DOI: 10.1007/978-3-540-31955-9_4.

[16] E. Rowell, *Big-O Cheat Sheet*, Aug. 2016. [Online]. Available: https://www.bigocheatsheet.com/ (visited on 07/05/2020).

[17] DigitalNinja, *Fuzzy Clarity: Using Fuzzy Hashing Techniques to Identify Malicious Code*, Apr. 2007. [Online]. Available: https://web.archive.org/web/20180403132547/http://www.shadowserver.org/wiki/uploads/Information/FuzzyHashing.pdf (visited on 27/01/2020).

[18] F. Pagani, M. Dell'Amico and D. Balzarotti, 'Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis', in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18, Tempe, AZ, USA: Association for Computing Machinery, 2018, pp. 354–365, ISBN: 9781450356329. DOI: 10.1145/3176258.3176306.

[19] V. Roussev, 'An evaluation of forensic similarity hashes', *Digital Investigation*, vol. 8, S34–S41, Aug. 2011. DOI: 10.1016/j.diin.2011.05.005.

[20] J. Oliver, C. Cheng and Y. Chen, 'TLSH - A Locality Sensitive Hash', in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, Nov. 2013, pp. 7–13. DOI: 10.1109/CTC.2013.9.

[21] A. Azab, R. Layton, M. Alazab and J. Oliver, 'Mining Malware to Detect Variants', in *2014 Fifth Cybercrime and Trustworthy Computing Conference*, Nov. 2014, pp. 44–53. DOI: 10.1109/CTC.2014.11.

[22] B. Wallace, 'Optimizing ssDeep for use at scale', in *Proceedings of the 25th Virus Bulletin International Conference*, M. Grooten, Ed., Virus Bulletin, Nov. 2015. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2015/11/optimizing-ssdeep-use-scale (visited on 12/12/2019).

[23] G. Wicherski, 'PeHash: A Novel Approach to Fast Malware Clustering', in *Proceedings of the 2nd USENIX Conference on Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, ser. LEET'09, Boston, MA: USENIX Association, 2009, p. 1. [Online]. Available: https://www.usenix.org/conference/leet-09/pehash-novel-approach-fast-malware-clustering (visited on 25/10/2019).

[24] Mandiant, *Threat Research: Tracking Malware with Import Hashing*, Jan. 2014. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html (visited on 18/01/2020).

[25] J. Choi, H. Kim, J. Choi and J. Song, 'A Malware Classification Method Based on Generic Malware Information', in *Neural Information Processing*, S. Arik, T. Huang, W. K. Lai and Q. Liu, Eds., Cham: Springer International Publishing, 2015, pp. 329–336, ISBN: 978-3-319-26535-3. DOI: 10.1109/SAINT.2012.48.

[26] M. Chikapa and A. P. Namanya, 'Towards a Fast Off-Line Static Malware Analysis Framework', in *2018 6th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, Aug. 2018, pp. 182–187. DOI: 10.1109/W-FiCloud.2018.00035.

[27] A. P. Namanya, Q. K. A. Mirza, H. Al-Mohannadi, I. U. Awan and J. F. P. Disso, 'Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing', in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, Aug. 2016, pp. 91–98. DOI: 10.1109/FiCloud.2016.21.

[28] P. Silva, S. Akhavan-Masouleh and L. Li, 'Improving Malware Detection Accuracy by Extracting Icon Information', in *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, IEEE, Apr. 2018. DOI: 10.1109/mipr.2018.00088.

[29] VirusTotal, *Multi-similarity searches*, 2019. [Online]. Available: https://support.virustotal.com/hc/en-us/articles/360001398517-Multi-similarity-searches (visited on 26/04/2020).

[30] B. Hoyt, *dhash*, Aug. 2017. [Online]. Available: https://github.com/Jetsetter/dhash (visited on 29/04/2020).

[31] A. Jungheit, *VirusTotal vHash Maltego Transform*, Oct. 2019. [Online]. Available: https://github.com/arieljt/VTvHash-Maltego/tree/3e4bbe2c9bcdcf38d81fc08b6fd198655290ac2f (visited on 29/04/2020).

[32] 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 *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds., Cham: Springer International Publishing, 2017, pp. 119–138, ISBN: 978-3-319-60876-1. DOI: 10.1007/978-3-319-60876-1_6.

[33] Kaspersky Global Research & Analysis Team, *The devil's in the Rich header*, Mar. 2018. [Online]. Available: https://securelist.com/the-devils-in-the-rich-header/84348/ (visited on 22/01/2020).

[34] M. Poslušný and P. Kálnai, 'Rich headers: Leveraging this mysterious artifact of the PE format', in *Virus Bulletin International Conference 2019*, Virus Bulletin, Oct. 2019. [Online]. Available: https://www.virusbulletin.com/conference/vb2019/abstracts/rich-headers-leveraging-mysterious-artifact-pe-format/ (visited on 18/01/2020).

[35] J. Jang, D. Brumley and S. Venkataraman, 'BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis', in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 309–320, ISBN: 9781450309486. DOI: 10.1145/2046707.2046742.

[36] I. Shiel and S. O'Shaughnessy, 'Improving file-level fuzzy hashes for malware variant classification', *Digital Investigation*, vol. 28, pp. 88–94, 2019, ISSN: 1742-2876. DOI: 10.1016/j.diin.2019.01.018.

[37] X. Hu, K. G. Shin, S. Bhatkar and K. Griffin, 'MutantX-S: Scalable Malware Clustering Based on Static Features', in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, San Jose, CA: USENIX, 2013, pp. 187–198, ISBN: 978-1-931971-01-0. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/hu (visited on 14/10/2019).

[38] S. Mariani, L. Fontana, F. Gritti and S. D'Alessio, 'PinDemonium: A DBI-based generic unpacker for Windows executables', in *Black Hat USA 2016*, 2016. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Mariani-Pindemonium-A-Dbi-Based-Generic-Unpacker-For-Windows-Executables-wp.pdf (visited on 18/02/2020).

[39] M. M. Masud, T. M. Al-Khateeb, K. W. Hamlen, J. Gao, L. Khan, J. Han and B. Thuraisingham, 'Cloud-Based Malware Detection for Evolving Data Streams', *ACM Transactions on Management Information Systems*, vol. 2, no. 3, Oct. 2008, ISSN: 2158-656X. DOI: 10.1145/2019618.2019622.

[40] J. Ouellette, A. Pfeffer and A. Lakhotia, 'Countering malware evolution using cloud-based learning', in *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, Oct. 2013, pp. 85–94. DOI: 10.1109/MALWARE.2013.6703689.

[41] M. Hassani and T. Seidl, 'Using internal evaluation measures to validate the quality of diverse stream clustering algorithms', *Vietnam Journal of Computer Science*, vol. 4, no. 3, pp. 171–183, Oct. 2016. DOI: 10.1007/s40595-016-0086-9.

[42] M. Khalilian, F. Z. Boroujeni, N. Mustapha and M. N. Sulaiman, 'K-Means Divide and Conquer Clustering', in *2009 International Conference on Computer and Automation Engineering*, IEEE, Mar. 2009. DOI: 10.1109/iccae.2009.59.

[43] P. Li, L. Liu, D. Gao and M. K. Reiter, 'On Challenges in Evaluating Malware Clustering', in *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer and C. Kreibich, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 238–255, ISBN: 978-3-642-15512-3. DOI: 10.1007/978-3-642-15512-3_13.

[44] D. Pfitzner, R. Leibbrandt and D. Powers, 'Characterization and evaluation of similarity measures for pairs of clusterings', *Knowledge and Information Systems*, vol. 19, no. 3, pp. 361–394, Jul. 2008. DOI: 10.1007/s10115-008-0150-6.

[45] D. Kirat, L. Nataraj, G. Vigna and B. S. Manjunath, 'SigMal: A Static Signal Processing Based Malware Triage', in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13, New Orleans, Louisiana, USA: Association for Computing Machinery, 2013, pp. 89–98, ISBN: 9781450320153. DOI: 10.1145/2523649.2523682.

[46] G. Laurenza, L. Aniello, R. Lazzeretti and R. Baldoni, 'Malware Triage Based on Static Features and Public APT Reports', in *Cyber Security Cryptography and Machine Learning*, S. Dolev and S. Lodha, Eds., Cham: Springer International Publishing, 2017, pp. 288–305, ISBN: 978-3-319-60080-2. DOI: 10.1007/978-3-319-60080-2_21.

[47] Y. Fang, W. Zhang, B. Li, F. Jing and L. Zhang, 'Semi-Supervised Malware Clustering Based on the Weight of Bytecode and API', *IEEE Access*, vol. 8, pp. 2313–2326, Jan. 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2962198.

[48] scikit-learn developers, *Cross-validation: Evaluating estimator performance*, 2019. [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html (visited on 01/05/2020).

[49] G. James, D. Witten, T. Hastie and R. Tibshirani, *An Introduction to Statistical Learning*. Springer New York, 2013. DOI: 10.1007/978-1-4614-7138-7.

[50] ClamAV, *LibClamAV*, 2020. [Online]. Available: https://www.clamav.net/documents/libclamav (visited on 07/05/2020).

[51] K. Wilhoit, *AutoIt Used To Spread Malware and Toolsets*, May 2013. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/autoit-used-to-spread-malware-and-toolsets/ (visited on 24/05/2020).

[52] J. Bennett, *Compiling Scripts with Aut2Exe*, Mar. 2018. [Online]. Available: https://www.autoitscript.com/autoit3/docs/intro/compiler.htm (visited on 06/05/2020).

[53] Python Software Foundatio, *Mapping Types - dict*, 2020. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#typesmapping (visited on 07/05/2020).

[54] 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)*, Dec. 2016, pp. 70–77. DOI: 10.1109/PST.2016.7906939.

[55] 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)*, Jul. 2016, pp. 684–691. DOI: 10.1109/IJCNN.2016.7727266.

[56] J. Lenoir, *Gunpack: Un outil générique d'unpacking de malwares.* Jun. 2016. [Online]. Available: https://www.sstic.org/2016/presentation/gunpack/ (visited on 04/02/2020).

Appendix A

Abandoned unpacking methods and features

Multiple ideas that arose from studying previous work, had to be abandoned during implementation of the proof of concept due to poor results or time constraints. This appendix explains why they initially were planned to be used and why they had to be abandoned.

A.1 Abandoned unpacking methods

Several different unpacking methods were tested. The performance and success-rate of most unpacking methods was found to be far below expectations, and most attempts had to be abandoned.
The UPX software can be used to statically unpack a PE-file that has been packed using UPX. This unpacking would however often fail, even on files that have been manually verified to be packed by UPX. It is suspected that small modifications such as modifying checksums, are made by malware authors to make unpacking more difficult. Using ClamAV for unpacking files that are identified as being packed with UPX, was almost as fast and resulted in a higher success rate. Unpacking of files with the UPX software, was therefore removed from the proof of concept.
The majority of the abandoned unpacking methods were based on generic unpacking. The findings of [37] and [38] indicated that the use of generic unpacking can improve malware triage significantly. Implementations based on these research projects, were therefore tested on a small set of PE-files that had been identified as being compressed/obfuscated with packers listed as supported in the descriptions of these implementations. The authors of MutantX-S [37] did not seem to release their implementation to the public, but a researcher at Airbus Group Innovations, made an open-source implementation named Gunpack [56] based on MutantX-S. A generic unpacker that was released to the public by the original researchers however, is PinDemonium [38].

Setting up the correct environments and building the software was tedious, and unfortunately the results from initial experiments disappointed too. As mentioned in [38], the success-rate of PinDemonium was significantly lower when tested on malware found in the wild. This indicates that malware found in the wild, is not simply packed once by one packer and released into the wild without further modification. Malware authors are likely recursively packing or obfuscating PE-files to make malware analysis more difficult.
In the experiments performed using Gunpack and PinDemonium, a tiny portion of the packed files were successfully unpacked. Because of the additional overhead, timeouts of 1 and 5 minutes were required for Gunpack and PinDemonium respectively. With no method of knowing whether a file will be successfully unpacked or not until the file has been attempted unpacked, a huge amount of processing power is wasted. Performing generic unpacking was simply not worth it, when considering the tiny amount of successfully unpacked files, and the huge waste of CPU-time that could have rather been used at dynamic analysis.
A final attempt at generic unpacking was performed with Unipacker. Unipacker emulated execution of PE-files and could therefore run in Linux environments. In cases where Unipacker was able to return a PE-file, it seemed to do so in less than 4 seconds, and a slightly low success-rate, could therefore be more acceptable. The success-rate of Unipacker was tiny, which again led to an unacceptable waste of CPU-time compared to the provided value.

A.2 Abandoned features

There were also promising features that unfortunately were not used in the proof of concept.
It would be interesting to study how suitable features based on the *Rich header* would be in divide-and-conquer clustering. Since the *Rich header* is not present on all PE-files, it would be more suitable in divide-and-conquer clustering than other clustering approaches.
The findings of [26] indicated that imphash performed better than *peHash*, but there might be situations where peHash can complement imphash. It would therefore have been interesting to see if *peHash* could complement other features being used.
Using a perceptual hash such as *dhash*, instead of a cryptographic hash like SHA-2 for creating hashes of icons was attempted, but the libraries used to create a *dhash* from an icon often failed at parsing the icon. By using both *dhash* and a cryptographic hash, it would have been possible to compare icons using both perceptual and cryptographic hashes, but due to time constrains, this was also abandoned.
For performing slow clustering, using *ssdeep* with the improvements proposed in [22], could have been a better alternative. Even though the time-complexity is identical to clustering with TLSH, the overall cost per comparison is smaller, and the method should therefore be able to scale slightly better.

The final feature that was abandoned, was *Machoc hash*. A *Machoc hash* is a control flow graph hash implemented in the polichombr framework. A control flow graph is a graph based on all potential paths an executable can traverse. Polichombr is a framework developed and used by ANSSI (National Cybersecurity Agency of France), and it was therefore assumed that the feature performed well. *Machoc hash* is not mentioned in related work, since the only papers published on the feature, are written in French and Italian. After initial experiments had been performed, it was concluded that the available implementations for extracting a *Machoc hash* from a PE-file, were far too slow for scalable malware triage. The available Ruby implementation typically spent 3 to 15 seconds on disassembling and extracting the *Machoc hash* of a single PE-file, on a single CPU core. An implementation combining Python and Radare2 was also available, but this was even slower than the implementation in Ruby. Unlike the other features mentioned in this section, the main reason for not including *Machoc hash* in the proof of concept, was the very slow feature extraction.

Appendix B

Complexity of clustering with distance-based fuzzy hashes

Clustering files with distance-based fuzzy hashes is significantly more complex than clustering based on equal values. Many iterations must be made, resulting in a high time-complexity. Listing B.1 shows how it is possible to cluster files based on TLSH hash in Python.

Code listing B.1: Clustering files based on TLSH distance in Python

```
def cluster_with_tlsh(file_to_cluster, all_files, tlsh_clusters, update_centroid):
  """
  Cluster a file based on TLSH distance.

  Parameters:
  - file_to_cluster:  The file to cluster. Must have the property tlsh,
        containing the calculated tlsh hash of the file.
  - all_files:        A dictionary of all files. All the objects must have
        the property tlsh, containing the calculated tlsh hash of the file.
  - tlsh_clusters:    A dictionary of TLSH clusters,
        where the centroids are used as keys.
  - update_centroid:  Specify whether the centroids should be updated
        after adding a file to a cluster (True / False).
  """

  # Use a distance threshold of 100 (see related work)
  THRESHOLD = 100

  # Best score is initially 101. If distance is 101 or
  # higher, the files are not sufficiently similar.
  best_score = THRESHOLD + 1

  # Reference to the TLSH cluster that matched this file the best
  best_centroid = None

  # Iterate over all existing clusters first
  for centroid in tlsh_clusters.keys():
    # Calculate distance to centroid of cluster
    score = tlsh.diff(file_to_cluster['tlsh'], centroid)
    if score < best_score:
      # If distance is less than threshold / currently closest,
```

```
      # set this cluster as the new "closest" centroid.
      best_centroid = centroid
      best_score = score

  if best_centroid is not None:
    # If a centroid was close enough for a match,
    # add this file to the matched cluster.
    tlsh_clusters[best_centroid]['items'].add(file_to_cluster['sha256'])
    file_to_cluster['tlsh_cluster'] = best_centroid
  else:
    # If no cluster matched, create a
    # new cluster with this file as centroid
    best_centroid = file_to_cluster['tlsh']
    tlsh_clusters[best_centroid] = {
      'label': None,
      'training_purity': 0,
      'items': set([file_to_cluster['sha256']])
    }
    file_to_cluster['tlsh_cluster'] = best_centroid

    # And then iterate over all files that are not in a TLSH cluster
    for other_file in all_files.values():
      if (other_file['tlsh'] is not None
          and other_file['tlsh_cluster'] is None
          and tlsh.diff(file_to_cluster['tlsh'], other_file['tlsh']) <= THRESHOLD):
        # If distance is less than or equal to threshold,
        # add the other file to the newly created cluster.
        tlsh_clusters[best_centroid]['items'].add(other_file['sha256'])
        other_file['tlsh_cluster'] = best_centroid

    # Attempt to label the newly created cluster
    # (in case some of the matching items had labels)
    label_cluster(tlsh_clusters[best_centroid])

  if update_centroid:
    # Update centroid if specified to do so
    update_tlsh_centroid(tlsh_clusters[best_centroid])
```

Appendix C

Published open-source software

This appendix describes the open-source software that has been developed as part of this research project.

C.1 Proof of Concept

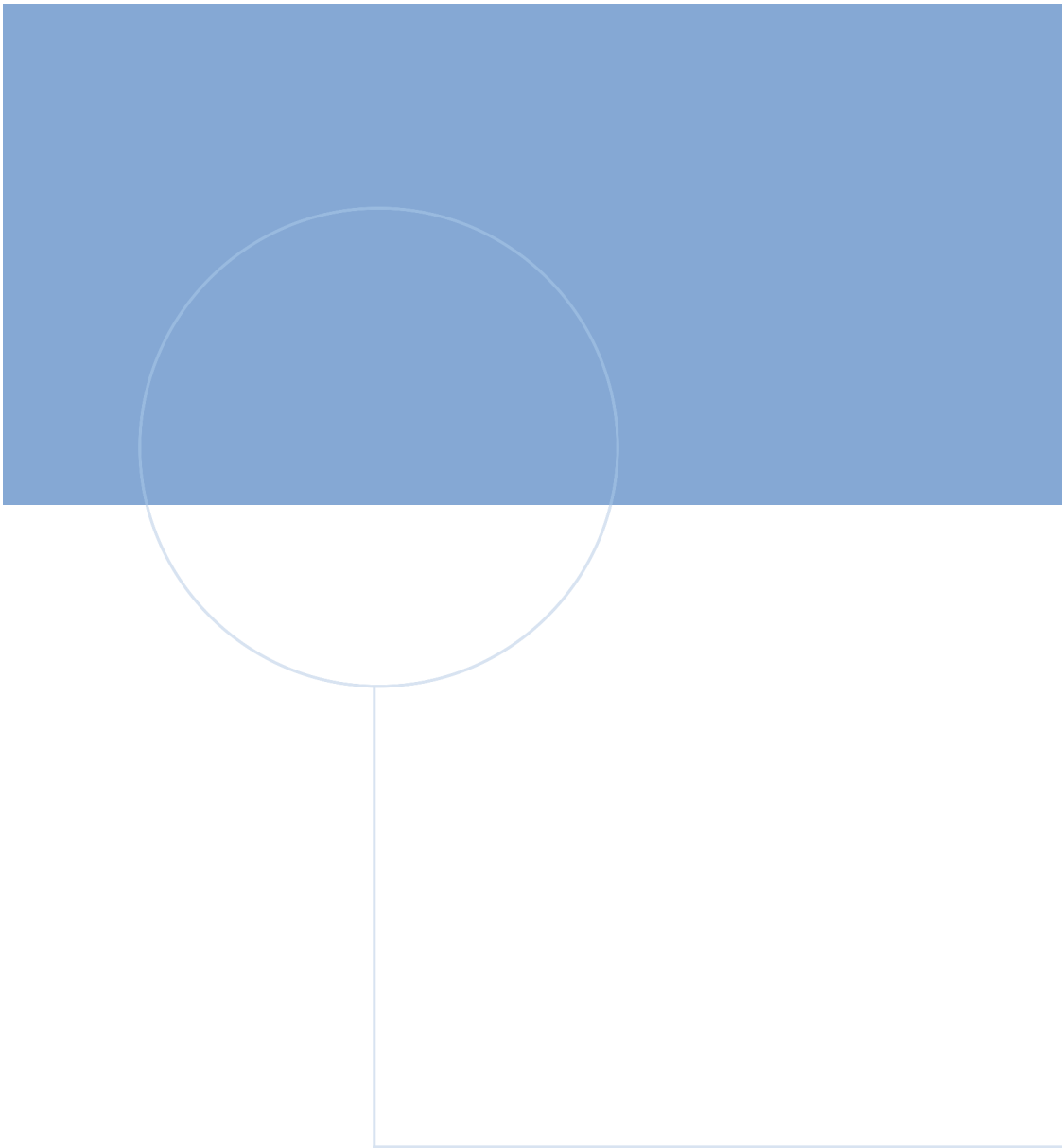
The proof of concept has been implemented to compare DAC-clustering to more naive clustering methods, but contains code that might be useful for others performing experiments on triage of PE-files or wishing to verify the results. The proof of concept is described in detail in section 3.2, and published to Github: `https://github.com/57ur14/DAC-clustering-PoC`

C.2 Python module for extracting the icon of a PE-file

A Python module for extracting icons from PE-files had already been developed and published as open-source software. The module was only supported by the now deprecated Python 2 and would often crash due to unhandled exceptions when parsing files. A fork of this module was made, which is more resilient and runs under Python 3. This fork is published to Github: `https://github.com/57ur14/pefile-extract-icon`

C.3 Ruby script for extracting the Machoc hash of a PE-file

Although the proof of concept ended up not using *Machoc hash* to cluster files, others might find the developed script useful in the future. The script is mostly based on the Polichombr framework, but allows extraction of *Machoc hash* without invoking the whole framework: `https://github.com/57ur14/machoc`



NTNU

Norwegian University of
Science and Technology