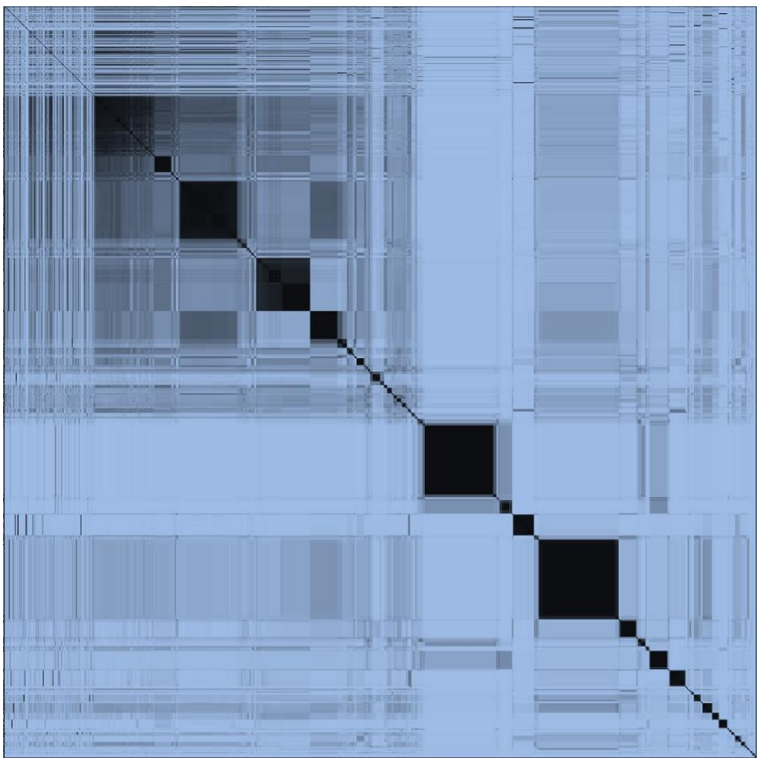
AALBORG UNIVERSITET
STUDENTERRAPPORT

Malware cluster analysis

based on Windows API function call sequences



Peter Grinderslev Stegger

Master Thesis
Master of IT, 4th semester
Aalborg University

30/04/21

Table of content

1 Abstract

This project examines how malware samples can be analysed to find malware clusters based on sequential behaviour in sequences of API function calls. The analysis is conducted by running malware samples through a Cuckoo sandbox hosting a virtual Windows machine. The reports generated is processed in a Python application to extract the API function calls as sequences of API names. Three data sets are created from the API sequence calls. All sets are cut and only the first 200 API function calls are included from each malware sample. In addition to this the second dataset have select API function calls filtered out, so only the most significant calls are included. The last dataset is like the second, but repeated sequences of API function calls are collapsed.

Calculation of distances between API call sequences are done with the Levenshtein distance and transformed into a ratio by dividing with the longest sequence length. The datasets are clustered using the OPTICS and hierarchical clustering algorithms. The silhouette score coefficient is used to evaluate the fitness of the clusters and distance matrixes are plotted to allow for visual evaluation as well.

The project concludes that it is possible to cluster malware by looking at the sequences of API function calls. Optimal clusters are found using the OPTICS algorithm on the third dataset. The best result is a mean Silhouette score of 0.8 disregarding noise and 0.6 including it. This shows that highly cohesive clusters of malware can be found using the proposed approach.

The project shows a potential in continuing research into temporal analysis of malware in general, but also specifically when considering API function calls.

2 Reading guide

The report is structured with an introduction followed by a state of the art, before the problem statement is introduced. This is order is chosen to point out the research gap before the problem is stated.

The theory section explains the theory chosen for the analysis. The section is presented before the analysis design section is. This order is chosen because the choice of theory affects the analysis design.

The analysis results section presents and evaluates the findings of the analysis.

The discussion highlights the issues with the analysis design and gives some ideas to how they can be solved.

Finally, a conclusion is given, followed by a perspectivation on how the research can be carried on from here.

3 Introduction

Malware is a threat towards private persons as well as professional corporations jeopardizing confidentiality, integrity and availability of data and services.

Every day more than 350,000 new malware[1] is registered on the internet (AV-Test, 2021). This is only the tip of the iceberg. One can only guess as to how large the number would be if all malware were registered. But even without this knowledge the number speaks volumes.

This volume is a problem because defeating new malware requires analysis. When all malware is analysed separately the process is extremely time consuming.

Malware is written by malware developers but most often it is not written from scratch (Calleja, Tapiador, & Caballero, 2016) (Ollmann, 2008). The developers continuously build existing malware into newer and more improved version. The different variants of malware that stems from the same source code is often defined as families or strains. Similarities can be detected between different versions within the same family. Clustering newly detected malware with known strains of malware will speed up the malware threat analysis process.

This project explores how malware samples can be clustered based on temporal analysis of API function calls.

[1] *new malicious programs (malware) and potentially unwanted applications (PUA)"*

4 State of the art

Malware analysis can be divided into several approaches as seen in Figure 1. Statical analysis where the suspected malwares binaries are analysed without being executed. Signature based detection is a kind of statical analysis though it is not represented in Figure 1. Dynamic analysis that requires the malware to be run to analyse its interactions with its environment. Finally, some approaches are hybrids that combines static and dynamic analysis. A fourth approach is analysis of the volatile memory of the operating system where "*… the memory of the target machine is dumped to obtain a memory image …"* (Sihwail, Omar, & Ariffin, 2018) so the analyst can *"… analyse the memory image looking for malicious activities…*" (Sihwail, Omar, & Ariffin, 2018).

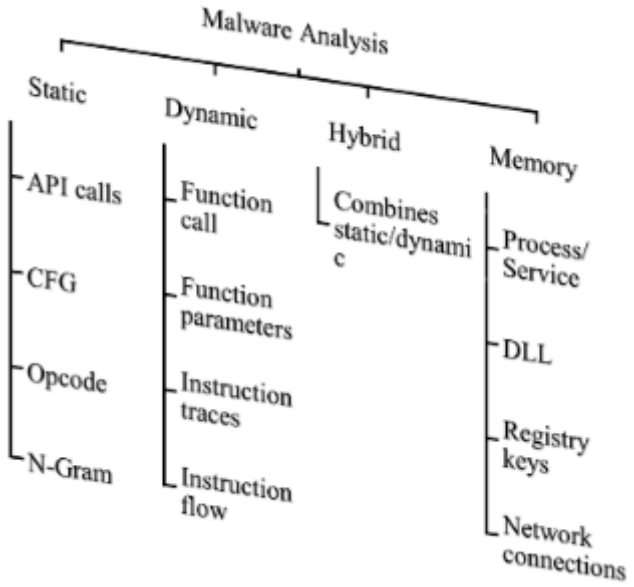


Figure 1 Malware analysis techniques and features (Sihwail, Omar, & Ariffin, 2018)

4.1 Static analysis

The most widely spread form of malware detection is signature based. This is used by most of the antivirus software and works by extracting a "*… unique signature from captured malware file and use this signature to detect similar malware*" (Sihwail, Omar, & Ariffin, 2018). File hashes or byte sequences can be examples of signatures used to identify malware. It is an efficient approach against common types of malware, but it requires an updated signature database. This requirement makes it inefficient against new forms of malware since malware not in the database will not be recognised (Damodaran, Di Troia, Visaggio, Austin, & Stamp, 2017). However, once a new malware has been identified keeping a signature of it can ensure rapid detection. But to come up with signatures for new malware, new malware-suspects must first be analysed to determine if it is malicious or benign. For clustering of malware strains using a hash signature might prove ineffective since small changes in the malware binary can lead to completely new hashes.

4.2 Dynamic analysis

Dynamic analysis of malware is when malware under execution is observed. To do this several analysis techniques can be applied. Recording network traffic to and from an infected host machine is one approach. Running the malware on a host machine and recording its interactions with the operating system is another.

Others point to the risk of not observing all relevant information due to the high complexity of capturing all relevant events. This is being used as an argument for why dynamic malware analysis always should be accompanied by dynamic code analysis like debugging (Higuera, Aramburu, Higuera, Urban, & Montalvo, 2020).

A common issue with sandboxes used for dynamic malware analysis is also that they only explore one execution path before the test is concluded (Higuera, Aramburu, Higuera, Urban, & Montalvo, 2020). The issue here is that the malware might employ evasion techniques (Afianian, Niksefat, Sadeghiyan, & Baptiste, 2020), or only perform malicious activity under certain conditions.

A lot of research in dynamic analysis looks at API function calls. This makes sense because API function calls reveals high level information about the behaviour of the malware. Different takes on this data have been done in the field of research. Some research treats the individual API function calls as features. Then clustering of the malware can be done based on statistical information about individual API function calls. Other research however has "*observed that sequences of function calls generally describe an action better*" (Széles & Coleşa, 2019). They split the API call sequences into N-grams and used those as features. Furthermore, API function calls were grouped into levels of importance, based on system impact, allowing the researcher to focus on the most significant API function calls. The clustering in this case was performed using the k-means algorithm, and the similarity measure was a comparison of n-gram frequency. They concluded that N-grams where N varied from 5-7 worked best, and that a mid to high level significant API function calls gave the best trade-off between noise and information. The lower value n-grams where presumed ineffective because of the loss of semantic information. This study is like what (Hansen & Larsen, 2015) did where they also looked at frequencies of API n-grams in malware.

Clustering analysis on data from static analysis seems to be mostly OpCode n-grams. Here sequences of OpCodes are used to make the n-gram to create classifiers. The basic idea is that "*different versions of the same application or different generations of the same malware family will share similar code*" (Oprisa, Cabâu, & Pal, 2018). But n-grams are only a small subsequence of the malwares full behaviour. It would be interesting to consider the whole sequence. In a recent survey the authors concluded that "*… temporal analysis was found to be very effective in many domains, especially in the biomedical informatics domain. Such an approach can shed new light on malware behaviour by discovering frequent time-oriented patterns that differentiate between malicious and benign behaviour. Since every piece of data collected during dynamic malware analysis has a timestamp, we believe temporal analysis has great potential in creating advanced machine-learning solutions for the malware detection and categorization tasks*" (Or-Meir, Nissim, Elovici, & Rokach, 2019). This conclusion, together with the research results from analysing API function calls, indicates an interesting area of research that I do not find to be fully covered.

4.3 Malware evasion techniques

The different analysis methods face different challenges from the malware. The presence of software like Wireshark[2] (Wireshark Foundation, 2021) on the host machine can trigger evasive behaviour within the malware. Some malware shuts down if it runs in a virtual environment. Triggering evasive behaviour often causes the analysis to fail. But the evasive measures are part of the malware functionality, and certain types of malware will have the same defensive measures. As

[2] Wireshark is an application that captures network traffic and stores it in a file. It also provides functionality for analysing the network traffic.

evasive techniques are a crucial functionality within malware, is not a problem if it shows up during analysis. It is just important that the evasive techniques are not triggered.

Malware can make use of many evasion techniques that makes it harder to analyse them. A broad classification of evasion techniques could be the ability to detect debuggers, detection of antivirus installed and execution artifacts, as well as the detection of virtual machines, emulators, or sandboxes (Herzog, Tong, Wilke, Van Straaten, & Lanet, 2009). In more recent year's complex systems such as "*MalGene, an automated technique for extracting analysis evasion signatures*" (Kirat & Vigna, 2015), has given a more detailed view on different evasion signatures. *"MalGene leverages algorithms borrowed from bioinformatics to automatically locate evasive behaviour in system call sequences"* (Kirat & Vigna, 2015). By analysing 2810 evasive samples with MalGene they were able to group them in to 78 evasion techniques. Their approach was to run the samples in different environments, where one environment would trigger an evasion technique, and another would not. By aligning sequence calls, an *evasion point* could be determined, by looking for a deviation in the environment that triggered it. Once the point was determined, a signature of the evasion technique was extracted by using multiple steps to reduce the amount of insignificant API function calls. Then the signatures were alle clustered using hierarchical clustering followed up by manual clustering.

Dynamic analysis is being countered by detection of the analysis environment. This could be detecting the presence of known analysis tools or detecting if the environment is virtual. In fact,"*… observations attest that evasive behaviour is mostly concerned with detecting and evading sandboxes*" (Afianian, Niksefat, Sadeghiyan, & Baptiste, 2020) Malware can also use logic bombs, and other means of evasion.

A recent survey concluded that defenders against malware evasion techniques were pursuing four major strategies: "*Reactive approaches, more transparent analysis systems, bare-metal analysis, and several endeavours toward more generic approaches, namely, path exploration*" (Afianian, Niksefat, Sadeghiyan, & Baptiste, 2020). They concluded that the first three approaches were mostly effective against specific types of evasion techniques, excluding a portion of known evasion techniques. Furthermore, they concluded that evasion techniques could be effective against evasion techniques that are detection-independent, such as control flow manipulation, lockout evasion, and fileless malware.

Hardening of sandboxes has been a key component in some analysis designs. The strategies for during this include hiding all traces that the machine is virtual by uninstalling guest-additions, installing commonly used software like Java and PDF readers, and changing various setting to make the virtual machine look used (Muhovic, 2020). Other also point to simulating end-user behaviour, implementing time jumps to simulate passing time, editing registry values, and adding common artefacts like files (Mills & Legg, 2020). They found that employing these anti-evasive techniques scored higher under test. Showing that to efficiently do dynamic analysis of malware using anti-evasive techniques is a must.

4.4 Summary

It can be concluded that researchers have many approaches to malware analysis. Furthermore, it can be concluded that the malware authors make all approaches more difficult with malware evasion techniques. To do clustering analysis all approaches hold promise, since all approaches have been proven to identify malware with great success. However, some research (Or-Meir, Nissim, Elovici, & Rokach, 2019) points to a research gap in the field of temporal analysis of malware. It is

this gap I wish to contribute to by applying temporal analysis on malware API function call sequences. To record these in temporal, or sequential order, it is necessary to do dynamic analysis. It is also necessary to apply anti evasive measures to generate good data.

5 Problem statement

The state of the art is very well developed and shows good results in doing binary classification, and clustering where a ground truth is known. It also shows that API function calls is a good feature to use for clustering and binary classification. As a feature it works both with individual API function calls as features, but also with API n-grams as features.

I found that doing temporal analysis on data about malwares execution is encouraged as a research area. The encouragement is based on the effect the approach has had in domains like biomedical informatics (Or-Meir, Nissim, Elovici, & Rokach, 2019) (Zhao, Papapetrou, Asker, & Boström, 2017).

Based on this I want to answer the following:

> **How can malware samples be analysed, to find clusters, based on behaviour found in sequences[3] of API function calls?**

The analysis will capture the behavioural pattern of malware using dynamic analysis. Using the malwares samples API function calls in sequential order as my data base. The results should show if sequential analysis can have a place in malware analysis.

[3] While the analysis is still temporal, I think the term sequential is more appropriate. This is because the API function calls will only be considered in their sequential order, with no consideration to the time at which they occured.

6 Theory

In the following I will address the theories needed to solve the problem of grouping malware by behaviour. The chapter helps to make informed choices about the design of the analysis. In general cluster analysis is used to identify groups of objects within a dataset. It is a kind of explorative data analysis and is great for explorative analysis large data sets. Cluster analysis is used on all kinds of domains from malware analysis, to deciding on target demographics for a marketing campaign. The same clustering algorithms works across different domains. This is because the algorithms are unbiased, and only consider the data fed to them. This emphasises the importance in selecting good features.

6.1 Feature selection

It is important to select the most discriminative features to get good analysis results. Since there are so many dimensions to consider when analysing malware, it can be overwhelming but also infeasible to consider them all. Not only does it increase the computational overhead and the sheer amount of data collected. It also makes the analysis more complex and less efficient. With the aim of differentiating malware into strains the features that best provide separability between them must be selected.

Given that malware has been successfully detected using models based on API function calls it makes for an obvious feature candidate. It also makes sense that API function calls would reveal information about the nature of the malware, since API function calls are the way malware gets information of its surrounding operating system. To get better results the dataset must be reduced to only include the most important API function calls. To cope with this the data will be processed like (Hansen & Larsen, 2015) did. Here all unimportant API function calls were filtered out first by only considering API function calls there were more dominant in malware compared to API function calls recorded when running benign software.

Other issues with the collected data are the huge variety in API sequence call lengths. This can be seen simply in the distance in size of the generated malware reports that is spread evenly from just a few kilobytes to 1GB for the largest. Even if some similarity exists between samples in either end of the size spectrum the distance would be enormous. This can cause an issue if the selected distance metric has a bad time complexity. To cope with this issue, I will only analyse the first 200 API function calls of all samples.

Thirdly repeating patterns are collapsed "*to retrieve eventual hidden similarities, the sequence is modified so that it gives the succession of actions performed without taking care of their frequency. It is done by removing the repetition of the same API called in a row*" (Pirscoveanu & Hansen, 2015).

The problem with API function calls is that they are categorial data, and this makes it difficult to calculate distances between. So, a way of calculating the distance between two sequences of API function calls must be expressed. Once that has been done, any metrics-based clustering algorithm can be used on the data.

6.2 Distance metric

Choosing a good distance metric is important when doing cluster analysis and can have huge impacts on the results. The analysis process is also affected highly by the choice of distance metric. Finding clusters involves calculating the distances between all data objects. Choosing a distance metric with a poor time complexity will drastically decrease analysis speed.

Distances between data objects can be many things. Naturally, an actual distance like kilometres is a good and comprehensible distance metric. When looking at sequences of API function calls it is more difficult to express a good comprehensible distance metric. Obviously, the metric must be meaningful when considering the domain, but it must also follow these four rules:

1. Symmetry:	$D(x_i, x_j) = D(x_j, x_i)$
2. Positivity:	$D(x_i, x_j) \geq 0$
3. Triangle inequality:	$D(x_i, x_j) \leq D(x_i, x_k) + D(x_k, x_j)$
4. Reflectivity:	$D(x_i, x_j) = 0 \; if \; x_i = x_j$

Figure 2 Distance metric rules

The rules ensures that the distances between data points are comparable and can be used by the clustering algorithms. Furthermore, the triangle inequality measure is important since it basically enforces the rule that the shortest path between two points is a straight line. If the distance metric does not follow this rule, it could cause cyclomatic issues when doing the actual clustering.

Normal Euclidean distance calculations cannot be used directly on categorial data. I have chosen to use the Levenshtein distance instead (Levenshtein, 1965). The basic idea of the Levenshtein distance is to calculate how many inserts, edit or delete operations are needed to change one sequence of symbols into another. The metric is well known and can be applied to a range of domains (Dogan & Oztaysi, 2019) (Zhao, Papapetrou, Asker, & Boström, 2017). Besides being applicable to a range of usages, the metric also obeys the above-mentioned rules for distance metrics. This makes it suitable for cluster analysis.

The Levenshtein distance is probably most known for being used on text, which is a sequence of characters. But the distance formular works for any sequence of comparable symbols. If for example we should change “cat” to “hat” it would be necessary to do one edit operation, changing the ‘c’ to an ‘h’. The distance would here be one. Considering the distance from the sequence “malware” to “software” it would be necessary to do one insertion and three edits to make the distance four.

In this projects domain the symbols would be API function calls rather than letters. However, the approach remains the same. It makes good sense because if a malware author adds just a few API function calls to a new version of his creation then the distance would equate to the amount of inserted API function calls. But if two pieces of malware does the same API function calls, but in vastly different order, then the distance is larger because the sequential ordering of the symbols matter. This makes the distance metric highly suitable for the research question in this project.

The major drawback of the Levenshtein distance its time complexity. For two sequences of length n the time complexity is $O(n^2)$ typically referred to as quadratic time. This disastrous time complexity can have a major impact on the analysis process. Given long enough API call sequences it can drastically reduce the number of samples I will be able to process. This reinforces the choice of reducing all API sequences to a length of 200.

The order of the API function calls made by malware, is a result of the sequence in which the code is written. So, the deterministic behaviour of software will result in equal API function calls if executed under identical conditions. If the conditions are changed however, the conditional statements based

on these changes can affect the sequence of API function calls. This emphasises the need for an identical execution environment to perform a successful analysis using the Levenshtein distance as a distance metric. This must be considered when designing the analysis process.

6.3 Clustering techniques

Generating cluster can be done with a vast number of different clustering algorithms. Like the c-means algorithm where centroids are defined by object density, and subsequently objects are assigned a centroid based on proximity. There are multiple clustering algorithms that works with centroids, and some research has combined this to make multi-centroid cluster analysis (Oprisa, Cabâu, & Pal, 2018). But unless objects from dataset are chosen as the centroids, like in the k-nearest-neighbour algorithm, it requires the features to be numerical. For categorial data as in this domain suitable clustering algorithms must be selected.

6.3.1 Hierarchical clustering

Hierarchical clustering looks at linkage, and not centroids. This makes sense when analysing data that cannot be measured on some absolute scale. Here trees, or dendrograms, are formed using different linkage methods. The trees can be creates using either a bottom up (agglomerative method), or a top-down approach (divisive method). The difference is that in the bottom-up approach all objects are considered their own cluster at the start. The analysis approach will then cut down the number of clusters by merging similar clusters until trees reach a desired size. The bottoms-up approach does the opposite to reach the same result. So here all objects belong to one cluster from the start.

The analysis will then consist of dividing the cluster into separate clusters until only singleton clusters exist. "*Hierarchical clustering has the distinct advantage that any valid measure of distance can be used. In fact, the observations themselves are not required; all that is used is a matrix of distances*" (Spiegel, 2015). Getting the final number of clusters can be done by deciding on a minimum acceptable linkage. If displayed as a typical dendrogram this would be seen a horizontal cut. The downside of hierarchical clustering is the assumption that all data objects belong to a cluster. If the data set contains a lot of noise the algorithm is not good. Furthermore, the linkage method is also important. If single linkage is used it means that individual data points will be clustered together based on their single one closest neighbour. This approach could be thought to create chains where A is clustered with B first, and following that B is clustered with C. This creates the cluster ABC, but A and C may be direct opposites. If this pattern continues the cluster might end up having some endpoints that are highly unrelated. Therefor experimentation with linkage method would be a good idea. However, this experimentation is not within scope of this project. An average linkage method is there for chosen for this project. Well knowingly that different linkages will give different results.

6.3.2 OPTICS

A different approach to clustering that also does not make use of centroids is OPTICS (Ankerst, Breunig, Kriegel, & Sander, 1999). It works very similar to DBSCAN that "*… uses a simple minimum density level estimation, based on a threshold for the number of neighbours, minPts, within the radius ε(with an arbitrary distance measure). Objects with more than minPtsneighbors within this radius (including the query point) are considered to be a core point*" (Schubert, Sander, Ester, Kriegel, & Xu, 2017). Considering the domain under analyses here OPTICS is a good method since it allows arbitrary distance measures. Furthermore, OPTICS requires some density of data points before it will create a cluster. Changing the minimum density level will allow for more closely coupled clusters. Having a minimal density level will consider more data points to be noise. Within the current domain noise would be any malware sample that does not belong to any specific cluster.

Deciding if two data points are neighbours depends on the threshold value for the distance metric. Setting a high threshold means that a longer distance between data points is acceptable, and visa versa. But this is a difficult task, since *"… in an ideal scenario, there exists domain knowledge to choose this parameter (radius) based on the application domain. For example, when clustering GPS locations, a domain expert may decide that objects within 1 kilometre should be neighbors and we then can choose the appropriate distance function and set ε accordingly*" (Schubert, Sander, Ester, Kriegel, & Xu, 2017). For this project, a big part of the problem is in fact figuring out how well the proposed distance metric works. Some experimentation will be required to make good clusters. As a result, it is also very important to have a good way of evaluating the results from any of the proposed clustering algorithms.

6.4 Evaluation techniques

After the clusters are found it is important to evaluate how fit they are. Without any form of evaluation it is impossible to conclude anything about the outcome of the analysis.

Choosing an appropriate evaluation technique relies primarily on the presence of *a priori* knowledge. Having that knowledge will make for a ground truth. That can be used to evaluate if the clusters found fits the ground truth and then validate the analysis. This is called external validation. Since no prior knowledge is available for this project an external validation technique is not an option.

Internal evaluation on the opposite is evaluating clusters "*… in terms of the inner structures of the datasets themselves, rather than a priori knowledge*" (Abu-Jamous, Fa, & Nandi, 2015). A good coefficient for measuring how well a clustering algorithm has worked is the silhouette coefficient. The Silhouette coefficient for a single sample s is given as stated in figure 3:

$$s = \frac{b - a}{\max(a, b)}$$

Where:

a is the mean distance between a sample and all other points in the same cluster

b is the mean distance between a sample and all other points in the next nearest cluster

Figure 3 The Silhouette coefficient score

Calculating this formular gives a result of -1 for incorrect clustering, basically saying that the sample is more closely aligned with the next cluster over. A value of +1 signals the opposite, and a value of 0 indicates that the clusters overlaps. In general, the higher the scores, the better the clustering has worked. Most often cluster algorithms generate scores in the range of 0 to 1. The drawback of the Silhouette coefficient is that it generally does not perform the best on density-based clusters like OPTICS. But the fact that the coefficient gives some estimate of the success means that I will use it as an indication of success none the less.

I will also plot the results of various runs of the clustering algorithms to see how fit the clusters are. Since the two clustering algorithms takes different parameters, they must be evaluated differently.

The OPTICS algorithm takes one important argument that is the minimum number of samples required to form a cluster. Whereas the hierarchical algorithm takes the desired number of clusters. The output of both algorithms will naturally be the clusters for the samples. It is important to evaluate the clusters found in comparison to the fitness gained – here measured with the silhouette

score. If the score can be continuously increased by adding clusters, it might be a sign of over-fitting. At an extreme this could result in every sample being in individual clusters. Therefor I will use the elbow method to visually validate the analysis results. This will help me to spot where increasing different parameters starts to give diminishing returns. Since the OPTICS algorithm labels some of the samples as noise this will also be taken into evaluation since I would consider less noise better.

Visualising the data can also help to validate the clusters found. In a feature space where Euclidean distances makes sense it is easy to visualise the data. Given a simple feature space of 2 dimensions the individual data points can be plotted in a scatter plot, and it might be possible to see the actual clusters. However, with categorial data this does not work. To visualise the clusters, I will order the distance matrix based upon the clusters found, to see if a crisp pattern will show up. This can be done by plotting all the malware samples into a chart with all samples plotted in same positions alongside both the x and y axis. To this I am using MatPlotLib (Hunter, 2007). In the intersection between two malware samples their distance is plotted as a colour. The colour signals the distance between the samples, as displayed in the colour-bar to the right.

Since the Levenshtein ratio is used the max distance is one, and the shortest is zero. Zero is black, and one is white. If the malware is just plotted in randomly a noisy pattern will show up. When sorting the distance matrix by clusters, they will show up along the diagonal. Below is seen two examples. The clusters in the example are formed using two different clustering algorithms, yielding different matrices:

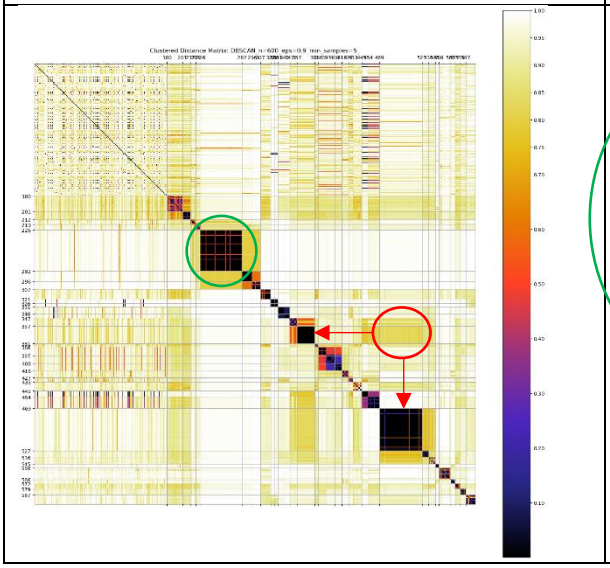
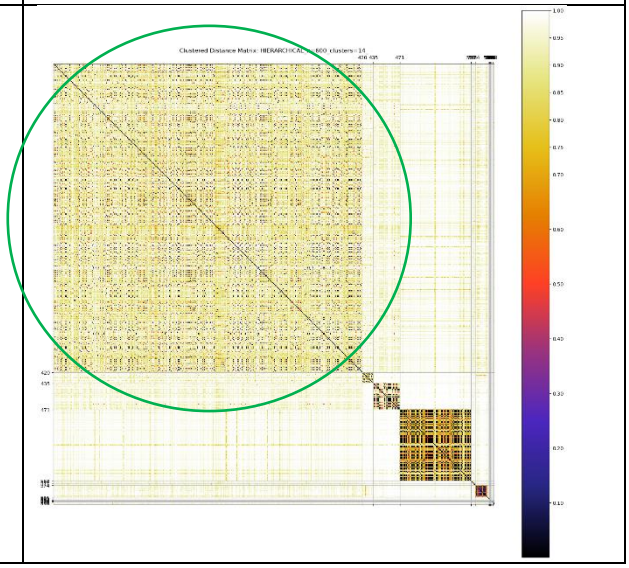
Distance matrix with clear and dense clusters	Distance matrix with clear but not so dense clusters
	

Figure 4 Sample visualisations of distance matrices

In the figure above to the left, it is clear to see the dark clusters of malware samples (Like the one highlighted by a green circle). The right matrix has a large cluster (marked by a green circle), but its colouring Is not dense. This means that the cluster is large, but its samples are not very similar.

Inter-cluster relationships can also be seen. If a dark shade is visible in the area where two clusters intersect (marked by a red circle pointing to the intersecting clusters), it is an indication of similarities between clusters. Given the domain it means the clusters share some API function calls sub-sequences. To truly understand what the shared functionality is, it would be necessary to do a deeper analysis of the intersecting clusters.

If white coloured bands form a cross centred on a cluster, it shows that the malware in the cluster, is dissimilar to other clusters (As seen in the green circle in the left example). The lighter coloured the cross, the more dissimilar the cluster is to any other clusters found.

The validation technique can be used to visually confirm the found clusters, but also to see the effect of changes made to the distance calculation, by comparing different results. Making it possible to compare the changes made during analysis and evaluate the best outcome.

6.5 Summary

A good foundation in theory is important to create an informed analysis. The foundation for the analysis in this project is to apply processing of API function calls as seen in previous comparable projects, with the aim of reducing complexity, and revealing more distinct functionality.

A tried and tested distance metric that analyses full sequences, and still adheres to the distance metric rules like triangular equality must be used to achieve the goals of the project. Here I have chosen the Levenshtein distance.

Many different clustering algorithms have been created throughout time. Picking the right one is a matter of matching its capabilities and restrictions to the project at hand. This project is not suitable for centroid based algorithms. OPTICS and hierarchical algorithms are both suitable for the domain. Both are chosen to do a comparison to find the best match for the project.

7 Analysis setup

Devising a good analysis setup is imperative in reaching good results. For this project where reduction of workload is a parameter for success it is important that the analysis setup considers efficiency. Furthermore, the setup must be created so that it introduces the least amount of error sources.

Following is a description of how I will conduct the analysis of the malware. Overall, the analysis process will consist of several phases that feed into each other, as seen in the figure below:

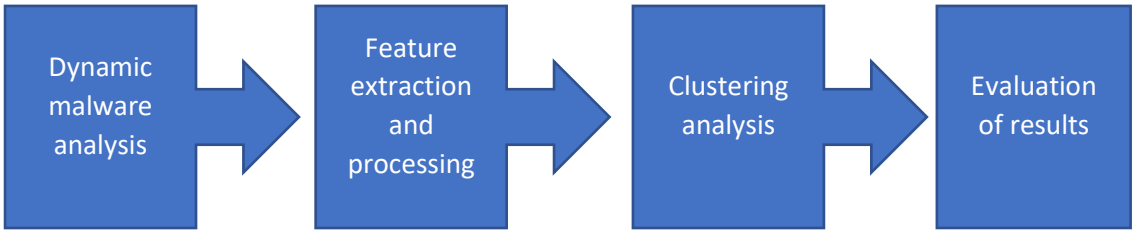


Figure 5 The analysis process

The first phase is a dynamic analysis of a selection of malware samples. The purpose of the phase is to generate analysable data about the malware samples. The second phase processes the data generated from phase one, making it suitable for further analysis. The preparation of the data will include calculating a distance matrix, that is used to perform the clustering analysis in phase three.

Based on the labels generated from the clustering analysis, the results will be analysed in the final phase. The analysis includes automated and manual steps. Following is a detailed description of the individual phases. The

7.1 Data collection

Normally collecting malware data is a tricky process. Malwares evasive behaviour is part of this reason. Furthermore, organisations tend to try and get rid of malware, not store it for later. Thus, the malware researcher has different options. One option would be to setup a honeypot which is a deliberately vulnerable machine. These machines are typically isolated in networks of no real production value but will still be connected to the Internet. This approach is time consuming, and results may vary depending on how much malware the honeypot attracts. It is not possible to predict how many malware samples that can be collected from the honeypot.

Alternatively, collection of malware can be obtained through collaborations with different organisations. This could be an organisation like VirusTotal that offers a "*free service that analyses files and URLs for viruses, worms, trojans and other kinds of malicious content… (their) goal is to make the internet a safer place through collaboration between members of the antivirus industry, researchers and end users of all kinds*" (VirusTotal, 2021). Through this collaboration I have acquired a ranges of malware samples collected from 2017 – 2020, and these will be used as data objects in the analysis. The dataset is quite large and contains hundreds of gigabytes of compressed malware binaries. I will only use a limited portion of the samples due to the projects limited time and computational resources.

7.2 Dynamic analysis

Cuckoo Sandbox (Guarnieri, n.d.) will be used as the malware analysis system. Cuckoo "*is an advanced, extremely modular, and 100% open-source automated malware analysis system with*

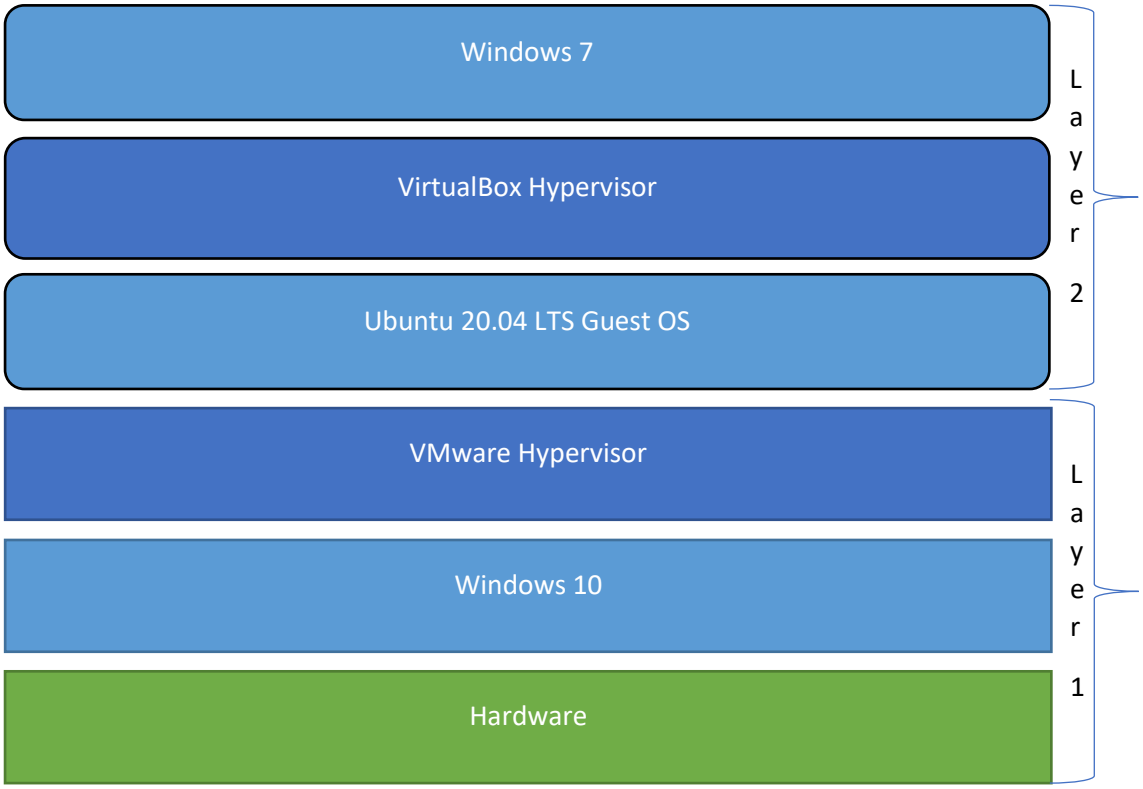
infinite application opportunities" (Guarnieri, n.d.). Using a readily available system has some benefits over creating a custom solution. Cuckoo allows me to go straight to the analysis phase, without first developing a system that does the same. Considering the scope of the project this is important[4]. Secondly Cuckoo comes packed with many tools for analysis that can capture the data necessary to conduct the desired analysis.

Cuckoo works by utilizing sandboxing. Sandboxing is a way of compartmentalizing runtime environments. Cuckoo uses virtualization where a fully operating system is hosted in a virtual environment. A key component in virtualization is the hypervisor which is the software that the virtual operating systems runs on. The hypervisor is basically an abstraction of physical hardware. A hypervisor can host multiple guest operating systems of different types, like Windows, Linux variants and Android[5]. This means that hypervisors are very versatile to use in malware analysis where it might be necessary to analyse samples on different operating systems.

Hypervisors come in two version, Type-1 ad Type-2. Type-1 hypervisors are bare-metal which means that they run directly on the host machines hardware. Type-2 are hosted which means that they run as software on an operating system running on the host machine. Seen from the guest operating systems perspective it makes no difference which type of hypervisor that is used. However, there is less overhead on a Type-1 hypervisor since there is no host operating system to fight with over resources. In my setup I am using a Type-2 hypervisor to run a single virtual machine. The drawback is that installing a guest operating system on a hypervisor requires a large amount of disk storage. Most hypervisors support dynamic allocation of storage meaning that a virtual machine can be created with a 100 GB hard drive without taking up more space than what is installed on the disc. For my test I have a laptop with a 450 GB hard drive available. This limits the number of virtual machines I can execute, and also the amount of malware I can have on disc at a time. The dataset from VirusTotal consist of 100's of GB data.

To further separate my host machine operating system from the malware samples I host my Cuckoo installation inside a virtual machine. This creates a setup that does require some configuration to run properly.

[4] The specific limitations in mind are manpower and time. Here the scope is one man, me, and part time.
[5] Android is a variant of Linux and as such it's a bit redundant to mention it here. However it felt appropriate due to the special status it holds as the worlds most widely used smartphone operating system.



Basically, it can be described as seen in the illustration above. On the host machine Win 10 is installed as the operating system, and VMware is installed as a Type-2 hypervisor. I call this part of the setup layer 1. On this layer an Ubuntu 20.04 LTS virtual machine is installed. This virtual machine hosts another Type-2 hypervisor, this one is a VirtualBox installation. I call this part of the installation layer 2. The second layer has Cuckoo installed next to the VirtualBox. The setup could be done using only 1 layer, but then Cuckoo would have to be installed on the host operating system. This would greatly increase the risk of getting infected with malware. So, the added layer further compartmentalizes my host operating system. Given access to a dedicated testing environment the setup could have been done differently with many added benefits.

Zooming in on layer two of the test setup it can be seen how cuckoo is placed within the Operating system next to the hypervisor. Cuckoo works by interacting with the hypervisor. Basically, the interaction consist of starting the virtual machine at a predefined snapshot. Secondly Cuckoo will submit the malware sample using the hypervisors interface, and run it in the virtual machine. During its execution time multiple information about the malware will be collected by an installed Python agent. The agent communicates over the network back to the Cuckoo host.

Therefor it is important that proper networking is setup on the Ubuntu host. I have set up a host-only adapter for the internal network. This connects the host and the guest allowing for network communication. To achieve a truly realistic test system full internet connectivity would be needed. But this could result in spreading the malware further to other machines which could have all kinds of negative results for the people that depends on the machines affected. However, for the sake of my analysis it would give more insight into the malware behaviour. Having no internet connectivity could also make the malware self-terminate giving no insight at all.

To deal with this I use InetSim (Hungenberg & Eckert, 2021) to simulate the internet. InetSim can emulate most known protocols and will respond to queries thrown at it. Naturally, it cannot figure out the data to transfer, but it can craft a response that would look normal. Hopefully normal enough to trick the malware into proceeding as it normally would.

Other tweaks have been done to the guest operating system to make it more vulnerable to malware infection. Inspired by the best literature (Muhovic, 2020) I could find on the topic I have done the following to create a realistic environment to run the malware in:

1. **Removed VirtualBox Guest Additions**
   Consist of device drivers and system applications that optimise the guest operating system for better performance and usability. These are by default a part of the OS when installed via. VirtualBox.
2. **Added Programs and Updates**
   Adobe Reader (XI 11.0.01), Adobe Flash player 11.9, Java 7, Windows Update KB958830 (Re-mote Server Administration Tools), LibreOffice, VLC Media Player, Firefox, Google Chrome,7Zip, paint.net.
3. **Anti VM Detection part 1**
   Changes settings, registry entries and add randomisation in the form of files with common extensions.
4. **Anti VM Detection part 2**
   On every reboot, registry entries and settings that are reset are changed again.

Besides the above the environment has also been made vulnerable by disabling the firewall and disabling Windows User Access Control.

With everything in place, the malware samples are fed to the Cuckoo sandbox. The sandbox will automatically submit, and execute the malware samples one at a time, and generate one report per sample. After execution of a sample, the virtual machine is reverted to a clean state ready to get a new malware sample.

7.3 Feature extraction

After the initial phase I will have one cuckoo report per malware sample. The reports detail the gathered information about the malware samples. But the data in the report needs to be processed to be suitable for cluster analysis.

I will extract all API function calls done by the malicious process during the dynamic analysis. The data can be extracted from the JSON file the Cuckoo sandbox generates. The structure of the JSON file is as follows:

```
{
        ”info”:
        ”procmemory”:
        ”target”:
        ”network”:
        ”signatures”:
        ”static”:
        ”dropped”:
        ”behaviour”:
                ”generic”:
                ”apistats”:
                ”processes”:
                        ”process_path”:
                        ”calls”:
                                ”category”:
                                ”status”:
                                ”stacktrace”:
                                ”api”:
                                …
        …
}
```

Figure 6 Structure of a Cuckoo analysis report in JSON format

The highlighted attribute API contains the name of the API call performed at the given time of execution. The API function calls are extracted in sequential order. However, the value is not as simple as just being a name, which basically corresponds to a windows method that the malware calls. Methods sometimes takes arguments, and these are also recorded. To compare the behaviour of malware samples it would seem logical to include the arguments as they can greatly affect the behaviour and return value of the method called. So, it could be suspected that when looking for malware families, uniformity would also be seen in arguments passed to API function calls.

“*Using the APIs together with the argument values, it is possible to track all the actions performed by the malware. This not only include the used APIs, but also the paths, mutexes and REGkeys accessed or modified during the experiment*” (Hansen & Larsen, 2015).

To try and get some insight into which kind of API function calls are frequently called by the malware I try to draw some statistics from the dataset. First some basic stats calculated based on the extracted Cuckoo reports:

Malware samples: 1279

Unique API function calls: 259 (See Appendix A)

Total API function calls: 15.073.378

Top 20 API function calls:

#	API function	Count	Frequency
1	LdrGetProcedureAddress	8440905	56%
2	RegQueryValueExW	491676	3%
3	FindResourceExW	486114	3%

4	GetSystemMetrics	443047	3%
5	NtClose	367465	2%
6	NtAllocateVirtualMemory	365843	2%
7	RegOpenKeyExW	266629	2%
8	NtFreeVirtualMemory	242361	2%
9	NtQueryDirectoryFile	242323	2%
10	LdrGetDllHandle	238316	2%
11	LdrLoadDll	221685	1%
12	GetTempPathW	216632	1%
13	NtDelayExecution	213363	1%
14	__exception__	186883	1%
15	NtProtectVirtualMemory	168955	1%
16	NtCreateFile	166958	1%
17	NtWriteFile	162642	1%
18	GetFileAttributesW	151603	1%
19	OutputDebugStringA	150109	1%
20	NtReadFile	149680	1%

Table 1 Top 20 API functions called with frequency of all API function calls

That stats show me that the most called API function is far more frequently called than all the other malware samples. If you query the Microsoft documentation for the Windows API functions it states that the functionality of the LdrGetProcedureAddress API function is to retrieve "*… the address of an exported function or variable from the specified dynamic-link library (DLL)*" (Microsoft, 2018). So basically, it is a gateway to other functionality, and holds no real semantic value. But this does not mean that it cannot be an indicator of malicious programs at play. Because the frequency of this API call could be far less if we looked at benign software. But do to its overwhelming representation I fear that it clouds analysis, and it is added as an API function that will be filtered out during the analysis.

7.4 Clustering

The cluster analysis is done in a Python environment using scikit-learn (Pedregosa, et al., 2011) and all charts are done using MatPlotLib (Hunter, 2007). Scikit-learn provides high level functionality that provides easy clustering of the dataset. The full source code in appendices C-F.

To best evaluate my analysis results three different feature representations will be tested, using both OPTICS and Hierarchical analysis. For each analysis method I will find the best values for the minimum samples' parameter for OPTICS, and the number of clusters parameter for Hierarchical.

Performing the clustering analysis requires many considerations for system design to deal with the large dataset, and the poor time complexity of some of the components. The Cuckoo sandbox is a heavy configurable out-of-the-box solution that delivers Cuckoo reports in json format. Everything done to the reports from there on is done in a Python script I have written. The script takes the Cuckoo reports as input, and produces the clusters found as output. To optimise the process several steps are done in parallel:

Processing of Cuckoo reports is done in parallel; results are stored to a file so the reports only needs to be processed once.

Distance matrices are created by calculating the distance between samples in parallel. Distance matrices are also stored in files, so that they only are calculated if new malware samples should be considered. The sorting of distance matrices by clusters is done using Quicksort, a fast and in-place sorting algorithm.

Processing ~1280 malware samples take around 6 hours on my machine for the first run where reports are processed and distance matrices are created. But after my initial run, graphs can be crated in minutes.

7.5 Summary

The analysis design has been created using both familiarity of technologies, best fit for the problem, and performance in mind.

Cuckoo Sandbox has been chosen to do the dynamic analysis because it can deliver the features chosen to analyse, and because of my own familiarity with it. Cuckoo is also known well throughout the malware analysis community providing many resources for help.

Processing the data generated by Cuckoo, as well as the actual clustering analysis, and finally providing some results for the evaluation is all done in Python. Python has many libraries for machine learning which is needed for the cluster analysis. Furthermore, python has a mature build in library for multiprocessing which is needed for increasing the performance of the analysis. This is all important to provide an analysis method that can help reduce the workload of malware analysis.

8 Analysis results

The results presented in the following were found from following the analysis design described in section 7. During the data collection 1287 malware samples were successfully processed in the Cuckoo sandbox generating an equal number of Cuckoo reports. In total a far greater amount was processed but the analysis turned out faulty.

Subsequently the reports were processed by extracting the API call sequences. Following that the call sequences were analysed in three different tracks for comparison. One track where the call sequences was trimmed to 200 API function calls (Track 1), one where the call sequences were also filtered (Track 2) (See Appendix B), and one where they also had repeating sequences collapsed (Track 3).

After that, the distances between all malware samples are calculated using the Levenshtein Ratio formular. The distribution of distances is shown in the figure below:

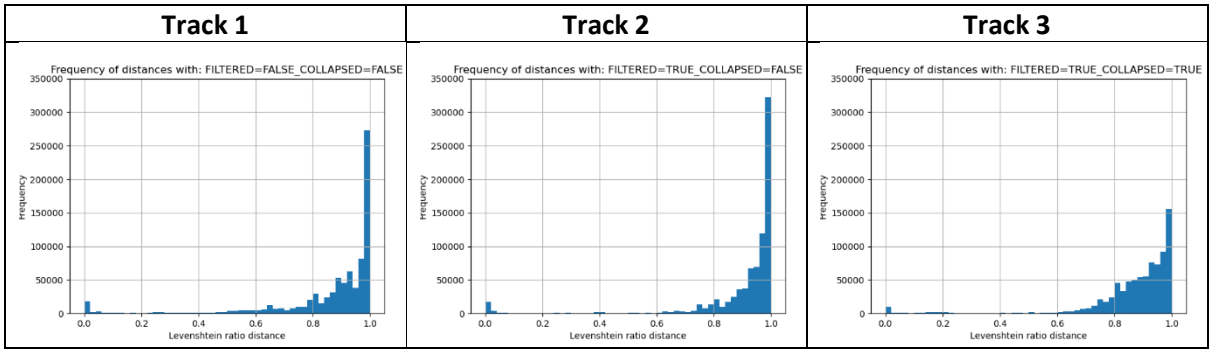


Figure 7 Frequency distribution for the three dataset tracks

The distribution of distance frequencies shows that having both filtered and collapsed API call sequences results in a more diverse distribution. Effectively halfing the number of distances of 1.0. This means that more malware samples have a shorter distance with filtering and collapse than without. This supports the idea that more of the comparable malware behaviour has been brought forward.

After all distances between the samples the results are put into a distance matrix, which is just a 2D arrays holding all distances between samples. The array is mirrored so that given a sample i and a sample j looking up into the array dm, dm[i][j] is the same as dm[j][i]. The clustering analysis can be done on this array and gives the following results for each of the three feature representations:

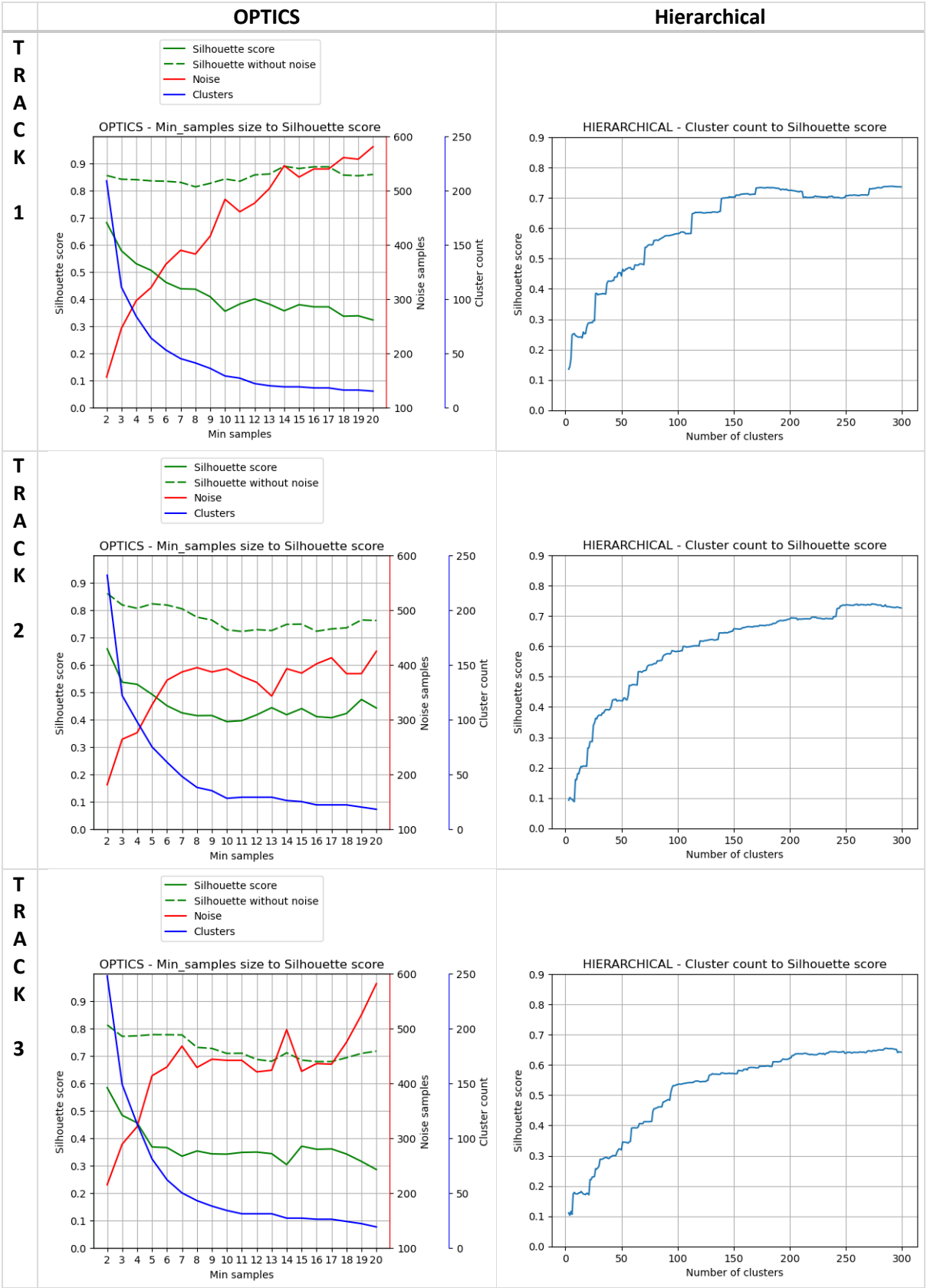


Figure 8 Comparison of Silhouette scores for the OPTICS and Hierarchical methods

The figure above shows different results for the two analysis methods, for each of the three tracks. The main influencial parametere for the OPTICS method is min_samples shown on the x-axis. For the

hierarchical the most influencail pramtre is the number of cluster to find. The charts in figure 8 shows the Silhouette scores found when trying diferent values for the parametres.

In general the silhouette score is the same for all three tracks with the hierarchical method. However it is a bit lower with track three. The steepnes of the curve is also higher before flattening out on track one and two. Besides that difference the results are very similar for the hierarchical method. The optimal point seems to be ~100 clusters, since the returns for adding clusters diminished there after. But the precise number is hard to estimate.

Using OPTICS the best Silhouette score is achieved using the smallest min_sample value of two. However that value gives a huge number of clusters, indicating that the result is over-fitted. Looking at the number of clusters it looks like there is an optimal value around a min_samples equals four or five. There the number of noisy samples are not to high, and the silhouette score is reasonable high with about 0.8 if noise is not included. The number of samples considered noise starts to stagnates at this point, as does the decline in the overall Silhouette score. Espicially when looking at track 2 and 3. Track 1 behaves differently with a steady increase in noise.

The curve for the number of clusters also has a sharp point wich could be considered as optimal using the elbow method. Considering the silhouette score of the clusters only, not considering the noise samples, a consistently high value over 0.8 is seen in all runs. This indicates that the fitness of the clusters are not improved by changing the min_samples value. In fact the reason the shilouette scores declines when increasing min_samples must be because more samples are considered to be noise. This indicates that minimum min_samples of two is the ideal in regards to the silhouette score. The following shows the distance_matrix visualised when trying different values for the n_samples parametre.

Using a value of 4 for min_samples for the OPTICS algorithm, and 100 clusters for the Hierarchical algorithm the following two distance matricies can be created:

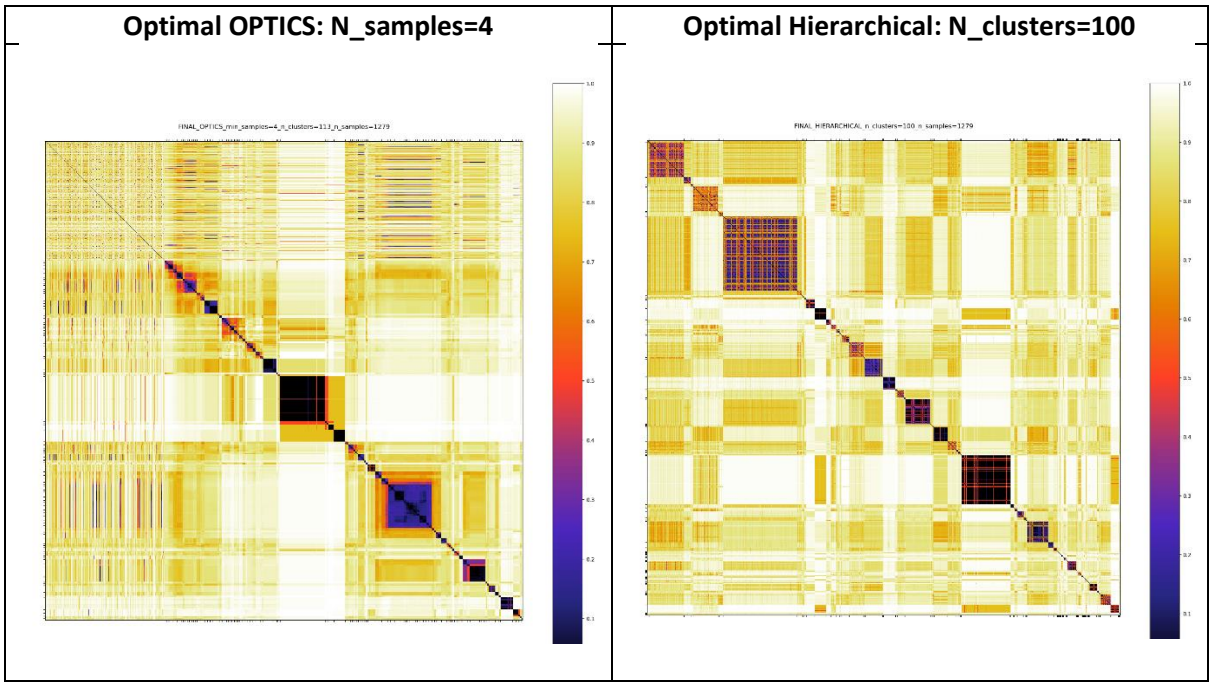


Figure 9 Comparison of distance matrices for OPTICS and hierarchical at estimated optimal values

The analysis results using the different analysis algorithms especially shows in the difference int the density of the clusters found. Since OPTICS also finding noise, the clusters it does find are denser. Whereas as the hierarchical method does not find noise.

For the displayed example in figure 9, the mean silhouette coefficient is ~0.5 with noise, and ~0.8 without. This tells me that the clustering algorithm has performed good, providing highly cohesive clusters. A visual evaluation also looks like some good clusters have formed. The upper left corner contains all the samples considered to be noise by the OPTICS algorithm. That area looks a bit spotted which indicates that several of the samples have a low distance to each other but did not form a dense enough neighbourhood to be considered a cluster.

The hierarchical method creates no noise. However, the clusters found looks less dense. This is confirmed by the lower Silhouette score of ~0.7. The visual evaluation of the clusters shows a different pattern in the ordering of clusters. The inter-cluster relationships look more scattered compared to the OPTICS method. This indicates that the OPTICS method is better at sorting neighbouring cluster in adjacency than the hierarchical.

Comparing the results for the different tracks and clustering algorithms I find the OPTICS on the third track to perform the best. In general, the third track does not score higher Silhouette scores. But with thoughts on the domain the third track is the more sensible one. Comparing the clustering algorithms, I favour OPTICS due to the higher Silhouette scores for clusters, and the better placement of adjunct clusters. I do find the large amount of noise to be an issue with the algorithm. However, I suspect that the amount of noise will be reduced in percentage if a larger body of malware is analysed.

I think the most interesting observations is the areas that intersects horizontally with one cluster and vertically with another. If these areas have a dark nuance, it means that the two clusters are like some degree. The darker the area the more similar the clusters. Those areas are interesting to analyse further to figure out what the similarity means. It could be that they do some similar evasion techniques or other malicious actions. To find this knowledge further analysis would be required.

9 Discussion

Repeating the analysis done in this report quickly becomes difficult with large datasets of malware. This is mostly due to the time complexity of the Levenshtein distance metric when calculating the distance matrix. This involves two operations that work in quadratic time: $O(n^2)$. Calculating the full distance matrix and calculating the Levenshtein distance between samples. In part this is handled on the Levenshtein calculations by limiting the API sequence to 200, reducing the runtime but not the time complexity. For the distance matrix the computational time will grow quadratically with increasing malware samples. The impact of this can be reduced by performing the calculations in parallel as done in this project. However effective that may be, it does not change the time complexity of the issue. If the results are to be used as a classifier, then a more effective model should be built on top of the clusters found.

Since the aim of the project is to find malware that behave similar, only malware samples were processed in the setup. This can be criticized for some reasons. Firstly, the result might show some clusters of malware that exhibit behaviour common for all software. The absence of benign software makes it impossible to decide if some behaviour is malicious or not. The assumption becomes that all behaviour shown is malicious. This is a flaw in the analysis design. But it could also be argued that since the aim is to find clusters within malware samples it is not necessary to include benign software. Even if some malware is clustered by some non-malicious behaviour, they are still clustered because they exhibit the same behaviour. But it does exclude the potential for building a malware detecting model in the future.

The analysis could also benefit from several enhancements already pointed out in previous research. Running the samples once in an identical Cuckoo sandbox only yields one execution path. Exploring multiple execution paths would give deeper insight into the malware behaviour. It would also make the analysis more complex since the different execution paths should be contributed to the same malware sample. The analysis time would also be longer, which could be an issue due to the time complexity of the distance matric calculation.

Cuckoo also offers support for distributing its installation which could drastically increase the number of samples that could processed at a time. Basically, Cuckoo provides a REST API that allows submission of malware for analysis to a remote node. This approach would make the analysis design a lot more scalable. It would mean that layer 2 of the current setup could run as a clone on multiple machines. It is necessary to setup a master machine to orchestrate the analysis. A distributed setup would look like the following:

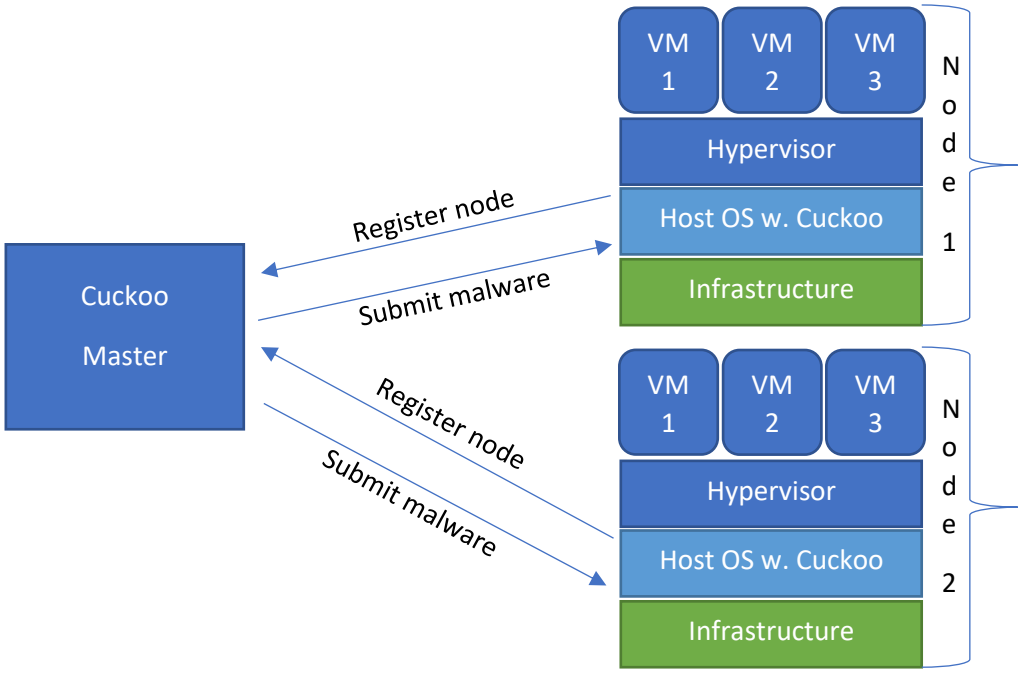


Figure 10 Distributed Cuckoo setup

The distributed setup would greatly increase the throughput as the malware analysis could be partitioned and processed in parallel on different machines. However, it would require access to more hardware, than is within scope of this project. But the second layer of my setup could qualify as a node in the distributed setup, and thus it is possible to scale up in the future.

Another drawback in the analysis design is the fact that the test environment is Windows only. This excludes malware targeting other platforms. To extend the analysis design to include additional platforms, multiple guest VM's operating different platforms must be created. This increases the hardware requirements as the test machine must host multiple VM's. Furthermore, the subsequent processing of the Cuckoo reports must also be changed. Testing malware for Android will obviously not create any Windows API function calls. So, the analysis must be able to examine the Android equivalent.

Looking at Windows API function calls a lot of information about the malware behaviour is omitted in the current setup where only API call names are used. More information could be included if arguments passed to the API call was included in the analysis. But also including other features besides API function calls would give a more detailed view of the malware's behaviour. It would make perfect sense to include events from the network traffic into the analysis. Creating sequences of data spanning multiple features can be done with Cuckoo reports since all the data generated has a timestamp. This would be an interesting topic for future research.

10 Conclusion

This project demonstrates that clusters in a large set of malware can be detected using the Levenshtein distance as a distance metric. This distance metric takes the sequential properties of the entire data object into account. This shows the potential in analysing malwares execution data as sequential data rather than statistical data as is seen in much former research.

Distinct clusters of malware was detected using both OPTICS and hierarchical clustering. Both clustering methods showed good results. The hierarchical method produced the best silhouette scores, but OPTICS came out on top during the visual analysis. The difference in results is caused by the hierarchical method looking at linkage, and the OPTICS looking at density. The OPTICS method also disregards noise which leads to smaller and more focused clusters but when calculating the means silhouette score the noise cluster negatively affected the score.

The project also shows that inter-cluster patterns emerge when displaying the clusters in a distance matrix. This is seen as dark areas in the intersection between two, or more, clusters. The patterns indicates that something is shared between the two clusters, but the pattern was not strong enough to be considered as one. The patterns are most clear when using the OPTICS algorithm. The coupling between the clusters is a sign of something worth investigating to the malware analyst.

The analysis of this project can be utilised when working with large malware sets to reduce the workload to the clusters and inter-cluster relationships found.

11 Perspectivation

Being able to compare how similar the behaviour of two objects is important because it allows for the analysis of large groups. In this project I have shown that such analysis can be performed on Malware and clusters of similar malwares can be formed. Furthermore, I have shown that inter-clusters can be identified giving reason to investigate the relationship between two clusters. Figuring out what connects them, and what does not. The continuation of this research could be to extract the sub-sequences that define clusters, and inter-cluster relationships, and add some semantic understanding to the sequences. Adding human understandable text to these sub-sequences would aid greatly in the rapid analysis of malware samples.

It would also be interesting to see if prediction of future API function calls could be created as a defensive mechanism. Essentially using something like Markov-chains to make predictions if an ongoing sequence of API function calls would possibly lead to something defined as malicious. If this could be detected while the malware is doing evasive behaviour, it could potentially stop the malware before any harm was done. I imagine that this would be even more effective if additional features besides API function calls where considered. Especially network traffic could have an impact here since a lot of malware does expect Internet connectivity and uses it for Command & Control communication and as a sign of being in a real environment instead of a sandboxed.

12 References

(n.d.). Retrieved March 31, 2021, from Docker: https://www.docker.com/

Abu-Jamous, B., Fa, R., & Nandi, A. K. (2015). *Integrative Clluster Analysis in Bioinformatics.* John Wiley & Sons, Incorporated.

Adderley, N., & Peterson, G. (2020). INTERACTIVE TEMPORAL DIGITALFORENSIC EVENT ANALYSIS. *Advances in Digital Forensics XVI*(589), 39-55.

Afianian, A., Niksefat, S., Sadeghiyan, B., & Baptiste, D. (2020, januar 21). Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM computing surveys*, pp. 1-28.

Aggarwal, C. C. (2015). Mining Discrete Sequences. In C. C. Aggarwal, *Data Mining* (pp. 493-529). Springer International Publishing.

Ankerst, M., Breunig, M. M., Kriegel, H.-P., & Sander, J. (1999). OPTICS: Ordering Points To Identify the Clustering Structure. *ACM Sigmod Record, 28*(2), 49-60.

AV-Test. (2021, February 5). *Malware Statistics*. Retrieved from av-test.org: https://www.av-test.org/en/statistics/malware/

Calleja, A., Tapiador, J., & Caballero, J. (2016, September 7). A Look into 30 Years of Malware Development from a Software Metrics Perspective. *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 325-345.

Damodaran, A., Di Troia, F., Visaggio, C. A., Austin, T. H., & Stamp, M. (2017, December 29). A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, pp. 1-12.

Dogan, O., & Oztaysi, B. (2019, June 11). Genders prediction from indoor customer paths by Levenshtein-based fuzzy kNN. *Expert Systems with Applications, 136*, pp. 42-49.

Guarnieri, C. (n.d.). *About Cuckoo*. Retrieved March 31, 2021, from Cuckoo Automated Malware Analysis: https://cuckoosandbox.org/about

Hansen, S. S., & Larsen, T. M. (2015). *Dynamic Malware Analysis: Detectionand Family Classification using MachineLearning.* Aalborg: Aalborg University, Institute of Electronic Systems.

Herzog, C.́ ., Tong, V.́ ., Wilke, P., Van Straaten, A., & Lanet, J.-L. (2009). Evasive Windows Malware: Impact on Antiviruses and Possible Countermeasures.

Higuera, J. B., Aramburu, C. A., Higuera, J.-R. B., Urban, M. A., & Montalvo, J. A. (2020). Systematic Approach to Malware Analysis (SAMA). *Applied Science*, 1-31.

Hungenberg, T., & Eckert, M. (2021, april 30). *Welcome to the INetSim project homepage!* . Retrieved from INetSim: Internet Services Simulation Suite: https://www.inetsim.org/

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science \& Engineering, 9*(3), pp. 90-95. doi:10.1109/MCSE.2007.55

Kävrestad, J. (2020). *Fundamentals of Digital Forensics* (2. ed.). Skövde, Sweden: Springer.

Kirat, D., & Vigna, G. (2015, October 12). MalGene: Automatic Extraction of Malware Analysis Evasion Signature. *Proceedings of the 22nd ACM SIGSAC Conference on computer and communications security*, pp. 769-780.

Levenshtein, V. I. (1965, August). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics - Doklady Akademii Nauk SSSR, 163*(4), pp. 845 - 848.

Microsoft. (2018, May 12). *GetProcAddress function (libloaderapi.h)*. Retrieved from Windows Developer Documentation: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress

Mills, A., & Legg, P. (2020, October 12). Investigating Anti-Evasion Malware Triggers Using Automated Sandbox Reconfiguration Techniques. *Journal of Cybersecurity and Privacy*, pp. 19-39.

Muhovic, T. (2020). *Behavioural Analysis of Malware Using Custom Sandbox Environments.* Aalborg: Aalborg University.

Okolica, J., & Peterson, G. (2010). A COMPILED MEMORY ANALYSIS TOOL. In K.-P. Chow, & S. Shenoi, *Advances in Digital Forensics VI* (pp. 195-206). Hong Kong: Springer.

Ollmann, G. (2008, September). The evolution of commercial malware development kits and colour-by-numbers custom malware. *Computer Fraud & Security*, pp. 4-7.

Oprisa, C., Cabâu, G., & Pal, G. S. (2018). Multi-centroid Cluster Analysis in Malware Research. *Advances in Intelligent Systems and Computing, 674*, 94-103.

Or-Meir, O., Nissim, N., Elovici, Y., & Rokach, L. (2019, September). Dynamic Malware Analysis in the Modern Era - A State of the Art Survey. *ACM Computing Surveys. 52, 5, Article 88*, pp. 1-48.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., . . . Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*(12), pp. 2825-2830.

Pirscoveanu, R. S., & Hansen, S. S. (2015). *Analysis of Malware Behavior: Type Classification using Machine Learning.* Aalborg: Aalborg Univeristy.

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017, July). DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems*, p. 21.

Sihwail, R., Omar, K., & Ariffin, K. A. (2018, september 30). A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science Engineering Information Technology*, pp. 1662-1671.

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press .

Spiegel, S. (2015). *Time series distance measures.* Berlin: Elektrotechnik und Informatik der Technischen Universitat.

Széles, G. J., & Coleşa, A. (2019, January 30). Malware Clustering Based on Called API During Runtime. *Information and Operational Technology Security Systems*, pp. 110-121.

VirusTotal. (2021, April 4). *About us*. Retrieved from VirusTotal: https://support.virustotal.com/hc/en-us/categories/360000160117-About-us

Wireshark Foundation. (2021, april 30). *About wireshark*. Retrieved from Wireshark: https://www.wireshark.org/

Zhao, J., Papapetrou, P., Asker, L., & Boström, H. (2017, January). Learning from heterogeneous temporal data in electronic health records. *Journal of Biomedical Informatics*, pp. 105-119.

13 Appendix A – All recorded API function calls

__anomaly__	GetDiskFreeSpaceW	LookupPrivilegeValueW	ReadProcessMemory
__exception__	GetFileAttributesExW	MessageBoxTimeoutA	RegCloseKey
accept	GetFileAttributesW	MessageBoxTimeoutW	RegCreateKeyExA
bind	GetFileInformationByHandle	Module32FirstW	RegCreateKeyExW
CertControlStore	GetFileInformationByHandleEx	Module32NextW	RegDeleteKeyA
CertOpenStore	GetFileSize	MoveFileWithProgressW	RegDeleteKeyW
CertOpenSystemStoreA	GetFileSizeEx	NetUserGetInfo	RegDeleteValueA
CertOpenSystemStoreW	GetFileType	NtAllocateVirtualMemory	RegDeleteValueW
CHyperlink_SetUrlComponent	GetFileVersionInfoSizeW	NtClose	RegEnumKeyExA
closesocket	GetFileVersionInfoW	NtCreateFile	RegEnumKeyExW
CoCreateInstance	GetForegroundWindow	NtCreateKey	RegEnumKeyW
CoCreateInstanceEx	gethostbyname	NtCreateMutant	RegEnumValueA
CoGetClassObject	GetKeyboardState	NtCreateSection	RegEnumValueW
CoInitializeEx	GetKeyState	NtCreateThreadEx	RegisterHotKey
CoInitializeSecurity	GetNativeSystemInfo	NtDelayExecution	RegOpenKeyExA
COleScript_Compile	GetShortPathNameW	NtDeleteFile	RegOpenKeyExW
connect	getsockname	NtDeleteKey	RegQueryInfoKeyA
ControlService	GetSystemDirectoryA	NtDeleteValueKey	RegQueryInfoKeyW
CopyFileA	GetSystemDirectoryW	NtDeviceIoControlFile	RegQueryValueExA
CopyFileW	GetSystemInfo	NtDuplicateObject	RegQueryValueExW
CoUninitialize	GetSystemMetrics	NtEnumerateKey	RegSetValueExA
CreateActCtxW	GetSystemTimeAsFileTime	NtEnumerateValueKey	RegSetValueExW
CreateDirectoryW	GetSystemWindowsDirectoryA	NtFreeVirtualMemory	RemoveDirectoryA
CreateJobObjectW	GetSystemWindowsDirectoryW	NtGetContextThread	RtlAddVectoredContinueHandler
CreateProcessInternalW	GetTempPathW	NtMapViewOfSection	RtlAddVectoredExceptionHandler
CreateRemoteThread	GetTimeZoneInformation	NtOpenDirectoryObject	RtlCreateUserThread
CreateRemoteThreadEx	GetUserNameA	NtOpenFile	RtlDecompressBuffer
CreateServiceA	GetUserNameExA	NtOpenKey	RtlRemoveVectoredExceptionHandler
CreateThread	GetUserNameExW	NtOpenKeyEx	SearchPathW
CreateToolhelp32Snapshot	GetUserNameW	NtOpenMutant	select
CryptAcquireContextA	GetVolumeNameForVolumeMountPointW	NtOpenProcess	send
CryptAcquireContextW	GetVolumePathNamesForVolumeNameW	NtOpenSection	SendNotifyMessageW
CryptCreateHash	GetVolumePathNameW	NtOpenThread	sendto
CryptDecodeObjectEx	GlobalMemoryStatus	NtProtectVirtualMemory	SetEndOfFile
CryptDecrypt	GlobalMemoryStatusEx	NtQueryAttributesFile	SetErrorMode
CryptEncrypt	HttpOpenRequestA	NtQueryDirectoryFile	SetFileAttributesW

CryptExportKey	HttpOpenRequestW	NtQueryFullAttributesFile	SetFileInformationByHandle
CryptHashData	HttpQueryInfoA	NtQueryInformationFile	SetFilePointer
DeleteFileW	HttpSendRequestA	NtQueryKey	SetFilePointerEx
DeleteService	HttpSendRequestW	NtQueryMultipleValueKey	SetFileTime
DeleteUrlCacheEntryA	InternetCloseHandle	NtQuerySystemInformation	setsockopt
DeleteUrlCacheEntryW	InternetConnectA	NtQueryValueKey	SetUnhandledExceptionFilter
DeviceIoControl	InternetConnectW	NtQueueApcThread	SetWindowsHookExA
DnsQuery_A	InternetCrackUrlA	NtReadFile	SetWindowsHookExW
DrawTextExA	InternetCrackUrlW	NtReadVirtualMemory	ShellExecuteExW
DrawTextExW	InternetGetConnectedState	NtResumeThread	SHGetFolderPathW
EnumWindows	InternetOpenA	NtSetContextThread	SHGetSpecialFolderLocation
FindFirstFileExW	InternetOpenUrlA	NtSetInformationFile	shutdown
FindResourceA	InternetOpenUrlW	NtSetValueKey	SizeofResource
FindResourceExA	InternetOpenW	NtSuspendThread	socket
FindResourceExW	InternetQueryOptionA	NtTerminateProcess	StartServiceA
FindResourceW	InternetSetOptionA	NtTerminateThread	timeGetTime
FindWindowA	InternetSetStatusCallback	NtUnmapViewOfSection	UnhookWindowsHookEx
FindWindowExA	ioctlsocket	NtWriteFile	URLDownloadToFileW
FindWindowW	IsDebuggerPresent	NtWriteVirtualMemory	UuidCreate
GetAdaptersAddresses	IWbemServices_ExecQuery	ObtainUserAgentString	WNetGetProviderNameW
GetAdaptersInfo	LdrGetDllHandle	OleInitialize	WriteProcessMemory
getaddrinfo	LdrGetProcedureAddress	OpenSCManagerA	WSARecv
GetAddrInfoW	LdrLoadDll	OpenSCManagerW	WSARecvFrom
GetAsyncKeyState	LdrUnloadDll	OpenServiceA	WSASend
GetBestInterfaceEx	listen	OpenServiceW	WSASendTo
GetComputerNameA	LoadResource	OutputDebugStringA	WSASocketA
GetComputerNameW	LoadStringA	Process32FirstW	WSASocketW
GetCursorPos	LoadStringW	Process32NextW	WSAStartup
GetDiskFreeSpaceExW	LookupAccountSidW	ReadCabinetState	

14 Appendix B – Filtered list of API function calls

accept	InternetOpenA	NtQueryMultipleValueKey	RemoveDirectoryA
__anomaly__	InternetOpenUrlA	NtQueryValueKey	RemoveDirectoryW
bind	InternetOpenUrlW	NtReadFile	RtlCreateUserThread
closesocket	InternetOpenW	NtReadVirtualMemory	select
connect	InternetReadFile	NtResumeThread	send
ControlService	InternetWriteFile	NtSaveKey	sendto
CopyFileA	ioctlsocket	NtSaveKeyEx	setsockopt
CopyFileExW	LdrGetDllHandle	NtSetContextThread	SetWindowsHookExA
CopyFileW	LdrLoadDll	NtSetInformationFile	SetWindowsHookExW
CreateDirectoryExW	listen	NtSetValueKey	ShellExecuteExW
CreateDirectoryW	LookupPrivilegeValueW	NtSuspendThread	shutdown
CreateProcessInternalW	MoveFileWithProgressW	NtTerminateProcess	socket
CreateRemoteThread	NtCreateFile	NtTerminateThread	StartService
CreateServiceA	NtCreateKey	NtWriteFile	StartServiceA
CreateServiceW	NtCreateMutant	NtWriteVirtualMemory	system
CreateThread	NtCreateNamedPipeFile	OpenSCManagerA	TransmitFile
DeleteFileA	NtCreateProcess	OpenSCManagerW	UnhookWindowsHookEx
DeleteFileW	NtCreateSection	OpenServiceA	URLDownloadToFileW
DeleteService	NtCreateThreadEx	OpenServiceW	VirtualProtectEx
DeviceIoControl	NtCreateUserProcess	ReadProcessMemory	WriteConsoleA
DnsQueryA	NtDelayExecution	recv	WriteConsoleW
ExitProcess	NtDeleteFile	recvfrom	WriteProcessMemory
ExitThread	NtDeleteKey	RegCloseKey	WSARecv
ExitWindowsEx	NtDeleteValueKey	RegCreateKeyExA	WSARecvFrom
FindFirstFileExA	NtDeviceIoControlFile	RegCreateKeyExW	WSASend
FindFirstFileExW	NtEnumerateKey	RegDeleteKeyA	WSASendTo
FindWindowA	NtEnumerateValueKey	RegDeleteKeyW	WSASocketA
FindWindowExA	NtFreeVirtualMemory	RegDeleteValueA	WSASocketW
FindWindowExW	NtGetContextThread	RegDeleteValueW	WSAStartup
FindWindowW	NtLoadKey	RegEnumKeyExA	ZwMapViewOfSection
getaddrinfo	NtMakeTemporaryObject	RegEnumKeyExW	
GetAddrInfoW	NtOpenDirectoryObject	RegEnumKeyW	
GetCursorPos	NtOpenFile	RegEnumValueA	
gethostbyname	NtOpenKey	RegEnumValueW	
GetSystemMetrics	NtOpenKeyEx	RegOpenKeyExA	
HttpOpenRequestA	NtOpenMutant	RegOpenKeyExW	
HttpOpenRequestW	NtOpenSection	RegQueryInfoKeyA	
HttpSendRequestA	NtOpenThread	RegQueryInfoKeyW	
HttpSendRequestW	NtProtectVirtualMemory	RegQueryValueExA	
InternetCloseHandle	NtQueryDirectoryFile	RegQueryValueExW	
InternetConnectA	NtQueryInformationFile	RegSetValueExA	
InternetConnectW	NtQueryKey	RegSetValueExW	

15 Appendix C – Source code – Main.py

```
# This is my main script
import json
import multiprocessing as mp
import os
import time

import matplotlib.cm
import matplotlib.pyplot
import matplotlib.pyplot as plt
import numpy as np
from sklearn import metrics
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import OPTICS

import FeatureProcessing as fp
import my_sorter as my_sort
import process_cuckoo_reports as pcr


def dist_metric(x, y):
    print("X = " + str(int(x[0])) + " Y = " + str(int(y[0])))
    data_x = data[int(x[0])]
    data_y = data[int(y[0])]
    max_len = max(len(data_x), len(data_y))
    # I divide with MAX to get the Levenshtein ratio
    return fp.levenshtein_distance_dp(data_x, data_y) / max_len


def dist_metric_alt(x, y, dm):
    print("X = " + str(x) + " Y = " + str(y))
    data_x = dm[x]
    data_y = dm[y]
    max_len = max(len(data_x), len(data_y))
    # I divide with MAX to get the Levenshtein ratio
    return fp.levenshtein_distance_dp(data_x, data_y) / max_len


def alt_dist_metric(i):
    if i[0] == i[1]:
        return 0.0
    data_x = i[2]
    data_y = i[3]
    max_len = max(len(data_x), len(data_y))
    # I divide with MAX to get the Levenshtein ratio
    dist = fp.levenshtein_distance_dp(data_x, data_y) / max_len
    return [i[0], i[1], dist]


def mp_calc_dist_matrix(idxs, dm):
    calcs = []
    # Define all pairwise:
    for i in range(0, len(idxs)):
        for j in range(i + 1, len(idxs)):
            if i < len(idxs) and j < len(idxs):
```

```
                calcs.append([i, j, dm[i], dm[j]])

    # Submit to pools calculations to pools:
    pool = mp.Pool()
    res = pool.map(alt_dist_metric, calcs)
    pool.close()
    m_out = np.zeros((int(len(idxs)), int(len(idxs))))
    for r in res:
        m_out[r[0]][r[1]] = r[2]
        m_out[r[1]][r[0]] = r[2]
    return m_out


def swap(dist_mat, i, j):
    for y in range(0, len(dist_mat)):
        t = dist_mat[y][i]
        dist_mat[y][i] = dist_mat[y][j]
        dist_mat[y][j] = t

    tmp = dist_mat[i].copy()
    tmp2 = dist_mat[j].copy()
    dist_mat[i] = tmp2
    dist_mat[j] = tmp


def order_dist_matrix(dist_matrix, labels):
    for iter_num in range(len(labels) - 1, 0, -1):
        for idx in range(iter_num):
            if labels[idx] > labels[idx + 1]:
                temp = labels[idx]
                labels[idx] = labels[idx + 1]
                labels[idx + 1] = temp
                swap(dist_matrix, idx, idx + 1)


def store_ordered_dist_matrix_as_png(dist_matrix, labels, title):
    ticks = []
    for i in range(1, len(labels)):
        if labels[i] != labels[i - 1]:
            ticks.append(i - 0.5)
    plt.clf()
    fig, ax = plt.subplots(figsize=(20, 20), sharey=True)
    # fig, ax = plt.subplots(sharey=True)
    cmap = matplotlib.cm.get_cmap('CMRmap')
    cax = ax.matshow(dist_matrix, interpolation='nearest', cmap=cmap)
    # cax = ax.matshow(dist_matrix, interpolation='nearest')
    ax.grid(False)
    plt.suptitle('Clustered Distance Matrix')
    plt.title(title)
    plt.xticks(ticks, color="w")
    plt.yticks(ticks, color="w")
    fig.colorbar(cax, ticks=[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1.0])
    plt.savefig("images/Dist_matrix_" + str(title) + ".png",
bbox_inches='tight')
    plt.close()
```

```
def do_hierarchical_cluster_analysis_routine(api_call_description,
dist_matrix):
    print('DOING HIERARCHICAL AGGLOMERATIVE CLUSTERING:')
    n = len(dist_matrix)
    best_mean_silhouette = -1.0
    best_nc = -1
    best_labels = []
    ncs = []
    mss = []
    top = min(301, n)
    for n_c in range(3, top - 1):
        print('Trying with ' + str(n_c) + ' clusters:')
        agg = AgglomerativeClustering(n_clusters=n_c,
affinity='precomputed', linkage='average')
        labels = agg.fit_predict(dist_matrix)
        mean_silhouette = metrics.silhouette_score(dist_matrix,
labels=labels, metric="precomputed")
        print("For " + str(n_c) + " clusters the mean silhouette score
is: " + str(mean_silhouette))
        ncs.append(n_c)
        mss.append(mean_silhouette)
        best_nc = n_c if mean_silhouette > best_mean_silhouette else
best_nc
        best_labels = labels if mean_silhouette > best_mean_silhouette
else best_labels
        best_mean_silhouette = mean_silhouette if mean_silhouette >
best_mean_silhouette else best_mean_silhouette

    # Display info about cluster to silhouette:
    plt.clf()
    plt.plot(ncs, mss)
    plt.title('HIERARCHICAL - Cluster count to Silhouette score')
    plt.xlabel('Number of clusters')
    plt.ylabel('Silhouette score')
    plt.yticks(np.arange(0, 1, 0.1))
    # plt.ylim([0.0, 1.0])
    plt.grid(True)
    plt.savefig("images/EVAL_HIERARCHICAL_" + api_call_description +
".png", bbox_inches='tight')
    plt.close()
    # Show info about best found nCluster and store dm to image:
    print('Best # of clusters: ' + str(best_nc))
    print("The mean Silhouette score is: " + str(best_mean_silhouette))
    # sorted_dm = dist_matrix.copy()
    # order_dist_matrix(sorted_dm, best_labels)
    sorted_dm, sorted_labels = my_sort.optimal_sort(dist_matrix,
best_labels)
    store_ordered_dist_matrix_as_png(sorted_dm, sorted_labels,
"HIERARCHICAL_analysis_n=" + str(n) + "_nCluster=" + str(
        best_nc) + "_API_format" + api_call_description)
    print("Sorted dist matrix saved as image.")


def plot_optics_reachability(clust, X, title):
    space = np.arange(len(X))
    reachability = clust.reachability_[clust.ordering_]
```

```
    labels = clust.labels_[clust.ordering_]
    plt.clf()
    plt.figure(figsize=(20, 10))

    plt.plot(reachability)
    plt.title('Reachability plot')
    plt.xlabel('Samples')
    plt.ylabel('Reachability (epsilon distance)')
    plt.title('Reachability Plot')
    plt.savefig("images/Reachability_plot_" + str(title) + ".png",
bbox_inches='tight')
    plt.close()


def do_optics_cluster_analysis_routine(api_call_description,
dist_matrix):
    print('DOING OPTICS ANALYSIS:')
    n = len(dist_matrix)
    best_ms = -1
    best_mean_silhouette = -1
    best_labels = []
    list_min_samples = []
    list_mean_silhouettes = []
    list_mean_silhouettes_no_noise = []
    list_clusters = []
    list_noise_count = []
    for ms in range(2, 21):
        try:
            cluster_analyzer = OPTICS(metric="precomputed",
min_samples=ms)
            labels = cluster_analyzer.fit_predict(dist_matrix)
            plot_optics_reachability(cluster_analyzer, dist_matrix,
api_call_description + '_min_samples=' + str(ms))
            lbl_count = len(set(labels)) - (1 if -1 in labels else 0)
            mean_s_coefficient = metrics.silhouette_score(dist_matrix,
labels=labels, metric="precomputed")
            print('For min_samples=' + str(ms) + ' found ' +
str(lbl_count) + ' clusters, and mean_silhouette=' + str(
                mean_s_coefficient))

            list_noise_count.append(np.count_nonzero(labels == -1))
            list_min_samples.append(ms)
            list_mean_silhouettes.append(mean_s_coefficient)
            best_ms = ms if mean_s_coefficient > best_mean_silhouette
else best_ms
            best_labels = labels.copy() if mean_s_coefficient >
best_mean_silhouette else best_labels
            list_clusters.append(len(set(labels)) - (1 if -1 in
best_labels else 0))
            best_mean_silhouette = mean_s_coefficient if
mean_s_coefficient > best_mean_silhouette else best_mean_silhouette

            no_noise_dm, no_noise_labels =
get_noise_free_dm_n_labels_copy(dist_matrix, labels)
            no_noise_mean_s_coefficient =
metrics.silhouette_score(no_noise_dm, labels=no_noise_labels,
```

```
metric="precomputed")
            print('For min_samples=' + str(ms) + ' found no_noise
mean_silhouette=' + str(no_noise_mean_s_coefficient))

list_mean_silhouettes_no_noise.append(no_noise_mean_s_coefficient)
        except Exception as e:
            print("Could not do optics for min_samples=" + str(ms))
            print(e)

    # Display info about cluster to min_samples:
    plt.clf()
    fig, axes = plt.subplots()

    axes.spines['left'].set_color('green')
    axes.set_ylim([0.0, 1.0])
    axes.xaxis.set_ticks(np.arange(0, 21, 1))
    axes.yaxis.set_ticks(np.arange(0, 1, 0.1))
    axes.grid(True)

    fig.subplots_adjust(right=0.75)
    twin_axes = axes.twinx()
    twin_axes.spines['right'].set_color('red')
    twin_axes.set_ylim([100, 600])

    second_twin = axes.twinx()
    second_twin.spines['right'].set_position(('axes', 1.2))
    second_twin.spines['right'].set_color('blue')
    second_twin.set_ylim([0, 250])

    p1, = axes.plot(list_min_samples, list_mean_silhouettes,
color='green', label='Silhouette score')
    p2, = axes.plot(list_min_samples, list_mean_silhouettes_no_noise,
color='green', dashes=[6, 2], label="Silhouette without noise")
    axes.set_xlabel("Min samples")
    axes.set_ylabel("Silhouette score")

    p3, = twin_axes.plot(list_min_samples, list_noise_count,
color='red', label='Noise')
    twin_axes.set_ylabel("Noise samples")

    p4, = second_twin.plot(list_min_samples, list_clusters,
color='blue', label='Clusters')
    second_twin.set_ylabel('Cluster count')

    axes.legend(handles=[p1, p2, p3, p4], bbox_to_anchor=(0.5, 1.1),
loc='lower center')
    plt.title('OPTICS - Min_samples size to Silhouette score')
    plt.savefig("images/EVAL_OPTICS_" + api_call_description + ".png",
bbox_inches='tight')
    plt.close()
    print('')
    print('***** OPTICS Analysis done *****')
    print('Best min_sample=' + str(best_ms))
    print('Finds ' + str(len(set(best_labels)) - (1 if -1 in
best_labels else 0)) + ' clusters')
    # print('Samples counted as noise: ' + str(best_labels.count(-1)))
```

```
    print('Samples counted as noise: ' +
str(np.count_nonzero(best_labels == -1)))
    print('Mean silhouette: ' + str(best_mean_silhouette))
    # sorted_dm = dist_matrix.copy()
    # my_sort.tim_sort(best_labels, sorted_dm)
    # order_dist_matrix(sorted_dm, best_labels)
    sorted_dm, sorted_lbls = my_sort.optimal_sort(dist_matrix,
best_labels)

    noise_count = np.count_nonzero(best_labels == -1)
    print("Samples considered noise: " + str(noise_count))
    store_ordered_dist_matrix_as_png(sorted_dm, sorted_lbls,
"OPTICS_analysis_n=" + str(n) + "_min_samples=" + str(
        best_ms) + "_API_format=" + api_call_description)
    print("Sorted dist matrix saved as image.")


def get_noise_free_dm_n_labels_copy(dist_matrix, labels):
    no_noise_dm = np.delete(dist_matrix, np.where(labels == -1),
axis=0)
    no_noise_dm = np.delete(no_noise_dm, np.where(labels == -1),
axis=1)
    no_noise_labels = np.delete(labels, np.where(labels == -1))
    return no_noise_dm, no_noise_labels


def do_final_optics(dm, min_sampels):
    print('')
    print('***** DOING FINAL OPTICS *****')
    cluster_analyzer = OPTICS(metric="precomputed",
min_samples=min_sampels)
    labels = cluster_analyzer.fit_predict(dm)
    lbl_count = len(set(labels)) - (1 if -1 in labels else 0)
    mean_s_coefficient = metrics.silhouette_score(dm, labels=labels,
metric="precomputed")
    print('For min_samples=' + str(min_sampels) + ' found ' +
str(lbl_count) + ' clusters, and mean_silhouette=' + str(
        mean_s_coefficient))

    # orderd_dm = dm.copy()
    # order_dist_matrix(orderd_dm, labels)

    sorted_dm, sorted_labels = my_sort.optimal_sort(dm, labels)

    title = 'FINAL_OPTICS_min_samples=' + str(min_sampels)
    title += '_n_clusters=' + str(lbl_count)
    title += '_n_samples=' + str(len(labels))
    store_ordered_dist_matrix_as_png(sorted_dm, sorted_labels, title)
    print('.......... DONE!')


def do_final_hierarchical(dm, n_clusters):
    print('')
    print('***** DOING FINAL Hierarchical *****')
    agg = AgglomerativeClustering(n_clusters=n_clusters,
affinity='precomputed', linkage='average')
    labels = agg.fit_predict(dm)
```

```
    mean_silhouette = metrics.silhouette_score(dm, labels=labels,
metric="precomputed")
    print("For " + str(n_clusters) + " clusters the mean silhouette
score is: " + str(mean_silhouette))

    sorted_dm, sorted_labels = my_sort.optimal_sort(dm, labels)

    title = 'FINAL_HIERARCHICAL'
    title += '_n_clusters=' + str(n_clusters)
    title += '_n_samples=' + str(len(labels))
    store_ordered_dist_matrix_as_png(sorted_dm, sorted_labels, title)
    print('.......... DONE!')


def find_optimal_values():
    global j
    labels = ['FILTERED=FALSE_COLLAPSED=FALSE',
'FILTERED=TRUE_COLLAPSED=FALSE', 'FILTERED=TRUE_COLLAPSED=TRUE']
    for dm_id in range(0, 3):
        dist_list = []
        for i in range(0, len(m[dm_id])):
            for j in range(i, len(m[dm_id])):
                if not i == j:
                    dist_list.append(m[dm_id][i][j])
        print("Calculating frequency of distances in distance matrix:")
        print("Dist counts: " + str(len(dist_list)))
        plt.clf()
        plt.hist(dist_list, bins=50)
        plt.gca().set(title='Frequency of Distances',
ylabel='Frequency', xlabel='Levenshtein ratio distance')
        plt.title('Frequency of distances with: ' + labels[dm_id])
        plt.ylim([0, 350000])
        plt.grid(True)
        plt.savefig("images/Dist_frequencies_" + labels[dm_id] +
".png")
        plt.close()
        print(".......... DONE")
        print('')

        do_optics_cluster_analysis_routine(labels[dm_id] + '_API_SEQ',
m[dm_id])
        do_hierarchical_cluster_analysis_routine(labels[dm_id] +
'_API_SEQ', m[dm_id])


if __name__ == '__main__':
    start = time.time()
    cpus = 12
    mp.freeze_support()
    stored_dist_matrix = "data/dist_matrix.json"
    global glob_data
    data = []
    global toShare
    print("###GO GO GO###")
    m = []

    if os.path.isfile(stored_dist_matrix):
```

```
        print("Loading stored distance matrix")
        f = open(stored_dist_matrix, "r")
        j = f.read()
        f.close()
        data = json.loads(j)
        m = np.array(data)
        print(".......... DONE")
        print('')
    else:
        workdir = "C:\\Users\\stegg\\OneDrive\\Documents\\Master
Projekt\\Data\\mal_reports\\"
        d = pcr.mp_get_all_files_api_sequences(workdir)
        # print("Samples: " + str(len(data)))
        print("Creating a new distance matrices")
        for dms in d:
            X = np.arange(len(dms)).reshape(-1, 1)
            currX = -1
            start = time.time()
            m = mp_calc_dist_matrix(X, dms)
            end = time.time()
            print("Time take: " + str(end - start))
            print(".......... DONE")
            print('')
            data.append(m)

        print("Saving distance matrix to file:")
        m_list = []
        for meh in data:
            m_list.append(meh.tolist())
        m_as_json = json.dumps(m_list)
        f = open("data/dist_matrix.json", "w")
        f.write(m_as_json)
        f.close()
        print(".......... DONE")
        print('')
        m = np.array(m_list)

    # m has the distance matrices:
    find_optimal_values()

    #do_final_hierarchical(m[2], 100)
    #do_final_hierarchical(m[2], 250)

    #do_final_optics(m[2], 2)
    #do_final_optics(m[2], 4)
    #do_final_optics(m[2], 5)
    #do_final_optics(m[2], 7)

    #
malware_stats.do_api_analysis("C:\\Users\\stegg\\OneDrive\\Documents\\M
aster Projekt\\Data\\mal_reports\\")

    end = time.time()
    print('Time taken: ' + str(end - start) + " sec.")
    print('')
    print("********* ALL DONE **********")
```

16 Appendix D – Source code – FeatureProcessing.py

```
import difflib as diff
import os
import json
import numpy as np
import sklearn.metrics.pairwise as pw
from joblib import Parallel, delayed
from sklearn.utils import gen_even_slices
import multiprocessing as mp


def levenshtein_distance_dp(token1, token2):

    if token1 == token2:
        return 0

    if len(token1) == 0:
        return len(token2)

    if len(token2) == 0:
        return len(token1)

    distances = np.zeros((len(token1) + 1, len(token2) + 1))

    for t1 in range(len(token1) + 1):
        distances[t1][0] = t1

    for t2 in range(len(token2) + 1):
        distances[0][t2] = t2

    for t1 in range(1, len(token1) + 1):
        for t2 in range(1, len(token2) + 1):
            if token1[t1 - 1] == token2[t2 - 1]:
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                a = distances[t1][t2 - 1]
                b = distances[t1 - 1][t2]
                c = distances[t1 - 1][t2 - 1]

                if a <= b and a <= c:
                    distances[t1][t2] = a + 1
                elif b <= a and b <= c:
                    distances[t1][t2] = b + 1
                else:
                    distances[t1][t2] = c + 1

    # print_distances(distances, len(token1), len(token2))
    return distances[len(token1)][len(token2)]


def print_distances(distances, token1Length, token2Length):
    for t1 in range(token1Length + 1):
        for t2 in range(token2Length + 1):
            print(int(distances[t1][t2]), end=" ")
```

```
        print()


def hamming_distance(sequence_a, sequence_b, length):
    dist_counter = int(0)
    if (len(sequence_a) < length) or (len(sequence_b) < length):
        dist_counter = length - min(len(sequence_a), len(sequence_b))
    top = length - dist_counter
    i = 0
    while i < top:
        if sequence_a[i] != sequence_b[i]:
            dist_counter += 1
        i += 1
    return int(dist_counter)


# Gets the API call sequence performed by the malware process matching
the filename of the given Cuckoo result
def get_api_call_sequence(file):
    try:
        with open(str(file), 'r') as myFile:
            data = myFile.read()
            obj = json.loads(data)
            if 'behavior' in obj:
                behavior = obj['behavior']
                if 'processes' in behavior:
                    processes = behavior['processes']
                    # Find the right process
                    process = get_process(file, processes)
                    if process != -1:
                        calls = process['calls']
                        api_calls = []
                        for i in calls:
                            api_calls.append(str(i['api']))
                        return api_calls
    except:
        return []
    finally:
        myFile.close()


def lcs(X, Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None] * (n + 1) for i in range(m + 1)]

    """Following steps build L[m + 1][n + 1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif X[i - 1] == Y[j - 1]:
```

```
                L[i][j] = L[i - 1][j - 1] + 1
            else:
                L[i][j] = max(L[i - 1][j], L[i][j - 1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]


# Creates a dist matrix
def my_pairwise_distances(X, Y=None, metric="euclidean", n_jobs=1,
data=[], **kwds):
    # Check matrices first (this is usually done by the metric).
    X, Y = pw.check_pairwise_arrays(X, Y)
    n_x, n_y = X.shape[0], Y.shape[0]
    # Calculate distance for each element in X and Y.
    # FIXME: can use n_jobs here too
    D = np.zeros((n_x, n_y), dtype='float')
    cpus = mp.cpu_count()
    print("CPUS: " + str(cpus))
    pool = mp.Pool(12)

    for i in range(n_x):
        start = 0
        if X is Y:
            start = i
        for j in range(start, n_y):
            results = pool.apply_async(my_tmp, args=(X, Y, i, j, kwds,
metric, data))


    pool.close()
    pool.join()
    for r in results.get():
        D[r[1], r[2]] = r[0]
        D[r[2], r[1]] = r[0]
    return D


def my_tmp(X, Y, i, j, kwds, metric, data):
    dist = metric(X[i], Y[j], data, **kwds)
    return dist, i, j


# Gets the process from a Cuckoo json result that matches the given
file name
def get_process(file, processes):
    filename = get_filename(file)
    for i in processes:
        if str(i['process_path']).find(filename) != -1:
            return i
    return -1


# Gets the name portion from a file path
def get_filename(file):
    return os.path.basename(file).split('.')[0]
```

17 Appendix E – Source code – my_sorter.py

```
import numpy as np


def optimal_sort(dm, labels):
    idx = np.argsort(labels)
    sorted_dm = np.zeros((len(dm), len(dm)))
    for col in range(0, len(idx)):
        for row in range(col, len(idx)):
            old_col = idx[col]
            olr_row = idx[row]
            dist = dm[old_col, olr_row]
            sorted_dm[col][row] = dist
            sorted_dm[row][col] = dist
    sorted_labels = np.sort(labels)
    return sorted_dm, sorted_labels
```

18 Appendix F – Source code – process_cuckoo_reports.py

```
import os
import json
import FeatureProcessing as fp
import malware_stats as ms
import multiprocessing as mp

storedData = "data/apilist.json"
samples_to_process = 0
api_sequences_desired_length = 200

good_apis = ['NtOpenSection', 'NtLoadKey', 'FindFirstFileExW',
'NtDeleteValueKey', 'ExitThread', 'closesocket',
             'WSARecv', 'recv', 'socket', 'getaddrinfo',
'NtProtectVirtualMemory', 'RtlCreateUserThread', 'connect',
             'setsockopt', 'HttpSendRequestA', 'NtResumeThread',
'RegDeleteValueW', 'NtCreateSection',
             'NtMakeTemporaryObject', 'NtGetContextThread',
'StartService', 'RegEnumKeyExA', 'RegSetValueExW',
             'SetWindowsHookExA', 'NtReadVirtualMemory',
'WriteConsoleA', 'FindWindowExW', 'send',
             'WSASendTo', 'NtDelayExecution', 'NtOpenThread',
'InternetOpenA',
             'ShellExecuteExW', 'CreateProcessInternalW',
'VirtualProtectEx', 'FindWindowExA', 'RegSetValueExA',
             'SetWindowsHookExW', 'RegOpenKeyExA', 'NtSaveKeyEx',
'StartServiceA', 'ZwMapViewOfSection',
             'NtEnumerateKey', 'HttpSendRequestW', 'GetCursorPos',
'NtOpenDirectoryObject', 'LookupPrivilegeValueW',
             '__anomaly__', 'LdrLoadDll', 'InternetOpenW',
'RegEnumValueW', 'NtCreateUserProcess', 'ExitWindowsEx',
             'NtCreateFile', 'URLDownloadToFileW', 'NtQueryValueKey',
'TransmitFile', 'WriteConsoleW', 'RegDeleteKeyA',
             'RegEnumKeyExW', 'RegCloseKey', 'HttpOpenRequestA',
'NtCreateProcess', 'NtSetInformationFile', 'accept',
             'recvfrom', 'NtCreateKey', 'FindWindowW', 'sendto',
'MoveFileWithProgressW', 'ControlService',
             'NtSuspendThread', 'ioctlsocket', 'RegDeleteKeyW',
'NtQueryInformationFile', 'WSAStartup', 'FindWindowA',
             'RegCreateKeyExW', 'NtTerminateThread',
'HttpOpenRequestW', 'DeviceIoControl', 'NtTerminateProcess',
             'NtFreeVirtualMemory', 'WSASocketA', 'RegOpenKeyExW',
'InternetConnectW', 'CopyFileW', 'shutdown',
             'UnhookWindowsHookEx', 'OpenServiceA', 'DeleteService',
'RegEnumKeyW', 'WriteProcessMemory', 'select',
             'InternetConnectA', 'WSARecvFrom', 'NtQueryKey',
'NtSaveKey', 'NtSetContextThread', 'CreateRemoteThread',
             'RegDeleteValueA', 'RegQueryValueExA', 'GetSystemMetrics',
'CopyFileExW', 'RemoveDirectoryW',
             'NtCreateThreadEx', 'NtOpenMutant', 'OpenServiceW',
'CreateServiceA', 'NtOpenFile', 'NtSetValueKey',
             'RegQueryInfoKeyA', 'RegQueryInfoKeyW', 'ExitProcess',
'InternetCloseHandle', 'NtQueryDirectoryFile',
```

```
            'WSASocketW', 'system', 'NtDeleteKey', 'RegCreateKeyExA',
'DeleteFileA', 'gethostbyname',
            'RemoveDirectoryA', 'ReadProcessMemory', 'CopyFileA',
'RegQueryValueExW', 'CreateDirectoryW',
            'NtCreateMutant', 'WSASend', 'DeleteFileW',
'GetAddrInfoW', 'NtDeviceIoControlFile', 'NtDeleteFile',
            'InternetOpenUrlA', 'NtReadFile', 'CreateServiceW',
'bind', 'RegEnumValueA', 'listen', 'NtOpenKey',
            'NtWriteFile', 'NtQueryMultipleValueKey',
'OpenSCManagerW', 'LdrGetDllHandle', 'DnsQueryA',
            'InternetOpenUrlW', 'NtWriteVirtualMemory',
'InternetWriteFile', 'NtOpenKeyEx', 'NtEnumerateValueKey',
            'InternetReadFile', 'OpenSCManagerA',
'CreateDirectoryExW', 'FindFirstFileExA', 'CreateThread',
            'NtCreateNamedPipeFile']


def mp_get_all_files_api_sequences(workdir):
    data = []
    if os.path.isfile(storedData):
        print("Loading stored API sequence call list")
        f = open(storedData, "r")
        j = f.read()
        data = json.loads(j)
        f.close()
        print("......... DONE")
        print('')
    else:
        lenghty = samples_to_process if samples_to_process > 0 else
"ALL"
        print("Creating new API sequence call list of length: " +
str(lenghty))
        files = os.listdir(workdir)
        if samples_to_process > 0:
            print('Processing ' + str(samples_to_process) + ' files
from: ' + str(workdir))
            files = files[:samples_to_process]
        else:
            print('Processing ALL files from: ' + str(workdir))
        file_list = [workdir + f for f in files]

        tmp_a = []
        tmp_b = []
        tmp_c = []

        pool = mp.Pool()
        result = pool.map(read_api_sequence_from_file, file_list)
        for r in result:
            if not (r is None) and len(r) > 0:
                if r[0] is not None and r[1] is not None and r[2] is
not None and len(r[0]) > 0 and len(r[1]) > 0 and len(r[2]) > 0:
                    tmp_a.append(r[0])
                    tmp_b.append(r[1])
                    tmp_c.append(r[2])
                else:
                    print('Discarded api seq: ' + str(r))
        pool.close()
```

```
        data.append(tmp_a)
        data.append(tmp_b)
        data.append(tmp_c)

        jsonfile = json.dumps(data)
        f = open(storedData, "w")
        f.write(jsonfile)
        f.close()
        print("......... DONE")
        print('')
    return data


def read_api_sequence_from_file(file):
    a = []
    b = []
    c = []
    if os.path.isfile(file):
        sequence_calls = fp.get_api_call_sequence(file)
        if sequence_calls is not None:
            a = sequence_calls[:api_sequences_desired_length].copy()
            filter_bad_apis(sequence_calls)
            b = sequence_calls[:api_sequences_desired_length].copy()
            collapse_duplicates(sequence_calls)
            c = sequence_calls[:api_sequences_desired_length].copy()
            return [a, b, c]


def collapse_duplicates(sequence_calls):
    sequence_calls.reverse()
    for ng in range(5, 0, -1):
        try:
            i = 0
            while i < len(sequence_calls) - ng * 2:
                a = sequence_calls[i:i + ng]
                b = sequence_calls[i + 1 + ng:i + 1 + ng * 2]
                if a == b:
                    del_list = range(i - 1 + ng, i - 1, -1)
                    i_to_del = i - 1 + ng
                    for j in del_list:
                        del sequence_calls[i_to_del]
                else:
                    i = i + 1
        except:
            print('aui')
    sequence_calls.reverse()


def get_all_files_api_sequences(workdir):
    data = []
    if os.path.isfile(storedData):
        f = open(storedData, "r")
        j = f.read()
        data = json.loads(j)
        f.close()
    else:
```

```
        files = os.listdir(workdir)
        for file in files:
            if len(data) >= samples_to_process:
                break
            if os.path.isfile(workdir + file):
                sequence_calls = fp.get_api_call_sequence(workdir +
file)
                if sequence_calls is not None:
                    filter_bad_apis(sequence_calls)
                    # TODO Collapse repeated sequences
                    if 0 < len(sequence_calls):

data.append(sequence_calls[:api_sequences_desired_length])
        jsonfile = json.dumps(data)
        f = open(storedData, "w")
        f.write(jsonfile)
        f.close()
    return data


def filter_bad_apis(sequence_calls):
    for sq in sequence_calls:
        if sq not in good_apis:
            while sq in sequence_calls:
                sequence_calls.remove(sq)


def pcr_test():
    print("Start collecting data:")
    test = get_all_files_api_sequences(
        "C:\\Users\\stegg\\OneDrive\\Documents\\Master
Projekt\\Data\\mal_reports\\")

    print("Count: " + str(len(test)))
    freq_report = ms.get_freq_report(test)
    print("Unique api calls after filter: " + str(len(freq_report)))
```