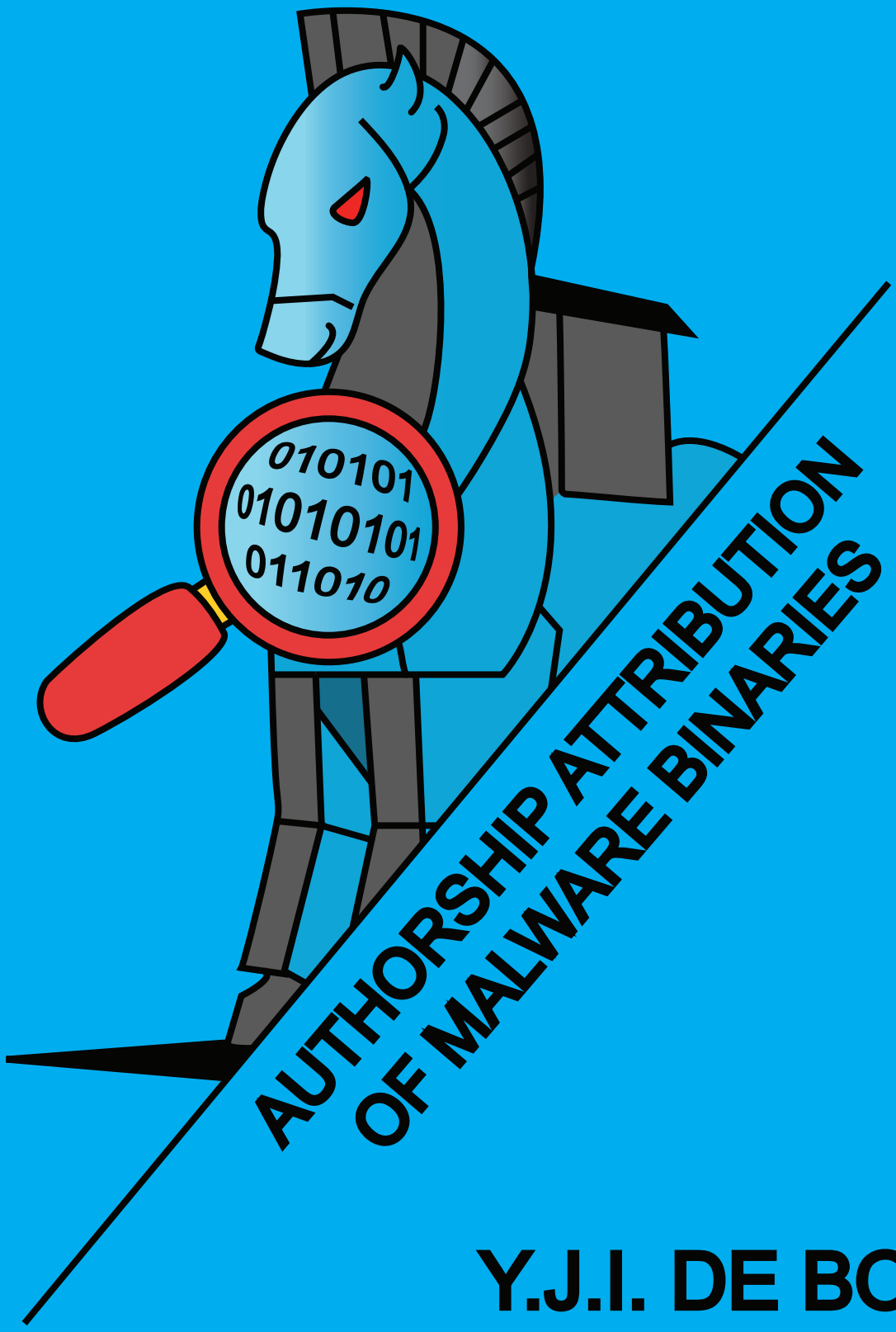


AUTHORSHIP ATTRIBUTION OF MALWARE BINARIES

Y.J.I. DE BOER

TU Delft

Authorship Attribution of Malware Binaries

by

Yorick J.I. de Boer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday July 13, 2020 at 14:00.

Specialization:	Computer Science - Cyber Security	
Student number:	4287304	
Project duration:	August, 2019 – June, 2020	
Thesis committee:	Dr. ir. S. E. Verwer,	TU Delft, supervisor
	Prof Dr. Ir. R. L. Lagendijk,	TU Delft
	Dr. M. Finavaro Aviche,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Attribution of the malware to the developers writing the malware is an important factor in cybercrime investigative work. Clustering together not only malware of the same family, but also inter-family malware relations together provides more information about the authors and aid further malware analysis work. In this report, previous work which concluded attribution on compiled binaries can be done by a programmer their style is questioned. Given insight on this matter, this report explores new clustering techniques for both static and dynamically derived features from malware binaries. Both methods are complementary as they provide very different types of data. In the static analysis, the data for the similarity comparison is derived from disassembled binaries, while in dynamic analysis the choice was made to record system calls executed by the malware during execution. We use a finer granularity than when comparing the data of the complete binaries with each other, such that instead of differences, fine similarities among malware families can be found. Evaluation of clusters is a difficult subject, because of its unsupervised nature and data quality related causes. However, upon manual inspection of the generated clusters, the newly developed clustering methods confirm previously discovered similarities but also find new connections among malware families.

Preface

Ik wil Team High Tech Crime onderdeel van de Nederlandse Politie bedanken voor de initiele onderzoeksvraag en het bieden van verdere ondersteuning tijdens deze Thesis.

Ik wil Sicco Verwer bedanken voor alle gesprekkken, ideeen en feedback.

Mijn Familie wil ik bedanken voor alle onvoorwaardelijke liefde en hulp, want zonder jullie was ik nooit zover gekomen.

Verder wil ik Vincent bedanken voor alle pauze potjes 0AD, en Rizkhi voor haar constante aanmoediging.

Delft, Juni 2020

Contents

1 Introduction 1
1.1 Goal. . . . 1
1.2 Similarity . . . 2
1.3 Research methods . . . 3
1.4 Research questions . . . 4
1.5 Scope . . . 5
1.6 Contributions . . . 5
1.7 Summary of results . . . 6
1.8 Structure of this report . . . 6

2 Related work 7
2.1 Static analysis . . . 7
2.1.1 Coding style . . . 7
2.1.2 Binary similarity . . . 9
2.2 Dynamic analysis . . . 9
2.2.1 Distributions . . . 9
2.2.2 Sequence alignment . . . 10
2.2.3 Graph matching . . . 11
2.2.4 Frequent patterns . . . 11
2.2.5 Current methods of API call sequence extraction systems and techniques . . . 11
2.3 Research gap . . . 12
2.3.1 Static analysis . . . 12
2.3.2 Dynamic analysis . . . 12
2.3.3 General . . . 13
2.4 Summary . . . 13

3 Theory 15
3.1 Malware. . . . 15
3.1.1 Code obfuscation . . . 16
3.1.2 Data transformation . . . 16
3.1.3 Packing . . . 18
3.1.4 Runtime evasion . . . 18
3.1.5 Evasion detection . . . 19
3.2 Metrics . . . 20
3.2.1 n-grams . . . 20
3.2.2 Cosine similarity . . . 21
3.2.3 t-SNE . . . 21
3.2.4 HDBSCAN . . . 21
3.2.5 Clustering evaluation . . . 22
3.3 Summary . . . 23

4 Static analysis 25
4.1 Dataset Exploration . . . 25
4.1.1 Google Code Jam. . . . 25
4.1.2 Malpedia. . . . 26
4.2 Feature extraction. . . . 28
4.2.1 From source code to machine code . . . 28
4.2.2 Instruction n-grams . . . 28

4.3 Classifcation . . . 29
4.3.1 Results: classification - GCJ . . . 29
4.3.2 Results: classifcation - Malpedia . . . 30
4.3.3 Discussion: classification . . . 31
4.4 Clustering . . . 33
4.4.1 Results: clustering - GCJ . . . 35
4.4.2 Results: clustering - Malpedia . . . 36
4.4.3 Discussion: clustering - GCJ . . . 40
4.4.4 Discussion: Clustering - Malpedia . . . 40
4.4.5 Limitations . . . 45
4.5 Summary . . . 45

5 Dynamic analysis 47
5.1 Theory . . . 47
5.2 Dataset generation . . . 48
5.3 Dataset exploration . . . 51
5.4 Feature extraction . . . 54
5.4.1 System call datasets . . . 56
5.4.2 Word2Vec . . . 58
5.4.3 Sequence alignment . . . 60
5.4.4 n-gram based . . . 60
5.4.5 Reduction of the sequence to a Markov chain . . . 60
5.4.6 Frequent sequential patterns and frequent item set based . . . 61
5.5 Method . . . 61
5.5.1 Filtering malware system calls . . . 61
5.5.2 Similarity analysis . . . 62
5.6 Results . . . 64
5.6.1 Full binary n-gram vector analysis . . . 64
5.6.2 Custom clustering based on frequent set finding . . . 68
5.7 Discussion and conclusion . . . 71
5.8 Limitations . . . 71
5.8.1 Dataset . . . 71
5.8.2 Technique . . . 72
5.9 Summary . . . 72

6 Conclusion and future work 73
6.1 Conclusion . . . 73
6.2 Research questions . . . 73
6.3 Limitations . . . 75
6.4 Future work . . . 76

A Google Code Jam Dataset - A possible issue 79

B Table comparison Friedex versions 81

C Table comparison TA505 comparison 83

D Dynamic analysis - cosine similarity - Malpedia dumped dataset 85

E Dynamic analysis - cosine similarity - Malpedia non dumped dataset 87

Bibliography 89

1 Introduction

The number of malware samples is still growing in 2019 in comparison to previous years. According to Anti-Virus company Symantec their latest report of 2019 [70] they detected almost 250 million new malware variants in just 2018. McAfee [47] shows similar numbers with a count of about 150 million new malware found in the first three quarters of 2018. These are the typical first lines of the introduction of any malware related scientific research. However, the size of these numbers has not decreased in the past years, instead the numbers have only grown. The only solution is to actually stop the production of malware, and the way to do that is by catching the perpetrators developing the malware. As a step towards this we want to attribute the malware to their respective authors.

Malware developers have a valid incentive to remain anonymous, since it is generally considered a crime to develop malware and spread malware. Therefore, they take every measure to hide or remove any digital fingerprints from their software. The development of malware can potentially take several years, for this reason parts of a malware program like any software are likely to be reused in newer versions of even in other programs. This can be done in the form of newer versions of the same malware, but can also occur between malware that have similar goals. We want to identify these similarities in this thesis.

1.1. Goal

In this research we want to analyze these malware binaries with the goal of clustering similar malware samples in an automated way for forensic purposes. Two main research methods can be distinguished, the analysis on the binary can be done either statically or dynamically, as illustrated in Figure 1.1. With static analysis the binary is not actually executed on a system, the binary is only decompiled such that the code can be inspected in a human readable format. With dynamic analysis the binary is executed on an actual system, so the behavior of the program can just be observed and recorded. Both of these have their up and downsides in terms of available features that can be extracted by the analyst, but also regarding the defences malware developers put up.

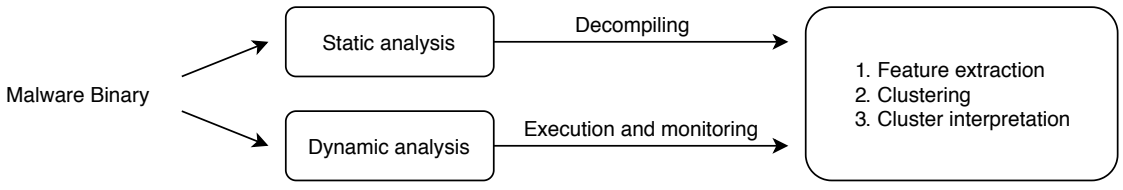


Figure 1.1: Summary of the flow of information and goals of this thesis.

In both cases we want the full results of the analysis to be human interpretable, such that it can help to assist in further analysis of a set of malware binaries. Being human interpretable in this case means we want to be able to visualize the results graphically. All similarity relations should become visible within a single figure for a set of malware samples.

1.2. Similarity

Similarity can be defined in multiple ways, we will look specifically at similarities among compiled software or binary programs.

The first definition is similarity based on the author's style of writing program code. On a high level the style can be expressed in terms of how a programmer structures their software. For instance some programmers may prefer larger functions, such as a large *main* function, while others are very diligent in splitting each of their programmers functionality in smaller seperate functions. Other examples of a style preferences are that one programmer may prefer to use for-loops over while-loops, or negate their if-statements to avoid deep nesting of their code. The style of a programmer is also influenced by the skill level of the individual programmer. This similarity measure is independent on the functionality of the program, two programs that have very different goals, can still be similar based on similarity in style. The question on whether programming style actually remains after compilation in binary programs, we leave to a seperate section in subsection 2.1.1.

Similarity regarding functionality of binaries can only be found when two or more programs attempt to (partially) do the same things, possibly in different manners. This form of similarity can be split into two subtypes. First the similarity can be measured as the literal similarity, two or more pieces of program are then said to be similar if the set of instructions a program consists of are the same, and the instructions appear in the same order. The second form of similarity is more complex to identify, it is when two or more pieces of software have the same input and same output, but use a different method of coming to the end result. Practically this means that the same goal is reached using a different set of instructions or with the same instruction but in a different order, formally this behavior is called semantic similarity. It could be caused on purpose to make two identical pieces of software from the same author look different, but could also be the result of compiling the same source code using different compilers, compiler versions, or with different optimization flags. An example of what happens when using different compiler flags is given in subsection 2.1.2.

All forms of similarities can be assessed on different granularities, we can assess the similarity on the level of the full binary, however we can also assess it in on the basis of smaller pieces within a binary, such as functions or even smaller on block level components such as by splitting a binary in loops, or the conditions of an if statement. Examples of these similarities appear when an author re-uses parts of their own work in different projects, or when the same external libraries are used among projects. By looking at similarity on a finer granularity it is possible to identify the specific components on which binaries are similar. In addition, it allows for clustering where a single sample belongs to multiple clusters as it shows similarities in multiple different places within the binary.

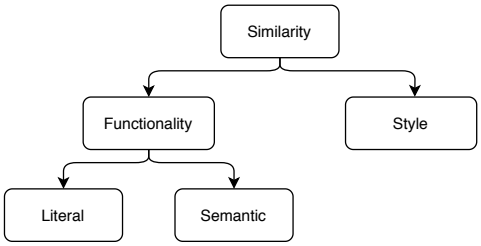


Figure 1.2: Taxonomy of similarity.

1.3. Research methods

Two methods for data extraction from software can be distinguished. The first is static analysis, and the second is dynamic analysis. Each method has its advantages and disadvantages, and result in different datasets. In Table 1.1 an overview is given of the difference in challenges between dynamic an static analysis. Below we will give a small introduction, however the analyses of malware is a topic that involves a lot of difficulties that need to be taken into account while gathering data, therefore this subject received its own chapter provided in chapter 3. Here we dive deeper in the problems revolving around analyzing malware encountered during the writing of this Thesis.

Challenge	Static analysis	Dynamic analysis
Analysis duration	Short	Long
Code obfuscation	x	
Encryption	x	
Stalling		x
Instrumentation detection		x
Environment detection		x

Table 1.1: Overview of difference in challenges between static and dynamic analysis.

Static analysis

Static analysis is generally easier to do in practice for large batches of binaries, because it is not restricted by the execution time of the program or complex and uncertain environment conditions the program needs to run in. Defences malware developers put in to make this process slower are the packing of malware (done by packers), in which they encrypt large parts of their binary or they use obfuscation tools to make the actual malware code analysis harder. Obfuscation changes a program in such a way that the input and output of the program remain identical to the non-obfuscated version of the program, while the program is much harder to understand from the code. A simple example of obfuscation is the insertion of dead code, the extra instructions do not affect the malware behavior but makes the malware its code longer so it is harder to determine the programs actual behavior.

Dynamic analysis

Given the obstacles malware builders put into their software to make static analysis harder and the lack of static information that remains after compilation we also explore author attribution dynamically. Just as in static analysis a defence against dynamic analysis is obfuscating what the malware actually does, however inserting dead code example from above is much less useful now because this behavior will simply be skipped during execution. Dynamic analysis is able to deal with packers and code obfuscation since the binary itself will take care of this during execution. The downside is that there are other measures malware developers use to make dynamic analysis harder. Instead, malware developers try stop their malware from being instrumented, examples of this are detecting whether the malware runs in a virtual machine, whether a debugger is attached to the malware process, or by waiting during execution such that an automated analysis platform may timeout the analysis before the actual malware execution even started.

Feature selection

The identifiable features we are interested in concentrate around detecting code style and similar code samples, because features such as ip addresses or strings in the binary code can easily be changed between samples and can be obfuscated by custom functions in the code. Code style of the author or behavioral characteristics however, cannot easily be changed. The code style of an author entails things such as whether the author favors C++ vectors over C-style arrays or how well an author handles file operations, such as whether the author efficiently handles file handle for the same file or if he opens and closes a handle for each operation. Further behavioral characteristics have to largely remain the same if the goal between two programs remains the same, certain libraries are used or parts of the code are re-used between projects. In practice attribution based on binary code fragments has been applied on several well known malware samples. One of these malwares is WannaCry, a cryptolocker malware, which has been found to to contain code fragments previously seen in software created by the malware development by Lazarus Group.

1.4. Research questions

In this research we would like attribute pieces of binary code to a specific author in an unsupervised manor to make it extendable to to author attribution of malware binaries. The reason for choosing an unsupervised technique is that with malware binaries labels identifying the author are either uncertain or do not exist al all. The main research question is therefore:

Main research question

How can we automatically attribute authorship among recent malware binaries using either static or dynamic analysis?

This main research question can be divided in the following 5 sub-questions. Sub-question 1, 2, and 3, will be discussed in the chapter 4. While sub-question 4 and 5 will be discussed in chapter 5. For the evaluation of the methods we try to use relatively recent malware samples, such that the results are actually relevent with current malware technology. In addition, we use "high-profile" malware such that the

Sub-question 1

Is it possible to find similarities among malware binaries using an authors coding style information?

Sub-question 1: Previous research concluded that it was possible to learn a model of a programmer's coding style after compilation in binaries. Based on the work from this literature we will assess if what was learned actually captures coding style and not something else. Possibly debunking the viability of learning coding-style from malware binaries, and therefore the infeasibility of clustering malware based on this feature.

Sub-question 2

What features that help attribute authorship can be extracted from recent malware binaries using static analysis?

Sub-question 2: A feature is a property that helps describe a characteristic of an item to be modelled. Applying this definition to our subject, we want to find a property that can efficiently be used to model a single malware binary such that it can be compared to other malware binaries. Starting from the features used previously we will evaluate how well they work for static binary similarity, and what has to be done to improve the feature.

Sub-question 3

How can similarities among malware binaries be found using features statically derived from recent malware binaries?

Sub-question 3: Finding the features alone is not enough to do the analysis. In addition to the feature a (distance) metric to compare the statically extracted features from different malware with each other. Using the feature(s) found in sub-question 3 we will try to answer how to evaluate similarity among multiple malware binaries on a as fine as possible granularity.

Sub-question 4

What features that help attribute authorship can be extracted from recent malware binaries using dynamic analysis?

Sub-question 4: This question is comparable with sub-question 2, although now the feature will now be extracted dynamically. Data from malware can be extracted in a large scala of manors dynamically, we will discuss these options, and argue why the chosen datatype is deemed best. Finally, we will discuss the method of feature extraction from the data.

Sub-question 5

How can similarities among malware binaries be found using features dynamically derived from recent malware binaries?

Sub-question 5: This is the dynamic analysis version of sub-question 3, just as with the static analysis we like to get the similarities on a as small as possible granularity. We will use the dynamic features derived from the malware as described in sub-question 4, and use non-conventional clustering method to find similarities among samples of different malware families.

1.5. Scope

As with all research projects related to malware this research will only cover currently available malware. An effort was made to use currently relevant malware such that the research could theoretically be used in practice, instead of being just a proof of concept. However, new malware might use new obfuscation methods which could result in that the proposed methods in this project no longer work on newer malware samples.

When similarities are found between malware we have to keep in mind that a developer did this on purpose to shift the suspicion to another person or group. For instance by adding code fragments of other malware to their own malware. Whether attribution to a specific person, group or country is correct and not in fact an attempt to put blame on someone else, is out of the scope of this research.

1.6. Contributions

This thesis will make several contributions to the field of binary analysis on malware. Starting from current research on author attribution we evaluated how well these methods work on recent malware samples. In addition, we provide multiple methods for malware clustering that is are dependent on complicated environments.

- An overview of what makes attribution and similarity identification harder on malware binaries than on non-malware software. During this research a lot of malware related things ought to be taken into account before the data could be gathered and analysis of the malware samples could even take place.
- Evaluation of methods from previous literature for author attribution on recent malware binaries. Static author attribution has mostly been on non-malware datasets.
- Clustering evaluation of inter malware family similarity on recent and high profile malware samples. This is done using both statically available features and as well as dynamic features in the form of system calls.
- An addition to the Cuckoo malware analysis system that is able to generate a dataset of system calls of the full system tagged with process and thread id of malware samples. This dataset could be used for clustering similar processes based on system calls.
- A system that works on the most recent version of the currently most used operating system and is relatively easy to put into practice and is maintainable over time. Past studies could no longer be build upon or even replicated since the required tools are no longer online or maintained.

1.7. Summary of results

We analyzed the same dataset both using a static analysis technique and a dynamic analysis technique and found that they can result in very different results for the same dataset. For static analysis we made the choice to analyze malware on function level of instruction sequences. For dynamic analysis, after taking in consideration evasion techniques, such as process injection, we made the decision to use Windows system calls/library-calls as the source of information of a malware their behavior. By using a low granularity on which similarities are calculated, we showed that we are able to more effectively find what parts are similar and why they are similar, than when treating then a malware binary as one blob of data. We were able to find several new connections between previously unconnected malware families using the developed technique.

1.8. Structure of this report

This report starts by summarizing current literature on author attribution and similarity analysis among malware binaries in chapter 2, from this past research the research gap for this new research is identified. Then in chapter 3 we go over all relevant things to keep in mind while doing scientific research on malware. It explains some of the measure malware developers take to avoid or hinder analysis and that could influence the research results. Next the thesis can be split in two main parts, the first one is analysis done using static program analysis and the second is analysis done using dynamic program analysis. In chapter 4 we start with evaluating malware using static analysis. Where we use a novel method based on system call traces to try to answer the main research question. After the static analysis we switch to dynamic analysis in chapter 5, where sequences of system calls are used to cluster similar malware together. Finally, in chapter 6 the conclusion of the entire Thesis will be summarized, and the answers to the research questions are formulated.

2

Related work

This literature review has two main goals, the first is to identify the current state of the art regarding author attribution or finding similarities between malware. The second goal is to identify features previously used for other purposes than binary similarity such as anomaly detection, but could potentially be adapted to fit the objective of finding similarities. Both goals are separately explored for static analysis and dynamic analysis.

2.1. Static analysis

We make a distinction between work that claims to capture code style and work that does binary similarity. Author attribution based on code style is different from just finding similarities among binaries based on code re-use, in the sense that we want to be able to attribute malware to an author independent of the program's functionality. This approach is comparable with study of stylometry for text in natural language.

2.1.1. Coding style

Before moving to source code, we can look at the study of writing style of natural text, also called stylometry, which existed before computer source code even existed. With stylometry an author of a piece of text can be attributed based on an author's writing style instead of the contents of the text [51]. Examples of an author's style that can be extracted from their text are the choice of words which is influenced by an author's vocabulary, average length of sentences, or the use of punctuation. Computer program source code is still text, but is structured very differently compared to natural text, since the goal of the text is different. The purpose of the text is not to tell a story, but to compute something, or rather to provide a recipe for the processor to computer something. Although source code is bound by some pre-defined structural properties and fixed keywords most source code languages still allow for flexibility regarding the exact structure of a program. In addition to structural choice, source code still consists of user defined words, such as variable names and function names, the choice of which can be heavily influenced by the author's taste, just as it is for natural text. Because of these similarities with natural text, stylometry, has also been applied successfully in source code for the purpose of author attribution [31].

Source code cannot be used directly on a computer's processor unit (CPU). Before computation the source code needs to be compiled to binary code. In Figure 2.1, we provide an example of the process of compiling from (1) source code to (2) binary code (compiling), and making the binary code readable again using a process called (3) *disassembling* to retrieve the instructions that are sent to the CPU. There are multiple tools available to generate this disassembly, examples are IDA[1], Ghidra[2], or Radare2[3]. Different disassembler may produce different binary code for the same binary code. We will use Radare2 for disassembling during this thesis, as it is open source and provides an Application Programming Interface (API) which helps to disassemble large datasets, by connecting it to the rest of application.

The compilation process, however removes many of the features that the stylometric author attribution of source code relies on. Examples of these features that a compiler (potentially) removes are code comments, variable and function naming style, usage of whitespace, but also code specific things such as that a while and

[1] `https://www.hex-rays.com/products/ida/`
[2] `https://github.com/NationalSecurityAgency/ghidra`
[3] `https://rada.re/n/`

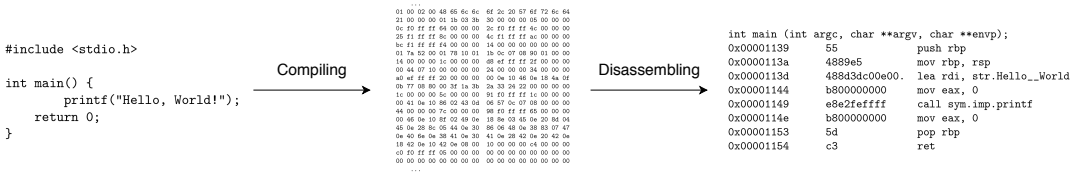


Figure 2.1: On the left we see that the source code is compiled to binary representation, after dissasmbling we are left with the output on the right.

for loop produce the exact same binary code after compilation as demonstrated by Figure 2.2. The following works explore the question on whether a compiler leaves enough (different) features unique to an author to still attribute binary code to the writer of the source code.

```
int while_loop() {
    int x = 0;
    while (x < 10) {
        x++;
    }
    return x;
}

push rbp
mov rbp, rsp
cmp dword [var_4h], 9
jg 0x12cf
add dword [var_4h], 1
jmp 0x12c3
mov eax, dword [var_4h]
pop rbp
ret
```

```
int for_loop() {
    int x;
    for (x = 0; x < 10; x++) {

    }
    return x;
}

push rbp
mov rbp, rsp
cmp dword [var_4h], 9
jg 0x12eb
add dword [var_4h], 1
jmp 0x12df
mov eax, dword [var_4h]
pop rbp
ret
```

Figure 2.2: Comparison of for and while loop after compilation use the GCC compiler. From the disassembled output after compilation (starting with *push rbp*) can be seen that only the memory locations of the jump instruction (*jg* and *jmp*) are different between the two functions.

In previous research discussed below, author attribution specifically has only been done statically, instead of dynamically on malware binaries. This is probably the most logical solution if the source code is not obfuscated, because of the scalability advantage of static analysis over dynamic analysis due to the speed of the analysis. Since static analysis is generally quicker than dynamic analysis, because the program does not have to be executed.

The first paper on the subject of attribution of binaries ever published was by Rosenblum et al. [62]. They used several features in combination with an SVM classifier. The features they used were Graphlets, Control flow Graphs, N-grams and idioms which are N-grams containing a wildcard. From their results it becomes clear that N-grams based features were the most important feature. To focus the attribution on user-written code they removed library functions after compilation. The highest reported accuracy for classification is 78% with 20 authors and 16 samples per author. They also attempted clustering using K-means, using up to 60 different authors with an undefined set of samples per author and achieved and Adjust Mutual Index value of 0.4.

Caliskan et al. [22] attempted to improve on the Rosenblum by using besides N-grams of of the disassembled code like Rosenblum, N-grams of the abstract syntax tree which was acquired by decompiling the binaries using IDA hex-rays. On a dataset of 100 programmers, they reported an accuracy between 65% with one training sample and 96% for 8 training samples on the GCJ dataset. Instead of SVM like Rosenblum, they used Random Forrest for the classification.

Meng et al. [48] went further by doing author attribution of binaries that were developed by multiple authors, classifying for each block of code the author. They introduced four new features. The first is the number of instruction prefixes used, the second are operand features which may show data access and data type patterns. The third feature is the number of constant values. In addition to the instruction features, control flow, dataflow and context features were used. For classification SVM on dataset sourced from github was used.

2.1.2. Binary similarity

In this subsection we discuss works that uses functionality of the code to find the similarity, instead of similarity based on the style of the code.

Diffing

When trying to solve the problem of author attribution as the similarity of binary code fragments, much more research than on similarity as code style has been conducted. Some research has the focus of finding small differences among binaries instead of finding small similarities among binaries. Examples of these are *BinDiff* [33] which constructs graph isomorphisms by matching tuples of graph nodes between two graphs greedily from one match to the other. First on call graphs and once these are found, on the CFGs. When no match can be found the node in one of the graphs is considered a difference between the graphs. In order to reduce the number of comparisons the sets of tuples are reduced, such as by only selecting nodes that have matching in and out edges or similar function names.

BinHunt has the goal as *BinDiff*, as it searches for binary differences instead of similarities. It improves on *BinDiff* by using symbolic execution and theorum proving.

Hashing

Hashing is used in similarity analysis to reduce the computational complexity of finding similarities among all binary samples and for reducing the feature space. Instead of comparing all samples with each other which requires $O(N^2)$ time for N samples, the features of a binary samples are hashed. Hashing can reduce the features space by causing hash collisions for features with different values. This is called locality sensitive hashing Jing [35], in the form of min hashing to find semantic similarities. Jang et al. uses hashes to reduce the feature space [32]. They experimented with this technique for reducing N-gram feature vectors to a hash value.

Semantic analysis

The semantical equivalence is determined on the level of registers, meaning such equivelence checker is able to check equivalence of two sequences of instructions even when the register values are different. In addition, information that could help during a semantic analysis from sequences of instructions, includes information such as argument types and memory addresses. This could for instance happen when using different compiler versions or different compiler flags between two on two identical pieces of source code. An example of this is illustrated in Figure 2.3, where a piece of code is compiled with different compilation flags.

Another reason for wanting to find similarities on a semantic level is the use of code obfuscators. Jin [35] used min hashing on the input and output behavior of basic blocks. The semantics are captured as four components on a basic block. (1) The effect on the registers, the effect on memory, if the basic block contains a function call, (3) the arguments of the function of b, and if the block contains a jump (3) the branch condition of this jump. Lakhotia [44] does something similar except they added an additional step in order to also make the literal register names generalized, leaving a symbolic representation of the blocks.

2.2. Dynamic analysis

There are various methods for identifying similarities with dynamic analysis, in the following sub-sections we will focus on the methods in literature that use system calls in their method. This type of feature has already been used in past literature for malware analysis tasks. They used distributions, sequence alignment, graph matching and frequent patterns.

2.2.1. Distributions

We identified literature that directly computes the difference between two system call traces based on a distance metric, or a distance based on the distribution of (subsequences) of system call traces. From the system

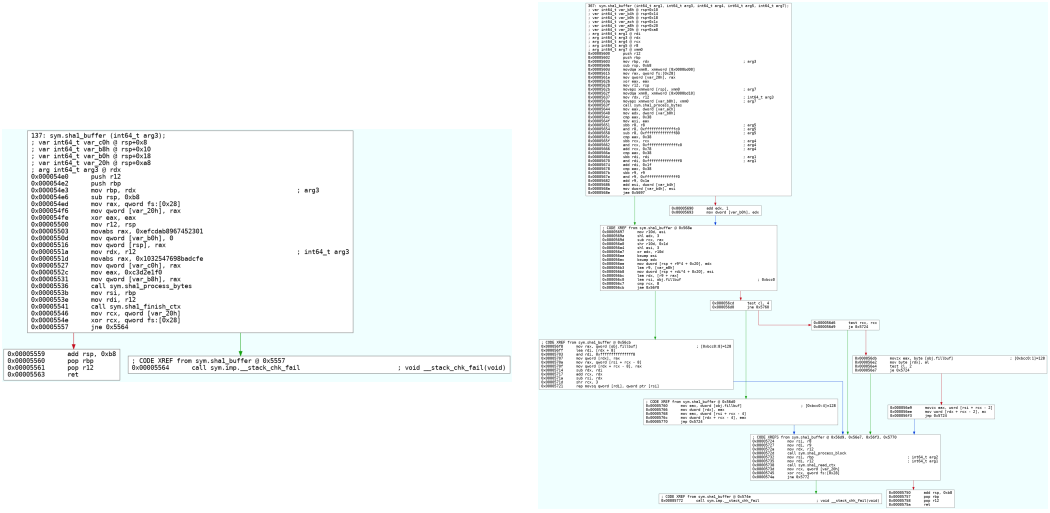


Figure 2.3: Comparison of a function from the GNU Coreutils[4] SHA1 binary compiled with different flags, on the left compiled without optimization, and on the right compiled with optimization flag *-O3*, which optimizes for execution time.

call traces a feature vector is generated, the full ordering in which the system calls appear is reduced to subsets of system calls in the form of n-grams. These still capture some of the ordering in which the system calls appeared, and have the advantage that they can be converted to finite feature vectors.

Canzanese [24] used n-gram based features from system call traces, and used it classify malware samples. They found that n-grams with the length 3 in combination with the random forrest algorithm produced the best results in terms of F1 score. Interestingly they did not use the full system call traces produced by the malware, but only the first 1500 calls. They argue that most of the interesting malware characteristics happen during these first system calls, such as host configuration modifications, communication with remote servers and new process creation. More interesting, since we want to do malware attribution, is clustering. Apel [17] had the goal to find relations among malware samples of polymorphic malware. They experimented with edit distance, a measurement based on Kolmogorov complexity. The best method they found was the use n-gram based vectors, they experimented with multiple sizes and found experimentally, just like [24] that 3-grams gave the best clusters. For the distance metric between the vectors they used Manhatten distance.

2.2.2. Sequence alignment

Sequence alignment is a process that is used to find similar subsequences among two or more larger sequences. It is an important subject in the field of biology where it is used for aligning DNA/RNA sequences. Sequences are aligned to maximize a score that is based on on the number of gaps in the aligned sequences, and the number of mismatched and matched elements of the sequences. The pairwise similarity between two traces of system calls is however quite expensive since all pairwise alignments have to be done for the entire dataset. Comparing all traces with each other has therefore a time complexity of $O(N^2)$ for the number of sequences N.

There are two types of sequence alignment, global and local alignment. In global alignment the entire sequence is aligned, with the possibility of gaps, while in local alignment the goal is to match parts of one sequence to parts of the other sequence, it optimizes for local regions to have the highest possible similarity. Famous algorithms, and still relevant algorithms are the Smith and Waterman algorithm [67], for local alignment and the Needleman–Wunsch algorithm [52], for global alignment.

System calls have also been studied in various works regarding malware detection [37, 38, 72]. They all use a popular modern DNA/RNA sequence alignment algorithms from biology and directly apply it to sequences of system calls to find difference between regular and benign and malicious behavior. Research by Kirat and Vigna [39] used alignment of system call sequences to search for evasive signatures in malware with their system Malgene. Given two system call traces of a malware sample, one executed on a non-cloaked virtual machine and one on a bare metal system. They align the traces to find the evasive point in the trace from the virtual machine. To improve the system call alignment it uses the notion of *critical system calls*, these are calls that are more important for the alignment than other system calls. Examples of these are system calls related to file operations, register events or process/thread events. Another improvement is made by BinSim [50]

which extends the sequence alignment for system calls, with Symbolic execution to find differences between two binaries. The advantage of using symbolic execution is that it is possible to gather all possible paths of a malware binary, however the disadvantage is that the system to acquire the data becomes very complex and symbolic execution may not work for all malware.

2.2.3. Graph matching

Graphs can be constructed from subsequent system calls, depending on the amount off additional data in addition to just the system call index numbers, they can be constructed as a Markov chain or as complete functional graph which requires keeping track of the program state.

Park [57] used system calls to do malware family classification by comparing the maximum common subgraph. However, finding the maximum common subgraph between two graphs is a known NP-hard problem, so this method evidently has scaling issues. Nikolopoulos [53] did unsupervised malware classification using system call dependencies. They created Markov chains by grouping sequences of system calls to a specific set of types. This is done by using several well known distance and similarity metrics using characteristics of the Markov chain, such as the weights and in-out degrees for corresponding nodes to all families currently in the corpus. If the distance is the closest between the sample and a certain family it is classified as this family, and moves on to the classification.

2.2.4. Frequent patterns

Frequent patterns in system call traces are sub-sequences of system calls that appear more often among similar malware samples than among non-similar malware. They can be used to mine correlations, associations or features from a database consisting of sequences. The mined frequent item sets can be used as a feature selection technique as done by Sami [64], use the patterns as feature directly as done by [76], or to create features as shown by Qiao [60], who first mined frequent item sets, then sorted these by support to generate n-gram vectors. From these n-gram vectors the distance between two sequences was computed. We question whether n-grams were the right choice, since n-grams do not yield much information here since they will maximally appear once for each sequence. In addition the number of features can be huge depending on the number of unique frequent item sets.

Not targeted at malware specifically, but text clustering using frequent item sets has been explored in [78] and [66]. However, these cluster whole documents, which may not be what we want since the granularity of a whole document, or in our case malware sample is too big. It may be possible that similarities are only visible on a finer granularity, such that a single malware sample may lie in multiple clusters, for instance if it combines two parts from two different malware samples that only share code through this sample.

We can compare frequent pattern mining and local alignment, the difference is that in local sequence alignment the subsequences are still in the order they appeared in. For malware traces this is not a logical assumption, a function in one malware may be called in the beginning of the execution while in another malware the function is called at the end of the execution. Frequent sequential set mining does not take this global ordering of local alignments into account, instead patterns between traces can be matched independent of their location in the system call trace.

2.2.5. Current methods of API call sequence extraction systems and techniques

A popular method for acquiring system calls from malware is done using a modified QEMU emulator. By instrumenting the virtual machine itself the whole system can be monitored instead of specifically injected binaries which is the way Cuckoo does its instrumentation. Examples of systems built on QEMU are TEMU [77], VMScope[34] and TTAnalyze[20]. Another approach for extracting system behavior is by using a hardware virtualization extension as is done by the Ether analysis system [28]. Between each instruction executed in the virtual machine the analysis platform causes and handles a debug exception as the instructions are transferred between the guest and the host system.

Another method used to help increase the amount of malware related information by countering anti analysis techniques is done by tainting the execution [36, 50]. By doing this the malware related processes can be traced through the system even when the malware uses techniques such as process injection. The practical problem with this technique is however that it is technically challenging since the virtual machine environment (QEMU) needs to be adapted to make this possible.

2.3. Research gap

Given the above literature we will discuss identified problems currently not addressed in the literature, in other words the research gap. We have divided this into three subsections, first we discuss the static analysis, second dynamic analysis, and third we shortly discuss general observations applicable to both sub-fields.

2.3.1. Static analysis

Author attribution has been shown to work using mostly the same method for larger and larger datasets over time. These datasets however mostly consist of non-malware software. Such that the author of the software did not take effort to make static analysis harder.

Regarding code style similarity for finding the author of a program, previous work calculating code style similarity over the full binary, however this method assumes most of the binary has been written by the same author. In practice two reasons can be identified on why this is not the case. Firstly, as identified by the previous work an author may use library functions, these were filtered out based on the name in their work. However, when an author removes function names from the binary, as a malware developer is likely to do, these functions can no longer be identified and therefore no longer filtered out. Secondly, an author may copy and paste parts of code from other projects that are not theirs, these parts cannot be filtered out at all and get wrongly attributed to the author of the rest of the code of the currently analyzed binary.

We should also make a distinction between the goal of finding similarities in a dataset where most of the binaries are the same and finding similarities in a dataset of binaries where the samples are completely different and the goal is to find only similar parts. Finding samples in a dataset where the samples are mostly the same is in practice used for finding a patched vulnerability between two versions of a program. In malware, this method could be useful for detecting changes between malware versions. For malware analysts, this is useful since they now only the new part of the malware needs to be analyzed again. For author attribution on malware this method is however not that useful, and we are therefore more interested in finding similarities between malware samples that are very different.

Looking further at previous work on at binary similarity as in semantic similarity, we see a large scala of quite complex methods. Things such as symbolic execution are interesting solutions in theory, for malware however, these methods are hard to put to practice. The reason for using these complex methods is to resolve the problem of finding similarities among binary code even when different compilers are used. However, it is questionable whether this is really an issue, certainly in the case of author attribution, where could be argued that a single author is unlikely to use different compilers for different projects. The number of C++ compilers for Windows is minimal, besides the default Windows Visual C++ compiler, there is only one real competitive compiler called Clang. However, since Visual C++ is the default compiler, it is more likely to be used by most projects.

2.3.2. Dynamic analysis

Regarding dynamic analysis we see that system call traces have been used in the past for malware similarity analysis. However, previous research assumed the relevant malware trace on the system was known in advance and all malware child malware processes start from this process. In practice this is not realistic, certainly not if the correct malware trace(s) need to be identified for all samples in large datasets. Although a process (thread) cannot spawn out of nothing it is possible to hide which process actually initiated the start of a new process by launching the new process under the parent of a different process. Since this is a common technique among malware to hide its activity a lot of information regarding the malware would not be traced or analyzed as malware process.

Secondly, all of the above dynamic analysis techniques for malware similarity compare full traces, instead of partial system call sequences. A similarity match between two binaries will thus only be found when the binaries are almost completely equal. The granularity of these analyses is thus quite high, which closes of the opportunity of finding smaller similarities that would appear when two binaries are from the same author but have different functionalities. We want to find similarities such as a few shared functions or the use of a certain library in both applications.

Thirdly, all recent works focus on malware detection, classifying or clustering malware families and not on inter-family similarities between malware from different malware families or authors. This is due to the lack of good labels for both authors and similarities.

Fourthly a practical matter regarding dynamic analysis are the systems used for gathering the data. All the systems used by the research discussed above to gather the data are no longer supported by their authors, and

given their complexity are not easy to redevelop for individual research. Examples of these are Binsim [50] which uses Bitblaze [68] a system not maintained since 2010. Other such systems are Anubis (website offline), used in [28], and used in various highly cited papers such as [40, 41], or another platform called Ether [28] which is also unmaintained since 2009. Even in the unlikely case these systems still work with newer host systems, they will not work with newer guest systems. No scientific malware research that uses more recent versions of Windows such as Windows 10 can be found to date. Some recent research still uses Windows XP for their malware experiments [36], this questions how well these results represent the real world, since Windows XP has been succeeded by numerous newer versions and already had only a world wide market share 5% in Januari 2017[5]. The version of the operating system is relevant since malware developers only target newer operating system versions, since they expose certain vulnerabilities that are not available in older systems. Another possible reason is that malware developers also know that analysis platforms generally run older operating system versions and therefore evade these systems.

2.3.3. General

Another important point can be made regarding the datasets used. It is often unclear what the quality of the acquired malware datasets is, although this has a lot of consequences for value of the results of the research. Older malware is easier to analyze or find patterns in than newer more advanced malware variants. Another influence is the level of skill a malware developer has, since a developer with more skill has better ability to implement measures to make analysis of their code harder. These issues are more elaborately discussed in section 3.1.

2.4. Summary

The two approaches, static and dynamic analysis are clearly separated in the related work. We started with work that used coding style as a measure to find similar binary samples. All of which use supervised classification on statically acquired features. After this we continued looking at work that uses the functionality of a program as its definition for similarity. For static analysis, we discussed works that used techniques such as diffing, hashing and semantic analysis. Regarding dynamic analysis, the types of features are more dives, however we put the focus on works that used system call sequences as their feature. Works that used n-gram distributions, sequence alignment, graph matching and frequent patterns were discussed. Lastly, regarding dynamic analysis, since the extraction of system mcalls is not a straightforward task, methods from past works were researched. Finally, the research gap for both static and dynamic analysis is reviewed, which gave the further direction for this Thesis.

[5] `https://gs.statcounter.com/windows-version-market-share/`

3

Theory

In this chapter, the difficulties with analyzing or reverse engineering Malware will be explained, and the consequences this has for the scientific value of the experiments. First, we will discuss techniques malware developers use to make the analysis more difficult or even near impossible. These are relevant to explain since it can affect the final research results and therefore the conclusions.

In addition, we will discuss the fundamental metrics and algorithms used in this Thesis in section 3.2.

3.1. Malware

Conducting scientific research on malware is difficult because there are a lot of uncertainties with the dataset. Questions such as:

- Which malware samples do we use? A large variety of samples can be found on the internet however not all samples should be treated equally. Some malware may be made by more advanced or serious developers as a means of living while others are just created to annoy a friend. These two certainly do not fall into the same category, but how is this determined?
- What is the quality of the data acquired from the malware? There are some websites that collect and index malware samples by their malware family, but often it is unclear or unknown who or what assigned the labels to the samples. Even the label names themselves are inconsistent among anti-virus companies.
- Does the malware still work or is it broken since the required command and control server is no longer online? If malware is dependent on some external server and fails to function without it the original malware behavior can no longer be observed. We might not even see any malicious behavior anymore.
- When executing a malware sample, is the observed behavior the actual "real" malware behavior or did the malware detect it was being analyzed and just showed dummy behavior? This could be caused by certain preconditions the malware author set are not met. Examples of such conditions are that the malware only runs at a specific time or the malware only runs if the locale of the infected PC is a specific target locale. It could also be that the malware has detected it is being analyzed and therefore shows some dummy behavior to fool the system.

All of these uncertainties follow from the fact that it is almost always unclear what malware is supposed to do without deep analysis for each and every sample.

Besides, these uncertainties it is hard to generalize malware research to all possible malware. The issue with malware is that samples can be very different from each other depending on the goal of the malware and the author's style or skill. From a research and analyst point of view, we would like to be able to analyze all malware using some generic technique, however, this is unlikely to ever be possible since each developer can create their own custom countermeasure, on top of this malware constantly evolves to adapt to new analysis techniques. We can draw the analog biological viruses, a research subject that has been around for much longer than computer viruses. These malware need to be analyzed separately because they can show completely different behavior from other malware and have different properties.

In general, most literature simply ignores this question and tries to prove that their method works by testing it on datasets containing enormous amounts of malware, however often without specifying exactly what samples they used especially for the larger datasets. In the extreme case it could very well be that a dataset of 1000 samples each that appear to be unique given their file hash, all are in fact the same malware, but packed differently.

In the following, three categories of malware evasion will be discussed, that were encountered during the writing of this Thesis and that may provide a better understanding of the things to keep in mind while doing research on malware.

3.1.1. Code obfuscation

Code obfuscation is used by malware developers to make it harder to understand for a malware analyst to understand what is going on in the malware's code. Obfuscation can manifest itself in many ways [16, 18], and there are too many options to explain them all. However, in order to get a basic understanding we will explain a couple of frequently used obfuscation techniques.

3.1.2. Data transformation

Binary code may contain strings of text, these are either for displaying when the software is executed, or to refer to system library functions. To a software analyst the location of these strings inside the binary can provide a good indication to what a part of the code does, even without really understanding the code itself. Therefore, the software developer can obfuscate these by encrypting them inside the binary and only once they are nescecary to display to the user decrypt them.

Dead code insertion

Dead code, is code inside a program that never actually gets executed. The insertion bloats the amount of code an analyst has to analyze, and makes it therefore harder to find the actual malware code during static analysis.

Another type of code insertion are opaque predicates. These are boolean functions for which the outcome is known a-priori, however these functions can be made to look very complex, making it hard for the analyst to determine it is opaque. A simple example of such an opaque function is given in Figure 3.1.

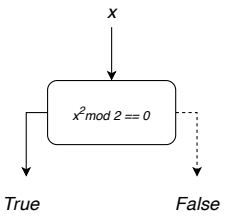


Figure 3.1: Example of an opaque predicate, independent of the input number the output will always result in $True$.

Virtualization

More advanced obfuscation techniques could wrap entire functions in a virtual interpreter. An example of such obfuscation engine is Tigress [25] with its own custom bytecode, program counter, stack pointer, and list of custom instructions. The virtual machine will then simply consist a loop that that runs over an array of instructions. A simple example of what this would practically look like is given in Figure 3.2 and algorithm 1.

Control flow flattening

Control flow flattening, as the name suggests provides obfuscation to the flow or structure of the program. On function level it works by first seperating the basic blocks of a function from each other, each block is then placed as the case of a switch statement. This switch statement acts as a dispatcher and is called indefinitely by the use of a surrounding loop. The blocks end with a reference to the switch case that contains the next block [45]. The result of the obfuscation is that there will no longer be a direct link between the basic blocks, since the switch statement always sits between them.

```
int fun(int x) {
    int z = 3;
    int y = z + x;
    return y;
}
```

```
int fun(int x) {
   // int z = 3;
   7:   c7 45 f8 03 00 00 00    mov    arg1,0x3

   // int y = z + x;
    e:   8b 55 f8                mov    arg1,arg2
   11:   8b 45 ec                mov    arg1,arg2
   14:   01 d0                   add    arg1,arg2
   16:   89 45 fc                mov    arg1,arg2

   // return y;
         c3                      ret
}
```

Figure 3.2: On the left an example program in C, on the right the assembly of the same program (Stack operations have been removed from the assembly)

Algorithm 1: Example of a function simulating a virtual machine for the function Figure 3.2. *pc* represents the program counter, the program is stored in as an integer array.

```
pc ← 0;
while True do
    instruction ← getInstruction(pc);
    switch instruction.operator do
        case MOV do
            instruction.arg1 ← instruction.arg2;
            pc ← pc + 4;
            break;
        end
        case ADD do
            instruction.arg1 ← instruction.arg1 + instruction.arg2;
            pc ← pc + 4;
            break;
        end
        case RET do
            pc ← pc + 1;
            break;
        end
    end
end
```

3.1.3. Packing

The second category is related to code obfuscation, however the binary is now obfuscated as a whole. As the name suggests the malware is packaged within a layer of obscurity. The original binary is encrypted and is no longer understandable as it looks like random data. Random data is of course not executable on the system, therefore a small routine is added to the malware that is able to extract or *unpack* the encrypted malware data. The malware analyst can try to reverse engineer this routine in order to extract the obfuscated malware data, as the malware can only fully be analyzed when in an unpacked state. For some common packers, reverse packers have been developed however if the developer of the malware has developed its own packer and the packer is a one of a kind it may be easier to just let the malware unpack itself by executing the malware on a sandboxed system. Once the malware has unpacked itself the memory of the unpacked malware executable can be dumped. For analysis, we can then make a RAM memory dump to acquire the malware sample in its unpacked state.

Acquiring the malware from memory is not a straightforward task. Unfortunately, even after the malware binary is unpacked it may not lead to a single binary that is ready for analysis and explains the workings of the malware. To avoid detection malware can use a technique called *process injection*. With this technique, malware runs its code in the address space of another (benign) process. To get the malware from the RAM dump, the dumped processes must be scanned for possible malicious injections and once the injected part is found the injected malware needs to be extracted from the process.

3.1.4. Runtime evasion

Process injection

Besides being destination to unpack itself in, process injection is also good way to evade detection while running. Process injection is a technique in which one process injects a thread into another third-party process. This is hard to detect since the parent thread and process of this new thread appear to be from the third-party process and not from the process that injected the thread. Korczynski [42] reported in their research that out of their tested malware the detected that 23.23% uses some form of process injection, therefore we cannot ignore this problem. To detect the existence of the malware a previously completely benign system process now needs to be scanned for malicious behavior. For malware analysis this makes the job extra hard since there are many stealthy techniques to do this injection. There are multiple techniques malware makers use to do process injection, on Windows these are possible without special permissions, while on Linux the same can be done once certain security features are disabled. The simplest technique for process injection is by DLL injection, this is done by first loading the DLL using `VirtualAllocEx`. Once the DLL is loaded to memory we can simply create a new thread in the target process by letting the malicious process execute `CreateRemoteThread` from the Windows API, to have another process execute the code contained in the DLL. There are a however much more advanced techniques that are currently functional on the latest Windows versions, examples of these are injection using the ALPC, Atom Bombing, or Stack Bombing [43]. If the injection is not recognized, and we still want to be certain all malicious behavior is monitored, we need to monitor the complete system, which greatly increases the amount of data to be analyzed.

System call monitoring evasion

One way of monitoring what a malware is doing or to detect malicious behavior is by monitoring a binaries interaction with the operating system, these interactions are necessary to make any changes on the system, such as file operations, register operations or process/thread creation. Developers of malware know this interaction with operating system gives away information about the malware its behavior and they therefore attempt to make these operations a stealth as possible. Malware can stop the ability of monitoring Windows API calls by manually loading the Windows API DLL (`ntdll.dll`) and extracting the system call numbers itself. After extraction the system call can be made by directly calling the system call index. It does this by loading the number of the system call into the `eax` register of the processor and then execute the `syscall` instruction (on 64-bit systems).

By doing this they circumvent at least three analysis techniques. The first is that during static analysis of the binary specific imported system calls are no longer visible. Normally when a Windows program uses library calls they are shown in the *imported functions* table in of the PE files header. However when using above technique this is no longer necessary, and therefore this table can remain largely empty.

The second thing the malware developer circumvents is that it is no longer possible to easily hook these library calls with custom ones. This could be desirable to for instance add code to functions that contain instrumentation code.

Thirdly doing system calls manually also stops the Windows event callbacks from triggering, therefore tools such as the official Windows Process Monitor will be unable to detect any system calls executed by the malware.

The fact that malware does this can be demonstrated by observing the difference in the count of unique system calls between the statically imported table and the system calls made dynamically. By using the Malpedia dataset of dumped malware binaries from subsection 4.1.2, and retrieving the imported functions table using Lief [71]. We observe that only 104 samples out of the total 750 samples have an imported functions table that actually contains functions.

System call obfuscation

In [49] propose an attack to make analysis using system calls harder. By replacing system calls dynamically with semantically similar system calls, the same malware has a different system call trace while exercising the same behavior. A simple example of this would be to insert a `NtSetInformationFile` that does not change anything on the file call between a `NtCreateFile` and `NtClose`. However, such an attack is still to be exposed in the wild.

Stalling

Evasion of analysis can also be done by delaying the execution of the malware by sleeping to let the timeout in sandbox systems stop the analysis before the execution even started. Oyama studied a variety of sleep behavior in malware [55], low level sleeps using an the `NtDelayExecution` is relatively easy to detect when monitoring system calls, higher level sleep functions such as `Sleep` and `SleepEx` call this same lower level function. However, he notes that sleep can also be achieved using stealthier techniques than calling these specific sleep functions. For instance by using dummy instructions, which could be achieved by calling a certain function a specific non-destructive function for a certain amount of times, causing the system to virtually hang on these calls for the duration of the calls. This subject has not yet been researched.

Analysis detection

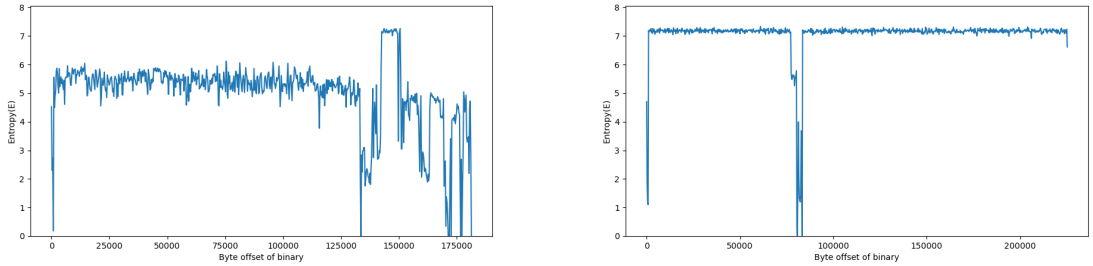
In order to follow the execution of a program, a debugger can be attached. However, a debugger can be detected by the to be debugged application. In Windows there is even the system call `IsDebuggerPresent` that returns whether a process is currently being debugged. Another method is to detect breakpoints by monitoring the system time. A breakpoint will cause the program to pause, the debugged program can detect this be checking execution time between parts of the program. If too much time has passed between two points during execution it may have been paused with a breakpoint.

Another tool security researchers use for dynamic analysis are virtual machines such as Virtualbox or Qemu. With these, the behavior of malware can be observed while no actual systems are threatened. An additional benefit of virtual systems is that they can easily be reset to an uninfected state such that multiple malware samples can be efficiently analyzed after each other. There are however ways for software to detect is running on a Virtual machine. Firstly virtual machine software comes with so-called guest additions, this software is installed on the guest and used make a functionality such a shared clipboard, file share, or adaptive graphics to the window size of the guest on the host possible. This guest-software can be detected by malware by scanning the existence of certain processes, files or register entries on the system. Secondly software could probe the vendor label of the hardware such as the motherboard or ethernet adapter. Thirdly it is possible to distinguish a real CPU from an emulated one [56]. This works by generating specific instruction sequences by using a fuzzer (a tool that generates many different inputs versions based on an initial seed), that generates different results specific for a certain emulation environment. This could for instance happen when there is a bug within virtual machine that causes a certain instruction to fail.

3.1.5. Evasion detection

Packers can be relatively easily detected compared to other evasive techniques. By measuring the entropy of a binary or piece of a binary we can get a good indication on whether the malware contains packed data. Entropy is a information theoretic measure for information or "surprise" or "uncertainty", expressed as number between 0 and 1, with 0 being low uncertainty, and 1 being high uncertainty. A non-packed binary will not contain a perfectly random set of bytes, but will show a pattern, with certain bytes occuring more often than others. An entropy close to 1 indicates the binary is likely to be packed or encrypted. By splitting a binary in chunks of bytes we can visualize the entropy pattern of the full binary to determine how likely it is the binary is packed.

To show the difference in entropy between a non-packed and a packed binary we plotted the entropy of two binaries in Figure 3.3. The entropy is calculated in chunks of 256 bytes for both binaries. The packed sample shows a very high average entropy of 7 bits per byte, in addition the entropy is very even over the full binary. Based on these two factors we can conclude that the binary of Figure 3.3b is packed.



(a) Benign application, Microsoft Windows Notepad

(b) Packed malware from Globeimposter malware family

Figure 3.3: Examples showing the difference in entropy pattern between a benign non-packed binary and a packed malware binary.

Detecting code obfuscations is harder than packer detection. A well studied de-obfuscation technique is symbolic execution [65, 75] which analyzes which program inputs cause which parts of the software to execute, Banescu et al. [19] showed that symbolic execution is effective against multiple code transformations such as flattening, opaque predicates and virtualization. Salwan et al. [63] demonstrated this by simplifying programs obfuscated with related to virtualization. Guinet et al. [30] developed the tool *Arybo*, which is able to simplify mixed boolean expressions used to generate opaque predicates.

Putting symbolic execution in practice however is a challenge to attach it to malware. Besides this practical problem, symbolic execution is slow and expensive operation since there is the problem of "path explosion" [69]. Since systems using symbolic execution try to discover every possible execution path within a binary, the number of unique paths grows significantly as the binary grows.

3.2. Metrics

In this section a summary of the workings of the data analysis techniques used in the rest of this Thesis is given. We will go over n-grams and cosine similarity as they are used in both the static and dynamic analysis chapters. The t-SNE algorithm is technique published in 2008 that can be used to visualize high dimensional data, by reducing the data to a 2d or 3d map. In this thesis t-SNE is used to map the relative distances among a set of malware samples to a 2d figure. HDBSCAN is a clustering algorithm that can be use to automatically cluster the similar samples, it can be used unsupervised as it does not require the number of cluster apriori.

3.2.1. n-grams

n-grams can be used to vectorize sequence of elements. From n-grams vectors can be used to vectorize a sequence, while capturing the sequential ordering of the elements. In addition the relative frequency of the elements is captured by counting the number of occurances of an n-gram and normalzing it by the element with maximal occurance count generated n-grams from a sequence.
For example from the sequence of elements $ab \rightarrow b \rightarrow c \rightarrow b \rightarrow b$ the following n-grams can be created.

- 2-grams / bi-grams: $(a,b),(b,b),(b,c),(b,b)$
- 3-grams / tri-grams: $(a,b,b),(b,b,c),(b,c,b),(c,b,b)$

In vector form, accounting for all possible combinations elements in 2-gram form these would become:

2-grams: $[(a,a),(b,b),(c,c),(a,b),(a,c),(b,a),(b,c),(c,a),(c,b)]$

This will result in the vector: $[0,2/4,0,1/4,0,0,1/4,0,0]$

The n-gram pieces are generated using a sliding window, which moves a single element at a time. Even though the sequences can be of different lengths the conversion to vector format allows to compare the similarity

between the sequences once they are put into vectorized format. Depending on the length of the patterns that need to be captured the size of the n-grams can be varied. Longer n-grams will however result in longer vectors since the number of possible combinations grows as the the length of the n-gram increases. Which can be formally explained as v^n, where v is the number of unique elements (vocabulary) in the dataset and n being the n-gram length.

3.2.2. Cosine similarity

Cosine similarity is a similarity measure between two vectors expressed as the angle between them as in Equation 3.2.2. The similarity between two vectors is given as the angle between two vectors and taking the cosine results in a number between 0 and 1, with an outcome of 1 meaning the vectors are the same. Cosine similarity is a more logical choice for document similarity and comparing n-gram vectors, as it does not take into account the magnitude of the vectors unlike non-$L1$ normed euclidean distance. This is important when documents are not of the same length, since if a document is longer it likely increases the frequency of words and therefore increases the distance between the two documents.

$$similarity = Cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{3.1}$$

3.2.3. t-SNE

The t-SNE [46] (T-distributed Stochastic Neighbor Embedding) algorithm is a dimensionality reduction algorithm that can be used for visualizing highly dimentional data. t-SNE does not retain the distances or the density between the points, instead it keeps the probabilities. Since we use t-SNE to evaluate clustering it is important to understand its parameters. The most important parameter is the perplexity, the most appropriate value of this parameter depends on the density of the data. The denser the data the larger the required perplexity. In order to verify how "real" the visually appearing clusters are, we will verify the t-SNE plots using a clustering algorithm (HDBSCAN) that is run the original non-dimensional reducted data.

T-SNE can also be used to visualize pairwise similarity matrices directly by mapping them to 2d space. Which is what we will use t-SNE for in this thesis. This way we can use any custom pairwise distance metric instead of the default euclidean distance.

3.2.4. HDBSCAN

The HDBSCAN [23] algorithm is a clustering algorithm that uses hierarchical clustering on top of the DBSCAN algorithm. One of the main advantages of this algorithm over and algorithm such as k-means is that it is not required to provide the number clusters beforehand. In summary it works by first creating a minimum spanning tree from all datapoint using a custom metric. The metric is defined as mutual reachability distance Equation 3.2.4, where a and b are datapoints. $core_k$ gives the distance to the core of the k nearest neighbours of points a and b.

$$d_{mreach-k(a,b)=max\{core_k(a),core_k(b),d(a,b)\}} \tag{3.2}$$

After constructing the minimal spanning tree, the tree is then converted into a hierarchical tree where each split represents the dividing line between two subclusters. In order to flatten the hierarchy into actual clusters, HDBSCAN requires the minimum cluster size parameter. At each split, the algorithm will test whether one of the possible clusters at the split has fewer points than the minimum cluster size, at which point the datapoint will be "removed" from the tree. If the split does not exceed the minimum number of datapoint the points are classified as noise and do not belong to a cluster. Otherwise, if the split results in two possible clusters that have more points than the minimum cluster size and the clusters persist. The last step is then to actually extract the clusters from the filtered tree. To make the decision of extracting a cluster, the algorithm uses the notion of stability of a cluster. The stability is the distance between the point in the tree the datapoint was removed from the tree and the point in the tree when the point where it became its own cluster. If the sum of all the stabilities of the child clusters is larger than the stability of the cluster, the cluster stability is set to be the sum of the stabilities of the child clusters. Otherwise, when the stability of the parent cluster is greater than the sum of its children, the cluster is then a finalized cluster, and all of the clusters sub-clusters will not become clusters. Once the root is reached the algorithm is finished and the flat clusters returned.

3.2.5. Clustering evaluation

After developing a new distance measure, clustering technique, or new dataset, we would evaluate how well the clustering works. If given a test set for which the labels are given, the performance can be directly evaluated by comparing the predicted clusters with the expected clusters. However, often this is not available, therefore we will have to resolve to different methods.

The first method is visually inspecting the result, if the data is multi-dimensional it first has to be projected to a 2d plane using a method such as t-SNE. Although simple this method makes it hard to compare results of different algorithms.

Therefore, a number describing the quality desired. We will discuss three metrics, *purity*, the *rand index*, and *F1* measure.

Purity

Calculating *purity* consists of two steps, first the majority class for each cluster is determined, then the second step is to count the occurance of this majority class for each cluster and divide by the total amount of samples. Formally this method is described as in Equation 3.2.5, where M is the set of clusters, and D the set of classes, N is the total amount of samples.

$$\frac{1}{N}\sum_{m\in M} max_{d\in D}|m\cap d| \tag{3.3}$$

The downside of *purity* is that having a cluster for every datapoint will generate a perfect score of 1, so *purity* does not quantify the quality of the clusters against the number of clusters.

Rand index

The *rand index* (Equation 3.2.5) is simple accuracy, requiring four categories of samples, *TP*, stands for *True Positive* which are the pairs of samples assigned to the same cluster, *True Negative* (*TN*) is assigned to a pair of different samples assigned to different clusters. *False Positive* (*FP*) are pairs of different samples in the same cluster, and lastly *False Negative* (*FN*) that are similar samples that are assigned to different clusters.

The pairs are calculate by:

$$Pairs = \frac{N(N-1)}{2} \tag{3.4}$$

$$RandInd = \frac{TP+TN}{TP+FN+TN+FP} \tag{3.5}$$

F1 score

Lastly, we can use the $F1$ score (Equation 3.2.5), this score penalizes false positives slightly more than the rand index. The score is the harmonic mean of precision and recall. Precision (Equation 3.2.5) is the fraction of sample pairs that are "correctly" clustered together among all pairs samples in a cluster, while recall (Equation 3.2.5) gives the fraction of "correctly" clustered pairs of samples among pairs of samples that should have been clustered together in a cluster.

$$Precision = \frac{TP}{TP+FP} \tag{3.6}$$

$$Recall = \frac{TP}{TP+FN} \tag{3.7}$$

$$F1 = \frac{2\cdot precision\cdot recall}{precision+recall} \tag{3.8}$$

3.3. Summary

In the first section, we discussed a broad range of countermeasures malware developers may take to hinder the analysis of their malware. It is important to keep these in mind during malware research as they may explain unexpected results, and provide a precaution for not blindly trusting results and drawing conclusions too fast.

Although there has been research to circumvent anti-analysis techniques, these have not been tested on malware, possibly given their current complex impractical state.

In the second section, we summarized the methods of some general methods used during this thesis. n-grams and cosine are not the most difficult concepts, it is however important to be aware of the characteristics of t-SNE regarding how to interpret the visualizations.

4

Static analysis

Static analysis is only viable on unpacked malware, (we will show this in subsection 4.4.2). Since for packed malware, only the unpacking routine will be possible to disassemble, which is possibly not written by the author and instead generated by an external tool. Therefore we can only use datasets of binaries that are either not packed or unpacked/dumped from memory such that we are left with the unpacked state of the applications. For verification purposes, the dataset also needs to contain some ground truth for each binary of a possible author.

We will be doing static analysis using n-grams over instruction sequences of disassembled binaries. The general idea is that a programmer's programming style such as how they organize their functions remains after compilation. The similarity will be assessed using two granularities, first by measuring the distance between distributions of instruction sequences of full binaries, which are likely more focused on the style as a whole. The second granularity is similarity on the level of blocks of code such as functions, which probably capture similarities regarding code similarity.

We can only evaluate how well the metric estimates the similarity based on the label of the sample. This means that even though samples from different authors may be very similar we are unable to determine if the source code (style) of these samples is also similar. For classification, this means that a sample may be misclassified, for clustering, this means that a sample may be placed in a cluster with another author.

Nevertheless, we will attempt some engineered clustering techniques, to attempt and cluster malware samples based on function similarity. The evaluation will be done visually.

4.1. Dataset Exploration

For the following experiments, we will make use of two datasets, the first is a dataset from Googles programming challenge, Google Code Jam. The second dataset is from Malpedia [59], a freely available curated Malware repository, that has an API from which all samples can be downloaded.

4.1.1. Google Code Jam

The first dataset a set scraped from *Google Code Jam (GCJ) programming challenge*[1] which is an online an algorithmic programming challenge. This dataset has also been used in all of the previous research on author attribution to evaluate the developed methods (subsection 2.1.1). The contest is held from 2009 to now (2019), but we will only use the dataset from 2009 to 2017, since the submission website changed since 2018, which made it harder to scrape. For this challenge a contestant has to submit a single source code file that is expected to solve the algorithmic problem at hand. Because the choice of program language is open to the contestant we filtered the scraped submissions on submission written in `C`, The GCJ source code was then compiled using GCC version 9.1.0, without any optimization flags. Only authors that have a minimum of 10 submissions are selected for evaluation, which results in a total number of authors of 6724.

However, for the experiments below we only used a subset of the samples. To limit the effects of a specific dataset, the average of multiple years is used and the subset within a year is randomized. Besides, because we want to leverage that functionality of the code is the same the samples from an author are selected such that for each author the samples to the same assignment are used.

[1] `https://codingcompetitions.withgoogle.com/codejam`

4.1.2. Malpedia

The second dataset consists of a set sourced from Malpedia[59][2], this website contains a curated dataset of malware samples with some samples labeled with possible authors of the malware. Since almost all malware is packed, which hides almost all malware functionality and author information except for the unpacking routine, we will use only the samples that have a memory dumped unpacked version available. The authors of Malpedia noted that by using unpacked samples the size of the malware corpus greatly reduces since, duplicates caused by different packers or packer arguments do not cause duplicates in the database. Therefore, we argue that although the same size of the database may be smaller it is a better approach than using tens of thousands of samples which appear very similar after analysis, because they are factually the same after unpacking.

Malpedia provides author labels for the malware samples in the database, however it is possible a sample is tagged with mulitple authors. Given the messy nature of malware, there is uncertainty with the labeling.

Filtering the dataset on, just the samples that have a been published in an unpacked state on the website leaves a dataset of 2591 PE executables, see Figure 4.1 for the distribution of the dataset. Some samples could not be analyzed, Radare2 seems to fail to find any functions other than the main function for some binaries. Another issue we faced was that Radare2 appears to get stuck on some binaries. Binaries that had these issues were removed from the dataset.

Classification

For the classification we only use the samples that have a single author attached to them, which leaves us with 82 authors. In addition, we require each author to have at least 3 unique malware families. With this last condition a dataset of 50 authors, with in total 271 samples remains.

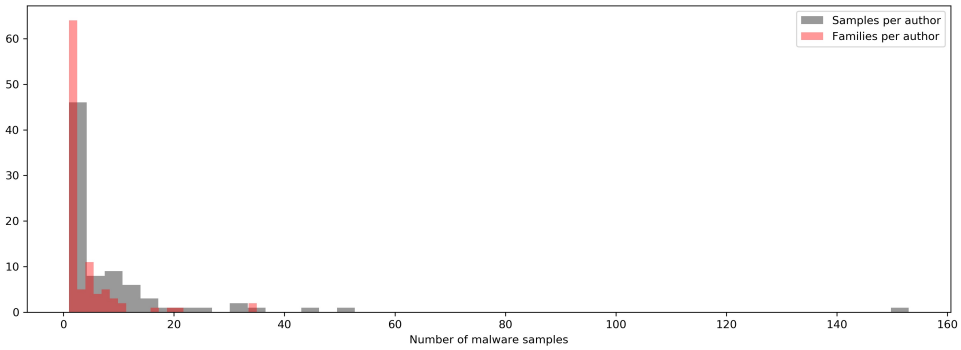


Figure 4.1: Histogram of Malpedia dataset, used for classification.

Clustering

Since clustering is used there are no real labels, therefore the quality of the clusters has to be determined visually, Using the whole dataset would result in too many data points in a graphical figure to still analyze, therefore we will take a subset of malware samples gathered from Malpedia. The used set is not a random set, but instead a set of samples that according to expert sources have indications that there are links among these malware families possible due to them having the same author. In the following, we will investigate whether we are able to find these links using the developed method. These actors currently have several malware binaries attributed to them, an overview is visible in table Table 4.1. After removing binaries from which less than 5 functions could be extracted the dataset contains 150 samples. We will shortly discuss the expected links between these malware families.

Dataset

Emotet is a modular malware, it began as banking trojan malware, it has however changed to be a method of delivery for other malware [11]. It is dependent on a command and control server to download its modules, therefore it is likely once the server is offline it will no longer function as intended. It is often seen together with *Trickbot*, as *Trickbot* uses *Emotet* as a loader for its installation [5].

[2] `https://malpedia.caad.fkie.fraunhofer.de`

Actor	Malware family	#Dumped	#Non-dumped
Mummy Spider	Dyre	1	1
Mummy Spider	Emotet (Geodo, Heodo)	10	8
TA505 (SectorJ04 Group)	Andromut (Gelup)	1	0
TA505 (SectorJ04 Group)	Flawedammy	1	2
TA505 (SectorJ04 Group)	Flawedgrace	1	1
TA505 (SectorJ04 Group)	Get2 (FRIENDSPEAK)	3	3
TA505 (SectorJ04 Group)	Locky	24	24
TA505 (SectorJ04 Group)	Clop	4	4
TA505 (SectorJ04 Group)	SDBbot	1	0
TA505, Indrik Spider	Dridex	38	50
Indrik Spider	FriedEx (BitPaymer, DoppelPaymer, IEncrypt)	5	2
Wizard Spider	Trickbot (Trickster, TheTrick, TrickLoader)	24	29
Pinchy Spider	Gandcrab (GrandCrab)	15	14
Pinchy Spider	REvil (Sodinokibi, Sodin)	14	0
Lunar Spider	Vawtrak (Catch, grabnew, NeverQuest)	14	15
Lunar Spider	IcedID (BokBot, IceID)	3	3
-	Gozi (CRM, Gozi CRM, Papras, Snifula, Ursnif)	2	2
-	GlobeImposter	4	4

Table 4.1: Table of malware samples plotted in Figure 4.13 with their assigned author sourced from the Malpedia [59] database. The same dataset is used later on in the dynamic analysis.

The **Trickbot** family is used to steal credentials and personal information, in addition to being delivered by other malware it can spreads through exploits over the network, or through phishing campaigns using infected documents [5]. Trickbot is believed to be related to the older malware family *Dyre* (Not in this analysis) [13].

The **TA505** group has multiple different malware families on their name. They have also been known to distribute Trickbot through spam campaigns [9]. *Andromut* and *Get2* are both downloaders for other malware. *SDBbot*, and *Flawedammy* and *Flawegrace* malware that try to establish remote access. *Locky* and *Clop* are both ransomeware malware. *Globeimposter* is also ransomware and has links to TA505 [8], however this has not been recorded on Malpedia.

Dridex is a malware focussed on acquiring credentials. There is a connection between the *Dridex* and the *Friedex* malware. Similarities have been found in the form of shared functions, similar PDB paths, and even similar build timestamps [3]. These leads may point to the malware originating from the same author(s). In addition, this malware has been dropped by Emotet in the past [12].

Friedex is ransomware malware, it encrypts the files on an infected system. Additionally, it can export files and can threaten to leak them in order to increase the pressure of paying for the decryption key. *Friedex* has been known to be dropped by *Emotet*.

The **Gandcrab** malware family was active for one and a half years starting in January 2018. It is a ransomeware malware variant, provided as a service to others to actually run the attack campaign [6]. *Gandcrab* has been to shown to have similarities with *Sodinokibi*.

Sodinokibi, has reportedly [4] been the successor of the *Gandcrab* ransomware from Pinchy Spider. The links were made based on the similar attack patterns, but also on code similarity [10]. The popularity of *Sodinokibi* also started to rise directly after the *Gandcrab* authors reported to stopped developing *Gandcrab* [5].

Vawtrak The aim of *Vawtrak* is to steal credentials, additionally, it is able to set up a remote connection through VNC and SOCKS servers. It is the predecessor of IceID malware [2]. Campaigns involving *Vawtrak* have stopped since 2017. *Vawtrak* reportedly contains parts of the *Gozi* malware [7].

IcedID's goal is to steal credentials by inspecting network traffic by setting up a proxy. The malware is suspected to be developed by the same group as who developed *Vawtrak*, which stopped operating right after *IcedID* was first seen [2].

Gozi's source code of *Gozi* was leaked in 2010, [7]. Just like *IcedID* it's goal was to steal credentials.

A summary of the connections are visualized in Figure 4.2.

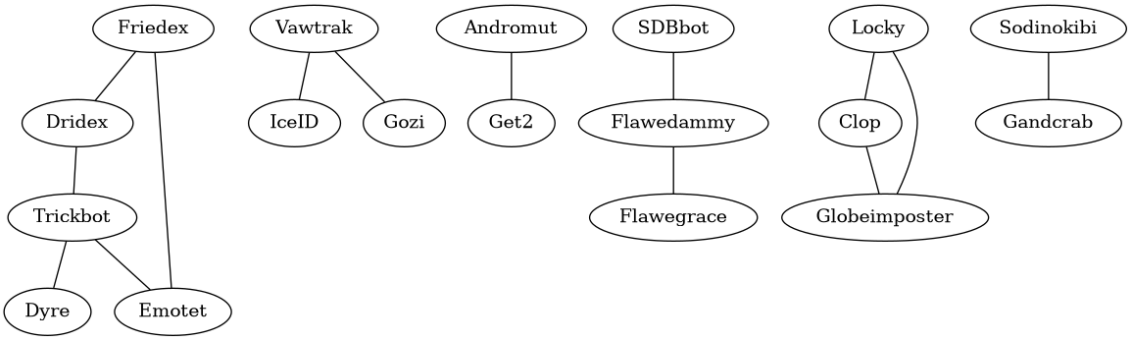


Figure 4.2: Summary of the expected connections among malware families based on individual research of Antivirus and cyber security companies.

4.2. Feature extraction

4.2.1. From source code to machine code

A programmer uses a higher level language to write his programs than the processor of a computer is able to interpret. Therefore, before a program can be executed by the central processor unit (CPU) it is required to translate the program to a language the CPU understands, also known as compiling the source code.

A compiled program can be decomposed into 5 layers:

1. Byte: The lowest layer are the bytes consisting of eight bits valued zero or one.
2. Instruction: Multiple bytes form a opcode with its operands, which together form an instruction
3. AST Block: A block in the AST could be body of a loop or the branches of an `if` statement.
4. Functions: A program is composed of a collection of functions.
5. Program: A collection of one or more functions form a program.

The layers can be viewed as a directed graph of graphs, where the functions are the nodes and the calling of a function creates an edge between the nodes. A program is a graph of graphs since, also called *call graph*, each function can also be viewed as a directed graph. These inner graphs are also called *Control Flow Graphs* (CFG). We will compare binaries on the function level since the tool we used for disassembling was not able to reliable extract enough information on block level.

4.2.2. Instruction n-grams

We will extract features on the instruction level, by disassembling a binary using Radare2[3]. Because the direct output of Radare2 contains some garbage, we will first sanitize the output.

Using the sanitized output we will create n-grams from the instructions. n-grams can be used as a method to create a probabilistic language model from sequences of symbols. We will use it to generate a distribution of the appearances of sequences of instructions in a binary. n-grams are generated by taking a sliding window of length n over the instructions and count the number of appearances of each sequence.

For example, we obtained the following raw disassembled output from Radare2:

```
lea rdx, [rax*4]
mov eax, dword [rdx + rax]
jmp 0x4781
```

This will first be sanitized such that memory locations are changed to a constant `MEM`, since they are likely to only appear once. The sanitized output then looks like:

```
lea rdx, [rax*4]
mov eax, dword [rdx + rax]
jmp MEM
```

[3] `https://rada.re/r/`

We recognize two variables regarding the type of n-gram on instruction level. (1) The number n of instruction in an n-gram and (2) the number of "words" in a gram which could lay between 1 and 3.

For example, this instruction sequence can be extracted to the following n-gram sequences:

- 1-gram with wordsize 2: `((lea rdx), ), ((mov eax), ), ((jmp MEM), )`
- 2-gram with wordsize 1: `((lea, ), (move, )), ((mov, ), (jmp, ))`

4.3. Classifcation

We first use classification of the author, because this was the research question of previous work instead of clustering. In addition, this allows us to validate our implementation against previous work.

Figure 4.3, illustrates the classification pipeline. Since the dataset for each author only consists of up till 12 samples, we use stratified k-fold to split the training and test set into k-sets, meaning that the number of training samples is $k-1$. This should reduce the effect a certain choice of training and testing samples has on the final precision. Stratified k-fold compared to regular random k-fold makes sure that each fold contains the same amount of samples for each class. This avoids the risk of a class not containing any training samples in a fold.

For the classification algorithm we use random forrest, since this showed the best results in previous work, in contrast to a classification using support vector machine (SVM). Regarding the hyperparameters for random forrest we saw very little effect when experimenting with our dataset, therefore we chose to keep them on the default given by the implementation we used (Scikit learn[4]).

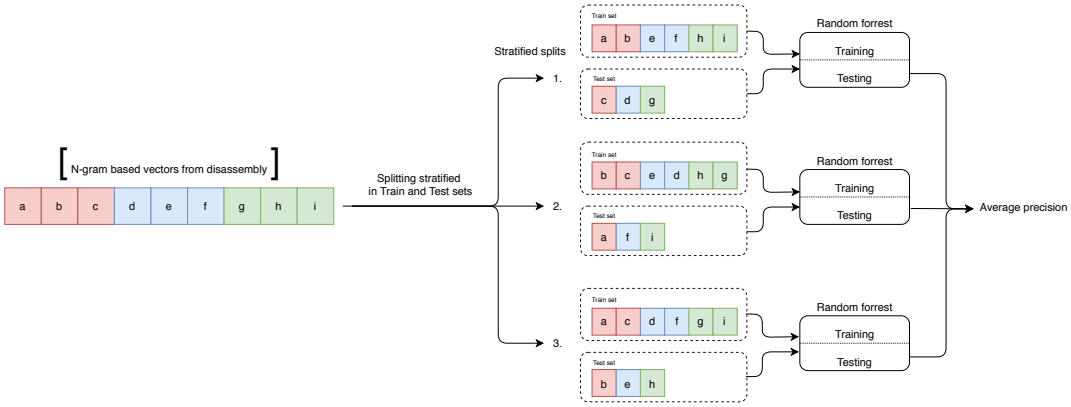


Figure 4.3: Flow of data during classification

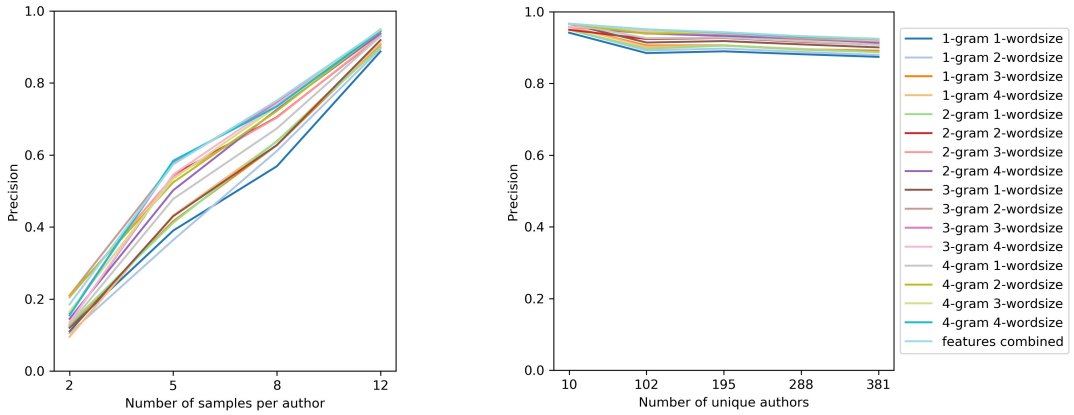
Random forrest determines for each sample the probability it being a certain class. The class with highest probability however is not always the correct class, as the sample of a different author may also be very similar. However, the correct class is possibly the second or third most likely class. Therefore, we also try a relaxed form of classification that will also evaluate a prediction as correct if the correct class lies within the r most likely predictions.

4.3.1. Results: classification - GCJ

The results from classification using Random Forrest on the GCJ dataset are shown in Figure 4.4. In Figure 4.4a the number of training samples per author are varied, there are $n-1$ training samples used and 1 test sample. From the figure becomes clear that a higher number of training samples clearly results in a higher average precision. From these results of Figure 4.4, we see that larger n-gram sizes overall provide higher precision. The *features combined* label is a model trained on the largest 8 n-gram types, (3-gram 1-wordsize untill 4-gram 4-wordsize). However, this combined feature does not perform much better than the largest 4-gram alone.

We only report precision, as each class (author) contains and identical number of samples. Therefore, the average precision and recall scores are identical.

[4] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`



(a) Varying the number of samples per author

(b) Varying the number of unique authors

Figure 4.4: Classifcation results GCJ dataset

In Figure 4.5 the combined n-gram feature is used with a relaxation parameter. $r = 4$ means that the prediction still counts as true positive if it lies within the 4 most likely predictions. We observe that an accuracy improvement of up to 1.4 times without relaxation can be achieved for the lower number of samples per author.

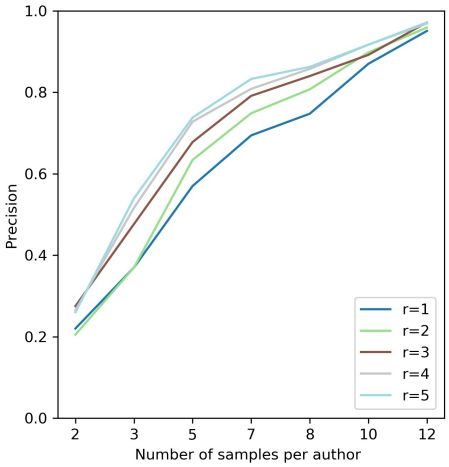


Figure 4.5: GCJ with 100 authors, r is the relaxation paramater

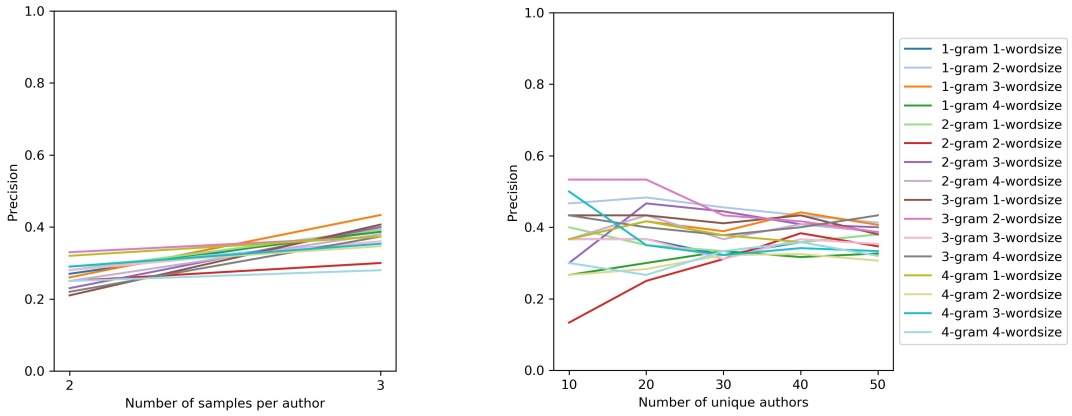
4.3.2. Results: classifcation - Malpedia

The same methodology as for the GCJ dataset is applied to the Malpedia dataset. We have however much less data than for the GCJ dataset. The number of samples is limited to 3 samples per author. Comparing the results in Figure 4.6 with the results from the GCJ dataset. We see a comparable precision for only a single and two training samples, the GCJ results however use 100 authors and the Malpedia set has only 50 authors, so the Malpedia precision score is slightly worse than the GCJ score.

The dataset used for Figure 4.6 does not distinguish samples of the same family for an author, but a different version. Therefore, the samples for a single author may be comparable in functionality and share a large amount of n-gram sequences. To see how much this affects the precision the dataset was filtered such that only a single sample of a malware family is present per author in the new dataset.

In Figure 4.7 the classification is done for a dataset that only contains a single sample per malware family. The precision scores are lower than the scores obtained on a set without the filter. This may show that code similarity, instead of code style has a high impact on classifying the correct author. The results are discussed

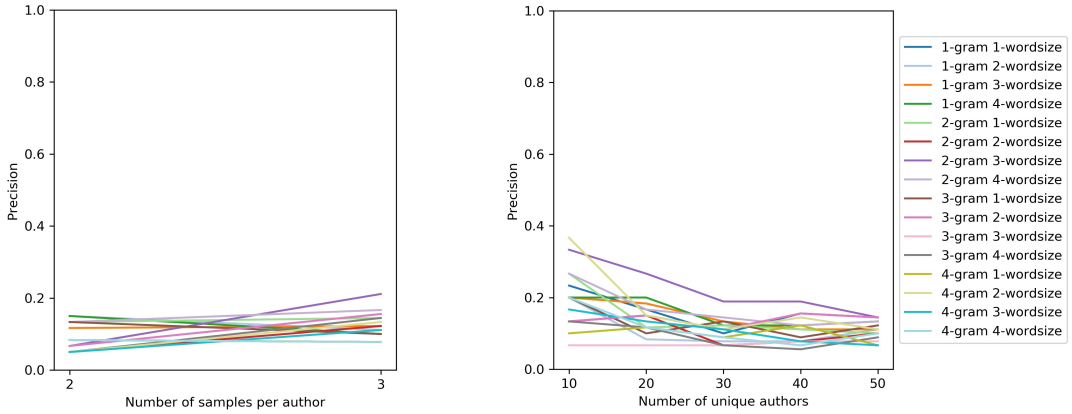
more elaborately in subsection 4.3.3.



(a) Total of 50 authors, varying the number of samples per author

(b) Varying the number of unique authors

Figure 4.6: Classifcation results Malpedia dataset



(a) 50 unique authors, Varying the number of samples per author

(b) Varying the number of unique authors

Figure 4.7: Classifcation results Malpedia dataset, with only a single sample per family

4.3.3. Discussion: classification

Comparing the results from previous work on the classification of authors on the GCJ dataset with our own results we can conclude that we can successfully reproduce the work from the literature as was described in section 2.1. Even though we only used n-grams unlike previous research which used graph features or disassembled code, the precision remains largely the same as in works that did use those extra features. This confirms the hypothesis made from the literature that the n-grams add significantly to the information to describe the separate samples.

As a side note, we found a possible issue with the GCJ dataset, that has not been identified by previous literature, that could influence the conclusion of the results, this is further explained in the appendix.

However, using the same classification methods on the Malpedia dataset results lesser scores in terms of precision. The results of the set for which the samples are not yet filtered, to remove samples from the same family are up to about 40% for the precision score. Which is not too bad, however when the samples are filtered, such that only a single sample per malware family remains the average precision drops below 20%.

There are two possible reasons for the worse results compared to the Google Code Jam results, the first is that the author labels of the Malpedia dataset are simply wrong. Someone labeled these samples and possibly mistakes could have been made here since the reasoning for why certain samples may belong to the same author are manual work and reasons for assigning a certain malware to a certain author can be diverse and

do not necessarily have to be visible within the malware's code. Examples of these identifiers could be the way malware is distributed, the countries the malware is most seen in, or even from the communication in the case of crypto locker malware. A possible second explanation is that the code (style) is actually similar among programs from different authors, which could mean the different author labels are actually the same author, or that the authors share code among each other. In other words, there is not enough difference between the programs for the classifier to learn. A third option explaining the results between the GCJ, and the between the two Malpedia sets is that what is learned by the random forest classification is not similarity in the sense of code style but in the sense of code similarity caused by the functional similarity of the code. We think this third option is the most likely cause, also given by the results from Appendix A. The GCJ samples are similar because they are small programs quickly written by an author, an author may favor specific functions written by him before to do common calculations, and copy these (boilerplate) functions among all their submissions. The higher precision from the Malpedia dataset that is not filtered to contain only a single sample per family per author, can be explained for the same reason. The same family could have only incremental improvements, re-using large parts of the code from the previous version of the software. The code largely remains the same.

4.4. Clustering

We use two methods for the clustering, the first is to cluster full binaries, the formed clusters are then evaluated by how close the clustered samples from the labeled binaries lay to each other. Since this method does not yield practically useful results we also propose a second method that produces visibly more useful results. The second method is to cluster on a finer granularity than on full binary level, and cluster based on the similarity of functions.

For both methods, just as for supervised classification we use n-grammed instructions as features to represent a binary sample. For the n-gram length we used bi-grams since from the classifications experiments we could see that the length of the n-gram had a relatively small effect on the accuracy. Smaller n-grams are however preferred because they result in a smaller feature vector than larger n-grams, because of the smaller vocabulary. After creating n-gram feature vector from p unique instructions, resulting in feature vectors of length p^n. Using these vectors we for all samples S we can compute the cosine distance among all samples, this results in a distance matrix D of size $S \times S$.

Clustering to determine the author of a binary

Since we can only evaluate whether binary code of the same author appears similar, metrics such as the mutual information score are useless. Since they also take into account the "mismatches", which we cannot evaluate on whether they are actual mismatches. Instead, we will compare how many of the samples of the same author lie within a certain threshold of each other.

For all the samples S, we will extract the samples S_e that lie within a certain threshold t of the currently evaluated sample. Because in the distance matrix the diagonal shows the similarity of the sample itself, we will remove this element from the evaluation. The performance of the clustering is then calculated by taking the percentage of samples that lie within the threshold t, by calculating $\frac{|S_e|}{S-1}$

Clustering similar functions

We also propose a second clustering strategy as depicted in Figure 4.9 to be used for clustering on a finer granularity to find the similarity among authors. Since the distribution of instructions over an entire program might be too dependent on the functionality the program implements.

A cluster is then first defined as a set of similar parts of code, from a larger program, among programs. We can however not simply put all vectorized blocks of code and feed these to a clustering algorithm such as K-means, or HDBScan see Figure 4.8. Large parts will not cluster at all, and the clusters we are looking for are relatively small given the total amount of all points.

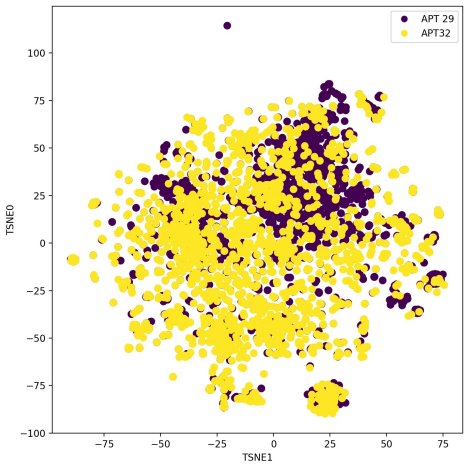


Figure 4.8: Sample illustrating that we cannot simple use a common clustering algorithm on the data, each data-point represents a block of code, in total two authors with each 1 programs.

Instead, it would probably be better to do a comparison of distributions on a finer level, the function level, such that it would still be possible to identify when a programmer copies a few functions between programs. We will therefore first split each binary in b blocks N_b, then we first calculate the cosine similarity pairwise between the blocks of two binaries, before comparing all binaries to each other. We then sort the blocks based on their similarity score, and call the points with the highest similarity a match. Then for each of these

matched points, we will look for functions that lie extremely close to each other, therefore, these points are only a match if the similarity lies within a certain threshold. By counting the matching functions a distance matrix D_s of size $N_{ba} \times N_{bb}$ can be computed. More formally the algorithm for comparing two malware samples with each other is given in algorithm 2.

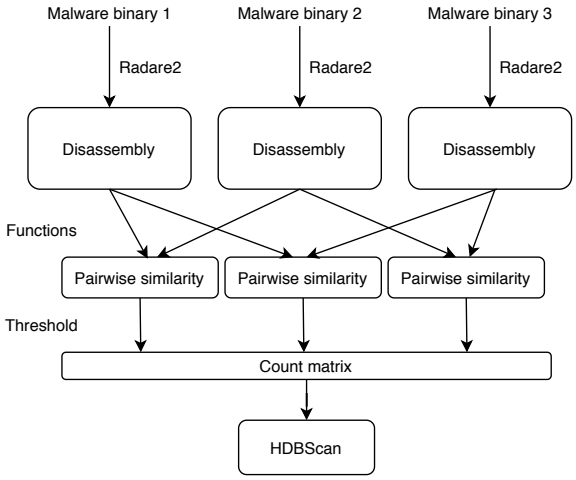


Figure 4.9

The similarity between the two binaries will then be based on the number of matched functions. From this, we can then create a new distance matrix with size $S \times S$. Each element of this matrix has the count of similar functions normalized by the total number of functions of a binary. The main limitation of these distance matrices is the amount of memory it consumes for large numbers of samples.

Algorithm 2: Algorithm to find similarity between two binary samples, on fine granularity

Input : 2 vectors B_a and B_b of n-gram vectors with shape $p^n \times b$
Output: Matching functions
$D_s \leftarrow$ cosine-similarity(B_a, B_b)
$D'_s \leftarrow \max(D_s)$
$N_{similar} \leftarrow 0$
foreach *element in* D'_s **do**
  if *element* >= *threshold* **then**
    $N_{similar} \leftarrow N_{similar} + 1$
  end
end
return $N_{similar}$

4.4.1. Results: clustering - GCJ

To get an idea of the full binary similarity a sample heatmap is shown in Figure 4.10 from the n-gram vectors. This show clear square block of high similarity among samples originating from the same author. Based on this illustration alone we can be quite confident whether we are able to identify clusters of samples from the same author.

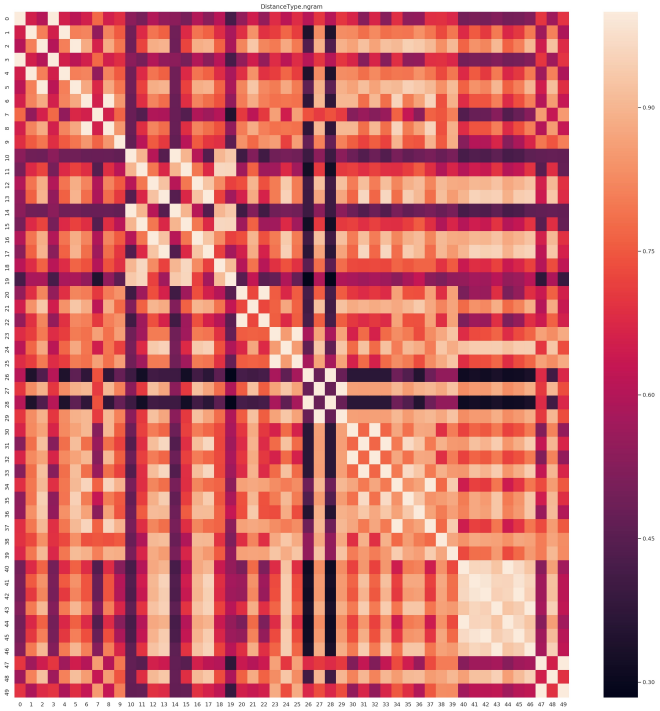


Figure 4.10: Heatmap displaying 5 random authors from the GCJ dataset and 10 of their samples.

This hypothesis is tested using the clustering evaluation described previously, the results are displayed in Figure 4.11. As expected the higher the threshold the less samples are classified as from the same author. Also, we see that increasing the number of authors reduces the average percentage only slightly, by about 15 percent.

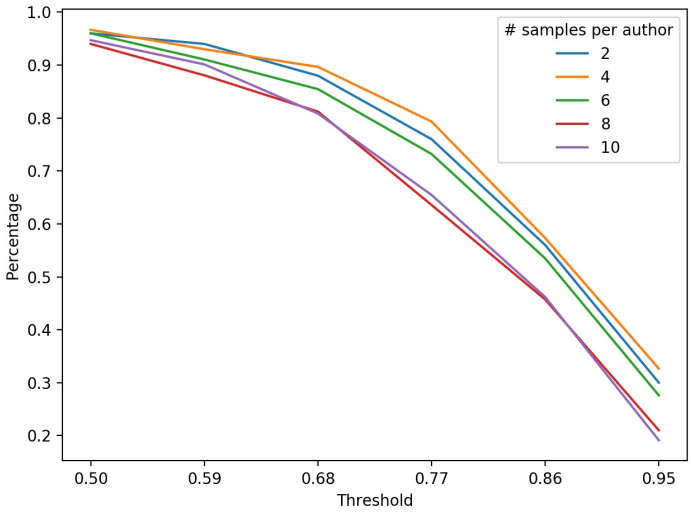


Figure 4.11: Percentage of samples of the same author within a subset of binary samples that is the result of the threshold value on the x-axis. Number of unique authors used is 500.

4.4.2. Results: clustering - Malpedia

We would like to verify whether the developed method is able to detect similarities between Malwares of different families.

Full binary similarity

The first method to find the similarities among malware samples is to calculate similarity over the full binaries. The results of this are shown in Figure 4.12. Although the malware samples within a family show clear similarity the possible similarities among the sample disappear in the noise or get cancelled out because of other parts in the binary.

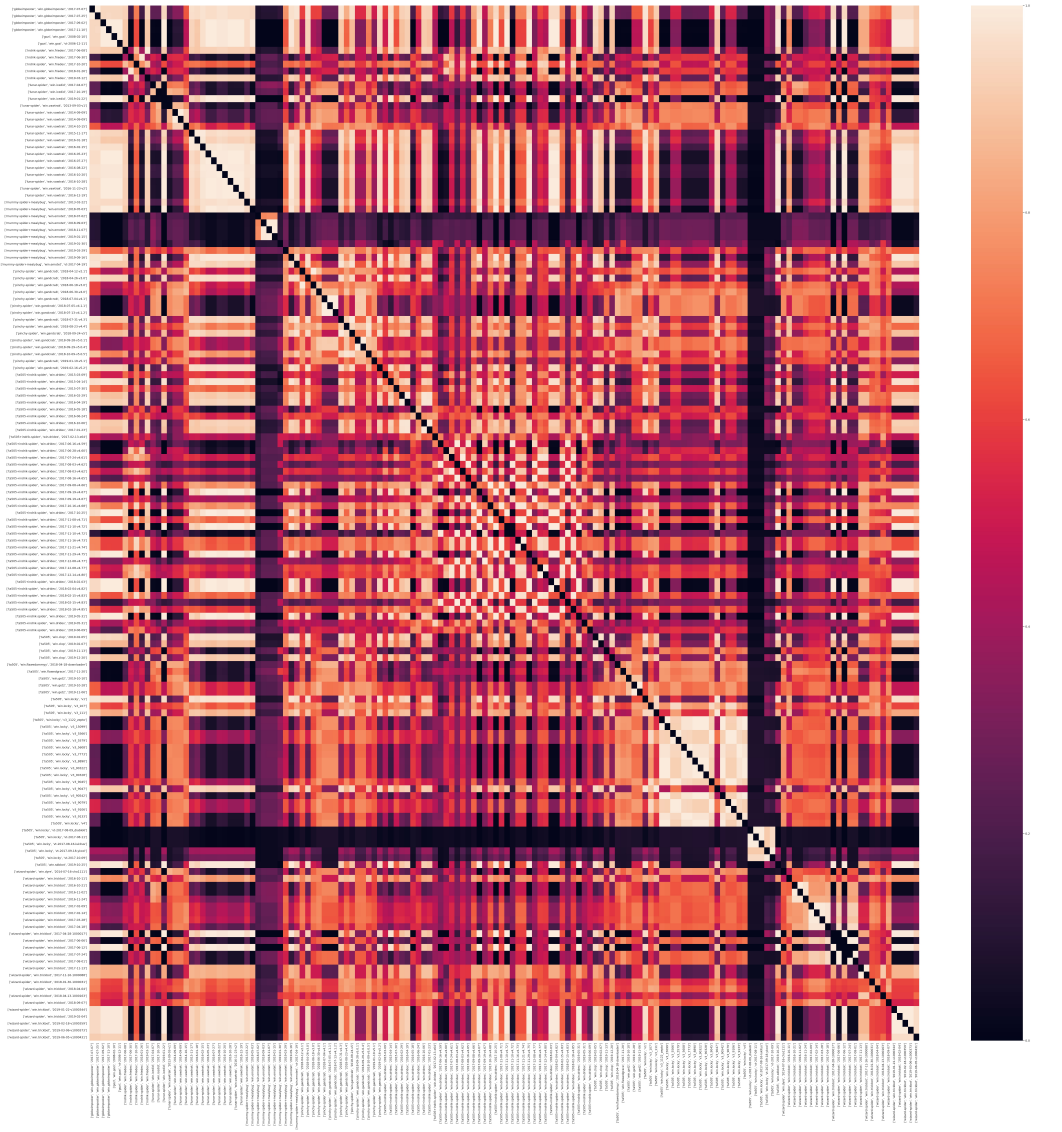


Figure 4.12: Cosine similarity between n-gram vectorized binaries. Each element is the similarity score between two samples, 1 being the highest similarity. The diagonal is not calculated.

Function similarity

Since directly calculating the cosine similarity of the full binary does not lead to desirable results, we will now proceed to show the results from the method to find the finer granularities. The resulting similarity matrix from this second method is shown in Figure 4.13. This heatmap shows the similarity among binaries on a function level. We used a threshold of 0.9 as the minimum cosine similarity between two n-gram functions vectors to be classified as similar.

The heatmap of Figure 4.13 uses the finer granularity clustering as explained in section 4.4 to plot a matrix M_s. We can use the heatmap directly to visually inspect the clustering results, however to automate the clustering we can also use a clustering algorithm on the computed distance matrix. In order to inspect the clustering results, we first need a way to visualize the clusters. We can do this by mapping the similarity matrix to 2d space. In Figure 4.14a we first transformed the similarity matrix of Figure 4.13 to a distance matrix by taking $1 - M_s$, then the distance matrix is transformed to 2d space using t-SNE [61]. We also experiment with multi-dimensional scaling, but t-SNE better represented the clusters as visualized in the distance matrix for this dataset.

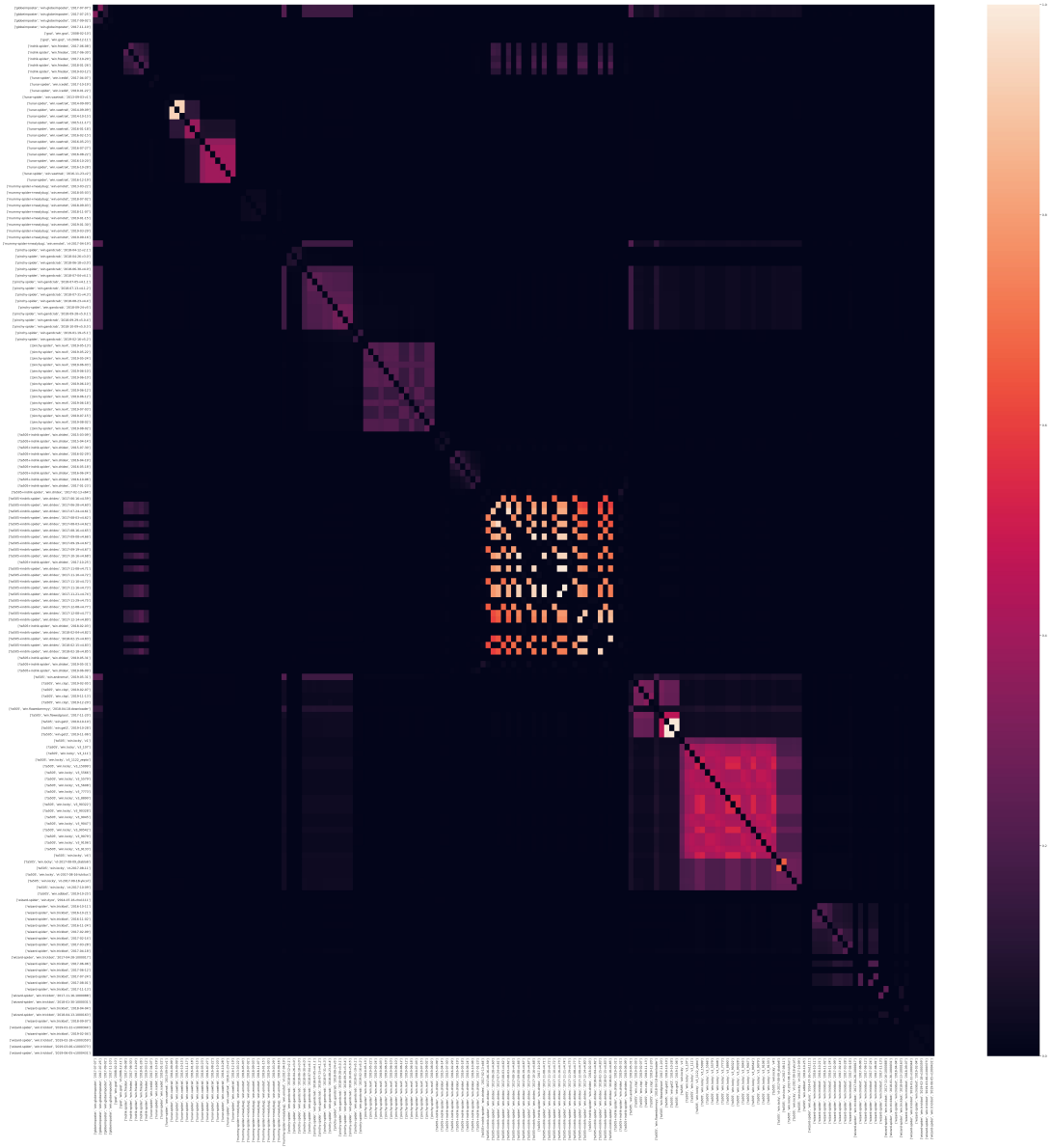
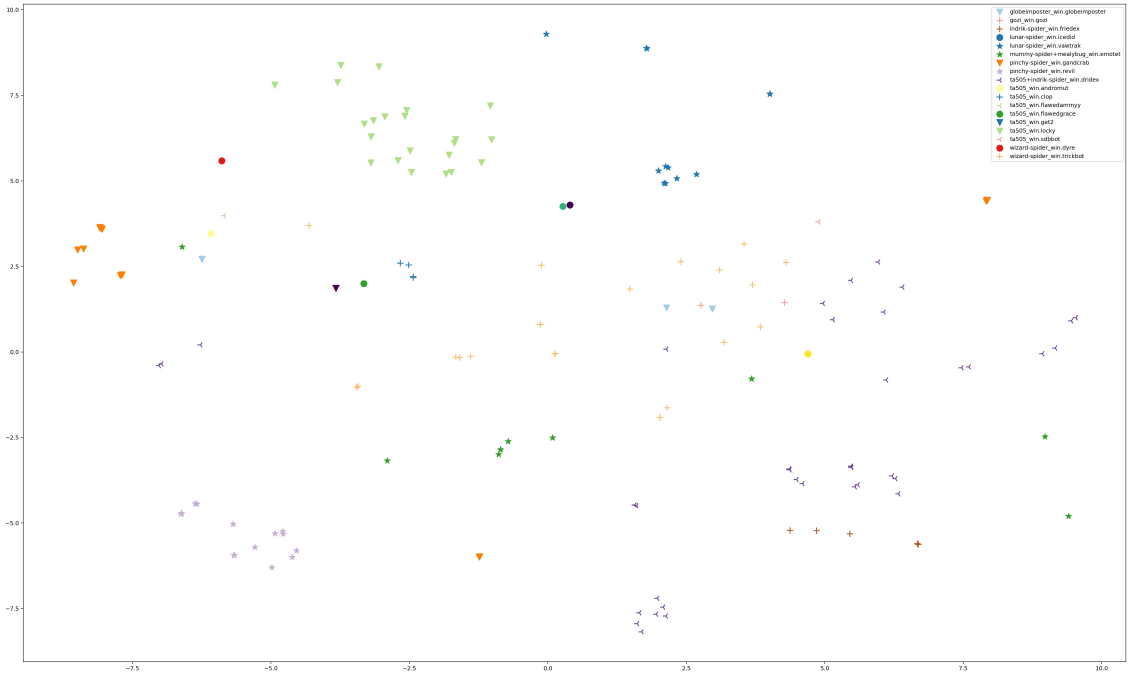


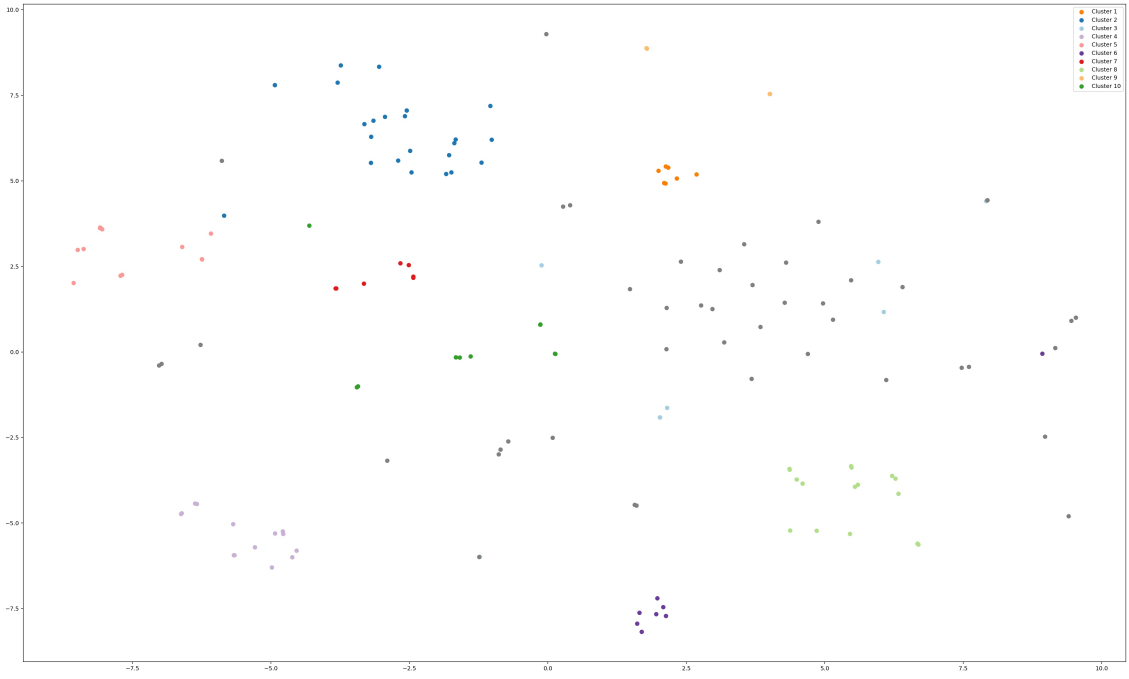
Figure 4.13: Each element is the number of matching functions normalized by the total number of functions of a binary.

Now to do the actual clustering in Figure 4.14b we used the HDBScan algorithm to find clusters with a minimum size of 5 elements in the original, non mapped data. The found clusters however mostly cluster just the samples of the family together, probably because the density of these samples is greater than the density of the samples from different families. Therefore, manual inspection of the similarity matrix is probably better here. In total, we found 10 clusters, while 27 of the 134 samples did not belong to any cluster, these clusters are color coded in the figure.

The first thing that can be noticed is that malware from the same family appears quite similar in terms of



(a) Projection of the distance matrix of Figure 4.13 using t-SNE, samples are labeled by their name.



(b) Projection of the distance matrix of Figure 4.13 using t-SNE, colored clusters are formed using HDBSCAN.

Figure 4.14

the number of shared functions. In a few cases the malware from the same family form sub-clusters within a family, possibly because of incremental newer versions. Overall however the families are clearly visible as they show up as square blocks in the heatmap. *Gozi*, *IcedID*, and *Trickbot* are however an exception on this.

Now what we are actually interested in is whether it is possible to find similarities among malware samples from different families or different authors.

From left to right on the x-axis of Figure 4.13, we firstly see that *Globeimposter* has many similar functions with *Gandcrab*. Besides, both of these malware families are ransomware malware, no links between these are currently mentioned in any malware reports.

Secondly *Friedex* matches functions with *Dridex*, both of these malware are already labeled as being from the same actor. They are linked as Friedex being an evolution of the Dridex malware[5]. It is interesting to still see similarities given the difference in goals between the two malware. The goal of Dridex was to just extract information from the infected host, while Friedex is ransomware malware.

Thirdly we see that the 2018 version of *Gandcrab* a ransomware malware has similarities with many of actor *TA505* their samples, while *Clop* and *Locky* are also ransomware, the other families assigned to *TA505* are not.

Fourthly *Clop* shows similarities with *Globeimposter*, *Gandcrab*, all of these malware are of the type ransomeware. However, they currently have a different author listed on Malpedia.

The diagram of Figure 4.15 shows a summary of the found similarities. The found similarities seem to correlate with the malware type in some cases, although we also found some similarities among samples that have a different purpose, and are currently tagged with different authors. Several Spider related groups appear to show similarities with TA505, with the *Globeimposter* malware which is currently not attributed could be attributed to the *Pinchy spider* group based on binary code similarity. We also see a possible wrongly attributed file, because one *Emotet* sample appears to be *Gandcrab*.

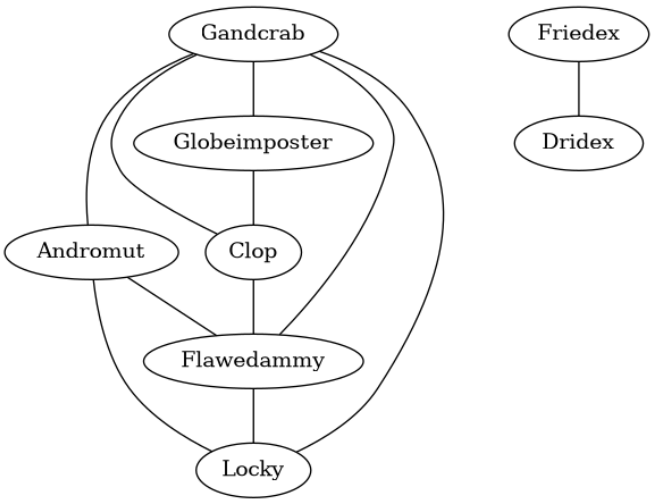


Figure 4.15: Graph summarizing the found connections from the analysis of the Malpedia samples

Finally, given the labels we have we use three clustering evaluation metrics. We see that the precision is higher than recall, the low recall means that some samples are individually scattered over the clusters. We see this back in the figures, the precision is relatively high since the clusters consist mostly of samples from the same family.

Purity index	Rand index	Precision	Recall	F1
0.93	0.78	0.84	0.33	0.41

Table 4.2

To support the hypothesis that the malware needs to be unpacked for this analysis to succeed. The same analysis was executed on the same malware samples, however the difference is that now the non-memory

[5]https://www.us-cert.gov/ncas/alerts/aa19-339a

dumped version of the malware is used. The results of Figure 4.16 are self-explanatory, no clusters are formed by the malware samples and therefore no useful conclusions can be drawn.

Figure 4.16: Projection of the distance matrix of non-memory dumped malware samples, dimensions reduced to 2d using t-SNE, colored clusters are formed using HDBSCAN. Even with a very low perplexity t-SNE forms a single large cluster.

4.4.3. Discussion: clustering - GCJ

Looking at the clustering results we first see that the GCJ clustering for finding similar authors We see that the number of samples per author does not influence the percentage of samples from the same author that lie close to each other. The difference between 10 and 2 samples are a maximal of 40% as it increases slightly as the number of samples rises. Naturally, as the required threshold to be marked a similar increased the percentage of clustered samples decreases.

4.4.4. Discussion: Clustering - Malpedia

In this subsection, we will discuss the results of the clustering methods on the Malpedia dataset. We first compare the function level clustering method to the full binary clustering method, then we will discuss three case studies with the goal of providing a better insight into how useful function level clustering. Finally, we discuss possible mistakes the method makes and things to keep in mind when interpreting the clustering results.

Comparing function similarity with full binary similarity

Regarding the clustering strategies for finding similarities among authors, we conclude that simply comparing vectors extracted from full samples does not result in desirable results. For instance, it provides no clear conclusion as some samples such as the *Vawtrak* malware appears to lie close to more than 50% of the other samples. We argue that the function level similarity analysis provides a better view since when two samples lie close to each other the families to which they belong also lie close to each other, which is not true for the binary level comparison.

Case studies

In order to further assess how well the function level clustering works we will inspect some of the found similarities deeper. We will not try to explain what the code of the selected binaries do, as this is out of the scope of this Thesis. Instead, we will analyze the underlying data to explain the results. We will look at three cases, first we will look at the found similarities between *Gandcrab* and *Globeimposter*, since this similarity is unexpected from the given literature. Secondly, we will look at the similarity between Dridex and Friedex, this

similarity was expected and we will look at how our found similarity aligns with the way the similarity was determined by antivirus companies. Thirdly we will look at what causes the found similarity in function between the most the TA505 group's malware.

These three cases will be discusses on possibly four points, depending on what shows interesting results:

1. Possible reasons for the lack of similarity among the same family.
2. On statistics of the functions, these include:
    - Average function length, as in the amount of instructions per function.
    - Percentage of functions matched[6] between two binaries.
    - How connected are the found similar functions to the rest of the call graph (mon-matched functions) in the global call graph.
3. The second point we will look is how the matched functions connect with each other to put the functions into a greater program perspective. A connection between two function is formed by one function calling the other functions. When not only a set of functions matches between two binaries but also the connections between these functions matches, it provides additional evidence for similarity of two functions. See Figure 4.17 for an example.
    - Connections between matched functions.
    - Connections between the matched functions and the full call graph. This number gives an indication of how connected the sub-graph of matched functions is to the rest of the call graph.
4. The kind of visible n-gram patterns and the difference between the matched and non-matched set of functions.

Before moving on with the three cases we will first provide the numbers for the full dataset, such that we can put the numbers found for the separate cases into context. In addition, this provides us with insights into the general characteristics of the method on this dataset. From Table 4.3 we see that the matched functions are significantly longer, as they contain more instructions, than the functions for which no match was found. We see that on average 35% of the functions could be matched to a function from another binary for binaries that had matches. Interestingly we see that matched functions are on average longer than functions that are not matched. Further, we see that we capture on average 22% of all edges in the full call graph. With captured edges, we refer here to the edges that are present in both binaries, the other edges are on the next row which are on average 23 edges. The matched functions together form a sub-graph in the two full call graphs that are compared. We see that on average 322 edges connect these sub-graphs to the rest of the call graph.

	μ	σ
Matched function length μ	194.383675	598.115463
Matched function length σ	228.637651	562.246477
Non-matched function length μ	62.981182	27.699254
Non-matched function length σ	97.447121	75.005913
# Total functions per binary	496.689010	267.723818
# Matched functions per binary	173.095411	192.961112
# Edges full function graph	1513.477959	1314.559087
# Edges between matched functions	339.977355	685.735263
# Non-matching edges between matched functions	23.164251	61.229428
# Connecting edges to the rest of graph, incoming to matched graph	215.631080	337.683090
# Connecting edges to the rest of graph, outgoing to matched graph	118.289956	189.719698
# Connecting nodes to the rest of graph, incoming to matched graph	57.000632	61.715662
# Connecting nodes to the rest of graph, outgoing to matched graph	64.411560	80.356982

Table 4.3: Statistics of full dataset extracted from call graph properties.

[6] A match between two functions is defined as the cosine similarity of two functions being higher than the threshold.

In Table 4.4 we explored the difference in n-grams patterns of the full dataset for the matched and non-matched functions. If all functions are equally likely to match depending on the type of n-gram the difference between the two sets should be close to zero. However, we see that the non-matched functions are more likely to include arithmetic operations such as *add mul* or logical operators such as *xor* and *and*, while matched functions have a greater tendency to include stack related instructions such as *call* and *push*. In addition matched functions also appear to include the *nop* instruction more often, which is an instruction which does nothing, but is used for timing purposes.

The *difference* from Table 4.4 represents the difference in the number of functions between the matched functions and the non-matched functions. It is calculated comparing all binaries (excluding binaries from the same family) to each other, which results in two sets of n-grams, of for the matched functions and one for the not-matched function. For each set the number of functions that contain each n-gram is counted for each n-gram. The counts of functions are normalized for each comparison by dividing it by the total number of matched or non-matched samples. Then the normalized counts are subtracted from each other by subtracting the normalized matched counts from the normalized non-matched counts.

Bi-gram	Difference μ	Difference σ	Count
('pop', 'ret')	-4.739948e-02	0.262723	6272
('add', 'pop')	-4.596471e-02	0.075678	6258
('add', 'lodsd')	-3.518091e-02	0.066502	130
('MEM', 'add')	-3.099159e-02	0.131117	2486
('call', 'pop')	-3.004333e-02	0.180293	6260
('xor', 'xor')	-2.993098e-02	0.052340	6132
('shld', 'add')	-2.400837e-02	0.009714	378
('add', 'ja')	-2.324371e-02	0.101472	60
('jbe', 'lea')	-1.522993e-02	0.020822	6061
('add', 'MEM')	-1.463469e-02	0.091122	4669
...	...	...	...
('nop', 'pop')	3.846148e-01	0.100643	465
('pop', 'nop')	3.957887e-01	0.098913	462
('nop', 'lea')	4.019235e-01	0.349538	776
('lea', 'nop')	4.088955e-01	0.347534	769
('nop', 'push')	4.303842e-01	0.296709	648
('nop', 'call')	4.330395e-01	0.182022	515
('call', 'nop')	4.563656e-01	0.136124	475
('push', 'nop')	4.708584e-01	0.302432	619
('nop', 'jmp')	6.732655e-01	0.172260	465
('jmp', 'nop')	6.760150e-01	0.166540	462

Table 4.4: Bi-grams are sorted on the mean difference, the top 10 rows shows the largest difference in mean where the non-matched samples are more than the matched samples, while the bottom 10 rows show the larges difference in mean where the matched samples are more than the non-matched samples (395 rows hidden). The table is filtered to only contain rows with at least a function count of 100 or higher.

Globeimposter and *Gandcrab* - Table 4.5 - The similarity between these two families is quite unexpected since no sources could be found that mention a possible connection between these two malware families. We will compare *Globeimposter* version *2017-07-07* and *Gandcrab* version *2018-06-30-v4*. The versions of *Gandcrab* before version 4.0, do not show similarities, it is not clear what the cause of this is other than that the versions before 4.0 are completely different from the newer versions. This is a likely possibility since the number of functions went from 80 before version 4 up to more than 230 with version 4, in addition the average function length dropped from 100 to 60. *Gandcrab* and *Globeimposter* together share 180 functions, for *Globeimposter* this is 50% of all its functions, while for *Gandcrab* it is even 78% of the functions. The sub-graphs of matched functions are quite intertwined with other non-matched functions, the matched functions do not appear to be clearly seperated modules based on the assumption all possible similar functions are found. Firstly because there are more outgoing edges than incoming edges (106 incoming vs 147 outgoing for *Globeimposter* and 28 incoming vs 151 for *Gandcrab*). Secondly, because between 40 and 50 percent of the nodes from the matched graphs are connected with the the rest of the unmatched functions.

	win.globeimposter	win.gandcrab
Matched function length μ	64.0778	66.5034
Matched function length σ	132.848	119.971
Non-matched function length μ	63.3594	66.0581
Non-matched function length σ	108.129	92.8941
# Total functions per binary	372	231
# Matched functions per binary	180	180
# Edges full function graph	1011	729
# Edges between matched functions	318	278
# Non-matching edges between matched functions	37	15
# Connecting edges to the rest of graph, incoming to matched graph	106	28
# Connecting edges to the rest of graph, outgoing to matched graph	147	151
# Connecting nodes to the rest of graph, incoming to matched graph	47	18
# Connecting nodes to the rest of graph, outgoing to matched graph	75	69

Table 4.5: Function statistics comparing *Globeimposter* to *Gandcrab* malware

In Figure 4.17 the global call graph is displayed for the similar for only the similar/matched functions. A call graph shows the connections among functions formed by one function calling another function. From this graph we see that 88 percent of all edges match between the two binaries, which gives a good indication for how well the cosine similarity measure performs for matching similar functions.

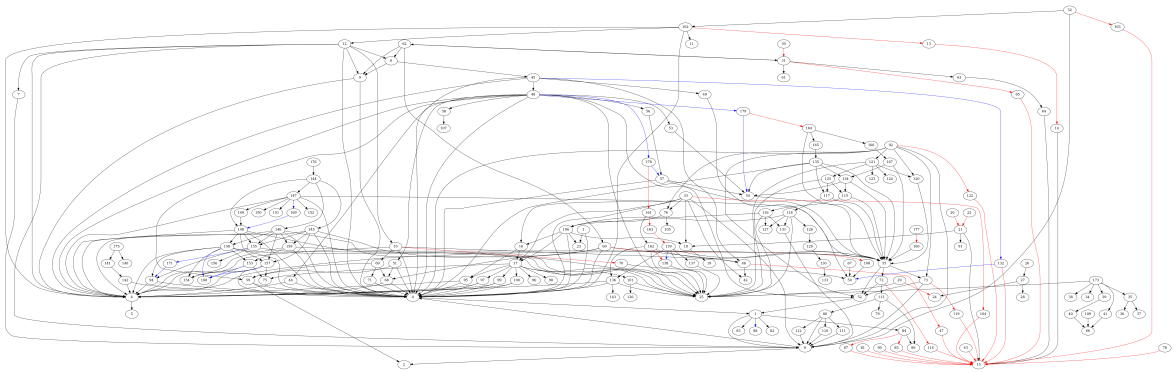


Figure 4.17: Global function gall graph of matched functions between *Globeimposter* and *Gandcrab*, the red edges show the edges that only appear in the *Globeimposter* malware and the blue edges show the edges only appearing in the *Gandcrab* malware. Black edges show connections that appear in both *Globeimposter* and *Gandcrab* malware.

We will discuss some of the nodes with a lot of non-matching edges in order to get a better understand of what kind of functions cause possible bad matches. Starting with the *node/function 15* 15 from the graph, which looks interesting as it has a lot of unmatched incoming edges that are present in *Gandcrab*, but not in *Globeimposter*. Function 15 turns out to be a small function, all it does is call a different function at a certain address. Function 29 is interesting as it both has an outgoing matching edge as it has an outgoing non-matching edge. Following the non-matching it leads to function 24, this function appears to be correctly matched given the connection to a sub-cluster of three other nodes which all are matched and have matched edges. Looking at the call instruction in function 29 that leads to 24, we see that for *Gandcrab* it is an unmatched function, that is therefore not in the displayed graph. By manual inspecting the *Gandcrab* binary we learn that this function is exactly the same as *Globeimposter* function 24. Apparently, almost the exact same function (Figure 4.18), possibly because of its simplicity, appears multiple times in the binary.

Dridex and *Friedex* - Only half of the Dridex samples show similarities with Friedex, the versions of *Dridex* appear to alternate between between two version, first being similar for one version and then the next version not being similar, after which the next version is similar again. So based on the function similarity it appears there were two categories of *Dridex*, that sometimes appear individually in the database, but also occasionaly appear together under the same version in the database. These two versions of *Dridex* are not similar to each other, as only 9 functions from the total of respectively 830 and 950 functions show high similarity with each other. The binaries within the two categories however do show similarities among each other.

```
  ; CALL XREF from fcn.00402ed1 @ 0x402fca
  ; CALL XREF from fcn.00407177 @ 0x40718f
22: fcn.00406718 (int32_t arg_8h);
; arg int32_t arg_8h @ ebp+0x8
0x00406718      push ebp
0x00406719      mov ebp, esp
0x0040671b      push 0
0x0040671d      call dword [0x414150]                    ; "Y\xc3V\x8b5\x80\x15B"
0x00406723      push dword [arg_8h]
0x00406726      call dword [0x41414c]
0x0040672c      pop ebp
0x0040672d      ret
```

Figure 4.18: Function *Globeimposter* appearing multiple times in *Gandcrab*, causing mismatches.

For the deeper comparison to *Friedex* we will use the *Dridex* version from *2017-12-14 v4.80*. All of the *Friedex* samples in the dataset match at least some functions of *Dridex*, the number of matches increases with every new versions of *Friedex* the sample with the most matches is the sample from 2018, however the 2019 version shows less similarity with *Dridex* again. Taking a closer look at the difference between the different *Friedex* versions we see clear indications of incremental versions, see Appendix B for the full comparison table. The number of functions contained in *Friedex* increases significantly between 2017 and 2018, going from 226 to 375 keeping the relative amount of matched functions to *Dridex* at about 60 percent, while the 2019 version drops to a total of 336 functions. The 2018 version of Friedex appears to add more functionality from Dridex than the 2017 version, while the 2019 versions also adds a couple of more functions over the 2018 version, but also removes or changes quite a few functions that have been in *Friedex* since 2017. These changes cause the drop in similarity seen between the 2018 and 2019 versions. Even though the amount of matched functions grows for the 2018 malware, the found similarity appears to be of better quality, since the amount of unmatched edges in the *Dridex* graph drops from 32% to only 21% in the 2018 call graph.

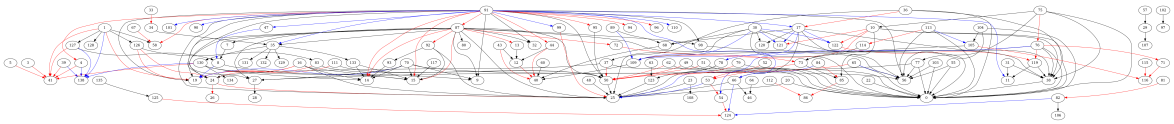


Figure 4.19: Merged call graphs for matched functions of *Dridex* comparison against *Friedex - 2017*

As can be seen in Figure 4.19, there are a lot more non-matching edges (30% of subgraph) in this graph compared to the comparison of *Gandcrab* and *Globeimposter* from previous case. The matched functions are on average (avg. 49, sd. 85) short compared to the average numbers of the full dataset. The amount of matched functions relative to the total amount of functions for Friedex is 62%, which is more than the average dataset. From which we could infer that although smaller function length result in about as many matches as for longer function lengths, although the quality of the matches is less good.

TA505 - We will be comparing a single sample from each of *TA505*'s malware families. We are interested to know whether all of *TA505* share the same set of functions, or whether they share different set for every pairwise comparison. Because we do only pairwise comparison we will take a single sample as baseline, for which we compare the other functions. If we find a single set of samples using this baseline sample the hypothesis that a single set of functions is shared among all families can be made likely. The results of the analysis can be found in Appendix C, we used Locky from *TA505* as the baseline to which other samples from *TA505* (*Andromut*. *Clop*, *Flawedgrace* and *Flawedammy*) are compared to. From this comparison we see that out of the total of 112 matched functions counting for all binaries 30 functions from Locky are found to be similar among all 4 comparisons. We see however a more clear similarity pattern between the pairs, *Andromut-Flawedammy*, and *Clop-Flawedammy*.

Although not as confirmed as the other malwares being created by *TA505*, other malware analysis indicated that the *Globeimposter* malware is also developed by *TA505* [8]. We already confirmed this similarity during the clustering analysis. Now we see that also the same functions match, one of the confirmed *TA505*'s malware, namely *Flawedammy*.

Possible mistakes made by the function level method

We will discuss possible mistakes the function level clustering method makes. The first problem has to do with the function size difference of matched functions vs non-matched functions, in general larger functions appear to be more easily matched than simple functions. Also, the quality of the matches of the shorter function appears to be worse based on the call graph data, as we see more edge mismatches with shorter functions. A possible solution for this would be to add a weight to each found similar function, that increases as the size of the found similar function grows.

A second issue is that currently, during a pairwise binary comparison, a function from one binary only matches to a single other function from the other binary we are comparing to. If the second binary contains multiple functions that are very similar there is high likelihood of imprecise matches. This issue can also be made less serious by using graph similarity, or by allowing a function to match to multiple functions. A possible reason for the occurance of multiple of these simple similar functions in a single binary instance if a function provides simple functionality such as only writing a string value to the register.

4.4.5. Limitations

Cosine similarity is not a computationally expensive operation, as it is a vector operation. Doing pairwise similarity among all samples of a dataset still does not have to be as expensive as it can be executed using a single matrix. However, we calculate the pairwise cosine similarity among all functions of two malware samples, instead of just computing a single pairwise similarity among all samples. This raises the complexity as we are now calculating a similarity matrix between each pair of malware samples. Therefore, the algorithm may not be fast enough for very large datasets.

4.5. Summary

We showed that classification using a labeled dataset is not the best option for malware author attribution. Instead of classification clustering is the more logical choice for malware. We showed that by engineering the distance metric in a specific way, clusters of different but similar binaries can be made, such that malware tagged as a different family can be clustered. Even though n-grams and cosine similarity are relatively simple, they appear to work decent enough to cluster malware together even when they are tagged as different families as was assessed visually. Searching for similarities in bigger datasets, provides information that can save a malware analyst time as similar functions do not have to be manually inspected again. In addition, it provides a method to explore the bigger picture in the malware landscape from similarities among samples that may first sight do not appear similar.

Lastly, we showed a deeper analysis to getter a better understanding of how the methods works, on the limited Malpedia dataset of memory dumped malware samples. The analysis showed how after finding similar functions we can discover the differences between versions of a specific malware family. We also showed that, although unconfirmed in other sources, the Globeimposter malware shows similarities with *TA505*'s, samples. In addition to this similarity, the other novel similarity found during this thesis between *Globeimposter* and *Gandcrab*, provides new insight into the connection between previously thought unrelated malware. Although the analysis showed interesting application it also exposed possible limitations of the method however, at the same time the analysis showed that features extracted from the greater function call graph could potentially further improve the similarity analysis.

5

Dynamic analysis

5.1. Theory

Data gathered acquired during the execution of a program is called dynamic analysis. With dynamic analysis, the execution of the program is followed as it is run on the processor. In order to gather this information, instrumentation needs to be added during the program execution. This can be done by attaching a debugger such that execution can potentially be paused and program variables can be read. Other options to capture the behavior of the software is by monitoring the program execution at the operating system level or even on (virtual) processor level. We will however use system calls to characterize a malware its behavior.

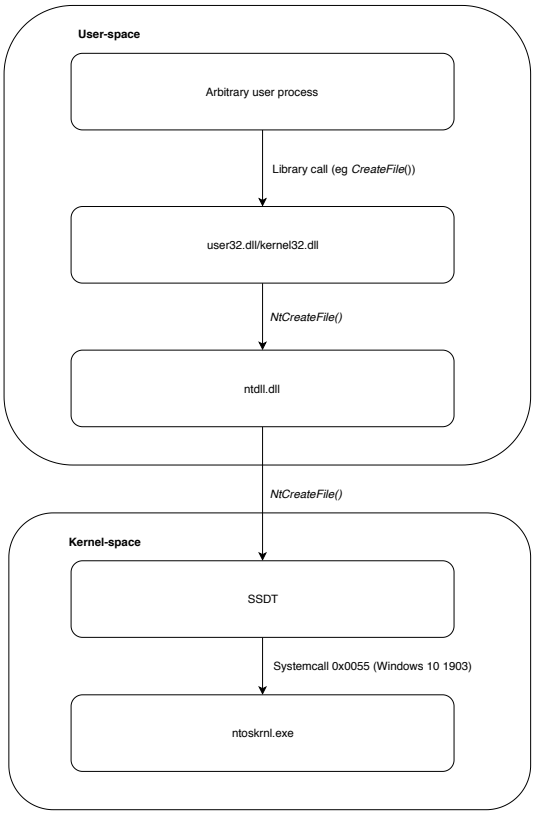


Figure 5.1: Windows architecture and telemetry placement

System calls are the lowest level calls made to the kernel and provide basic functionality such as file operations, register events, and threads starting and stopping. We will use these as a feature, because for malware or any other program to exercise any useful behavior it has to interact with the system and use system calls

to do so. It is not possible to build a useful application that does execute any system calls on the system, malware cannot escape analysis based on system calls, contrary to some other forms of dynamic analysis. Therefore, we argue it is the best feature to characterize a malware sample.

In addition, the analysis technique itself is completely stealth since system calls can be captured in kernel space while most malware runs in userspace therefore the malware is unable to detect this type of analysis. The benefits of dynamic analysis is that one does not have to worry about the encryption techniques on the binary, only on what the malware actually does. If the malware does a decryption step this simply gets recorded and can add information as characteristic of the malware.

Malware could however still attempt to obfuscate the actual behavior by inserting arbitrary dummy system calls such that the actual useful system calls that show information about the activity of the program gets lost in the noise. In addition, for the analysis to be successful it is required the malware actually executes, and executes the real malware behavior. This could be hindered by essential but offline command and control servers or the malware detecting it is being executed on a machine designed for analysis purposes.

Similarities among malware from the same family would mean that large parts of the behavior between versions remain te same. The difference among samples is then that there might be additions or removals in the trace, while the traces remain largely the same. The difference could then be visualized as a "diff" between two traces. When the execution profile of two samples then largely overlap with some gaps and/or additions it could be said that the malware are similar and as a result likely from the same author.

When malware is not from the same family similarities are likely much more subtle, and only small parts such a single function are shared among programs. This requires a different kind of method than when the similarities are broader. Given the system call trace of a sample, to find a similarity we now need to compare a block of system call events of arbitrary unknown length to all the blocks of sequential system calls of all other samples in the dataset. The problem can be illustrated in the following example, given three traces from two different malware samples, in reality, a system call trace for a single thread can consist of ten-thousands of calls:

```
a: [2, 3, 1, 2, 1, 2, 3, 4, 4, 5, 1, 2]
b: [8, 3, 2, 2, 3, 5, 1, 2, 3, 4, 4, 6, 1, 2, 5]
c: [1, 3, 5, 1, 8, 3, 1, 4, 6, 1, 2, 5, 8, 3, 2, 2, 1]
```

We want to find the patterns identified by color, as these are similar between these three samples. Two characteristics of the similarities can be identified:

1. The patterns do not appear at the same index among traces, instead they can appear anywhere, or even appear multiple times within the same trace. For example when the partial trace is the result of a function call that is executed multiple times.

2. One trace can have matches with multiple other traces that do not show similarities with each other. It turns out that malware sample *b* shows similarities with both sample *a* and *c*, however sample *a* and *c* do not show similarities with each other.

We argue that these two characteristics are a likely assumption, as the datasets consist of system calls per thread, without smaller subsections, such as functions, or block elements. Multiple functions can be executed in the same thread, and therefore if we want to improve the chances of finding similarities we argue it is better to find a method that is able to split the threads in sensible parts, and compare those instead. To strengthen our argument here we did a full thread n-gram similarity in subsection 5.5.2

5.2. Dataset generation

Dynamic analysis requires more preparation work than static analysis where the dataset is simply the malware binaries themselves. In addition, a lot more things can go wrong because of external factors than during static analysis. In dynamic analysis the data has to be gathered while the malware executes. A safe environment for the malware to run in needs to be set up such that the malware does not harm other systems and to actually record the data generated by the malware specific tools for gathering this data are required.

Since data gathering can take a significant amount of time we first explored existing datasets containing dynamic information. The existing datasets are however quite old (KDD [27] or NDD [74]) or do not contain sufficient information [26], since this set does not contain process/thread three information and uses the memory offset in the DLL instead of the full system call name. Therefore, the choice we made the choice to

collect our own dataset which has the additional advantage of having full control over the quality and over which parameters are collected.

For the secure malware execution environment, Cuckoo sandbox is used, this is an open-source framework specifically designed for malware analysis. It provides an environment that runs several analyses on the malware and provides a convenient way to submit large amounts of malware for analysis by managing a virtual machine that is able to restore to a clean state after a malware is done executing. For the virtual machine environment, Virtualbox was used, although a bare metal solution would be better to avoid detection of malware that do not want to execute on a virtual machine, a virtual machine cheaper, is easier to set up, and simpler to maintain since it we can simply revert to a previous clean snapshot after each malware execution. Considering this tradeoff the choice was made for a virtual machine setup.

In order to make the malware environment as up to date as possible, we use a currently recent version of Windows 10. All malware samples are run on Windows 10 with build number 1903 released May 2019. For this system, both the firewall and the malware scanner are disabled, such that the malware can run freely. Some simple measures to attempt to cloak the fact that the machine is a virtual machine, are adjusting Virtualbox parameters such as setting the system and motherboard vendor.

Although Cuckoo provides a couple of malware analysis tools it does not yet provide the exact data collection tools required for this research. The closest tool it provides is a tool that injects into the process, there are three problems with this tool. The first is that since it hooks directly into the binary it may be detected by the malware. The second issue is that it does not capture the malware system calls of the whole system as it has to inject into every binary to do this. The third problem is that it only captures a very limited amount of system calls, since our experiments have shown that some malware binaries had under a 100 system calls, this number is much too low to be a convincing representation of the real number of system calls.

We require a tool to record system-wide system calls that starts at the very beginning of the Windows boot process. Unfortunately, such a tool also is not available freely online, and therefore this tool is has been developed during this Thesis.

Data gathering tool requirements
The following formal requirements are setup to which the data collection tool needs to adhere.

- Capture all process IDs and parent process IDs, such that al full tree for all processes can be constructed starting from initial windows process (PID 4).
- Log all system calls made on a thread basis, such that context switching does not has an influence on the order of system calls for a process.
- Log process names, such that during the analysis it is known what program a PID corresponds to.
- The tool should be as stealth as possible to avoid the malware from being able to detect it is being analyzed.

Logging system wide system calls
System calls can be recorded using multiple strategies. However, some methods do not provide enough data or are countered by malware. Applications such as Microsofts Process Monitor do not capture enough data and can be fooled by malware. An option to capture the system calls made a specific binary is by hooking into the malware. However, this has two problems, if we want to capture system calls system-wide we have to hook into each process, besides that there are a lot of processes running on Windows system, the hooking into processes can cause instability, and it is also not possible to do this during the boot of the system. In addition, as discussed in subsection 3.1.4 more advanced malware could detect the hook. Therefore, we desire a technique that is more stealth and captures all system calls starting from the very first process on boot.

Registering the system calls in kernel space makes it harder or even impossible to detect such applications. A common technique to do this is by replacing entries in the System Service Descriptor Table (SSDT) with custom versions that contain logging capability. However, in recent Windows 64 bit versions the SSDT table is protected by a security measure from Microsoft called Patchguard. This makes it no longer possible to make changes to de SSDT, besides this problem in order to actually record all system calls, all definitions need to be replaced by custom ones, this could be quite hard to do since the correct definitions needs first to be found. Most of these functions are not documented officially and may even differ among Windows versions. We therefore, chose to use a lesser well-known technique that works with recent versions of Windows 10.

We will use a method for system call hooking that takes advantage of a hack in the eventlogmanager that allows to hook into the process the moment a system call is made. We can then record additional data of the process doing the system call. Besides the system call number the process id (PID) and parent process id (PPID) are recorded for each system call. Such that the process hierarchy can be created in the form of a tree. Unfortunately, windows does not keep track of the parent of a thread, therefore we are unable to hierarchically order threads under a process.

The developed tool is implemented as a Windows Driver, logs are streamed to the Windows event log, after the execution of a binary on the virtual machine this log file is sent to the host. Because the event logs to are stored in a combination of binary and XML the file is extracted using another tool to CSV 5.2. The exported CSVs can be several hundreds of Megabytes, therefore the individual CSVs are compressed using *XZ* compression algorithm, to reduce the size to only a few Megabytes per trace.

The only downside of the tool is that it requires Windows to run in so-called Test-Mode because Windows demands drivers to be certified by Microsoft, which this custom driver. For this research, this is not possible and therefore it requires to execute the analysis in Test-mode. Test-mode may be detected by malware however there is not really likely since it is not a mode analysis platforms generally require. Also, a regular non-analysis user might also use this mode, for instance on an older system where newer certificate drivers are not available for.

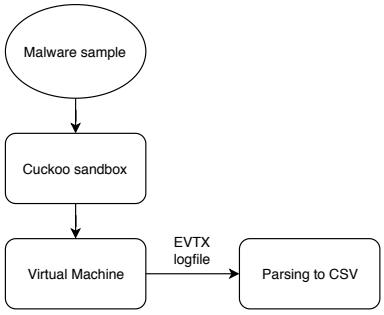


Figure 5.2: Data collection pipeline for single malware sample

We can compare this data gathering method to the built-in method used in Cuckoo, to see how it compares in its ability to capture executions traces of system calls. The lists of generated system calls are not directly comparable since Cuckoo captures system calls on a user level, while the driver captures them on the kernel level and not all system calls can be directly translated from user to kernel mode. However, to give an indication of the difference in the number of system calls between the two instrumentation methods we will compare the number captured system calls of some malware samples.

In Table 5.1 the system call counts are displayed of the initial malware process for four different malware samples. On average the kernel drivers captures 24 times more system calls using the kernel driver than the Cuckoo hook does.

	Total	Unique
Cuckoo	268	28
Driver	7721	90

(a) Emotet

	Total	Unique
Cuckoo	2072	54
Driver	60715	128

(b) Qbot

	Total	Unique
Cuckoo	6044	36
Driver	67373	82

(c) Ramnit

	Total	Unique
Cuckoo	992	41
Driver	27881	159

(d) Dridex

Table 5.1: Comparison of system call capture performance between Cuckoo and the new kernel driver

5.3. Dataset exploration

Several datasets were collected, from Malpedia and the malware analysis platform Any.Run [1], this last source was chosen because it contains samples for very recent malware. In addition, the platform also runs the malware and generates a report from which can be validated the malware still works. From this platform, we only use malware that is provided in Microsofts PE executable format, in other words regular "exe" files and not malware embedded in Microsoft office documents or Powershell scripts. The datasets from Malpedia are the same as were used for the static analysis, by using the same dataset we can compare the similarity results found between the static and dynamic analysis. The downside of this dataset is that the samples are older, therefore they may no longer show their original behavior when run on our analysis platform.

For each sample, a full trace of every system call by every process on the Windows machine is recorded starting from the boot of the machine. Each system call is tagged with the executing process id (PID), Parent Process Id (PPID), Thread Id (TID).

Besides this, every system call has a timestamp on Nanosecond precision, such that the order of the system calls for each thread is preserved. Unfortunately, the attached PIDs are not unique, because after a process or thread exits the ID can be reused for a new process or thread. Therefore, we retag duplicate IDs using algorithm 3 by looping over all processes sequentially, once a PID is found that had a different PPID in the past the new PID is uniquely retagged. This still leaves the possibility of a duplicate PID for the same PPID, however, we choose to ignore this given the very small possibility for this to occur.

Algorithm 3: Repairs re-used process IDs in trace of system calls

```
Input  : Sequence of PPIDs and PIDs of length l
Output: Repaired sequence of PPIDs and PIDS of length l, where re-used IDs are suffixed with their
        re-use count
duplicate-pids ← {}
foreach row in input sequence do
    if dupplicate-pids[pid] != row[ppid] then
        duplicate-pids[pid].append(ppid)
    end
    if length(duplicate-pids[pid]) > 1 then
        row[pid] ← pid*length(duplicate-pids[ppid])
    else
        if length(duplicate-pids[ppid]) > 1 then
            row[pid] ← pid*length(duplicate-pids[ppid])
        else
            row[pid] ← pid
        end
        if length(duplicate-pids[ppid]) > 1 then
            row[ppid] ← ppid*length(duplicate-pids[ppid])
        else
            row[ppid] ← ppid
        end
    end
end
```

Since the PPID and PID are known for each process, a full tree can be generated that has at its root the first windows process. An example visualization of such a tree can be seen in Figure 5.4. In summary, the full dataset and the problem statement can be visualized as in figure Figure 5.3. The red arrow shows the problem of process injection, while the bold system calls give an indication of the similarity we desire to find.

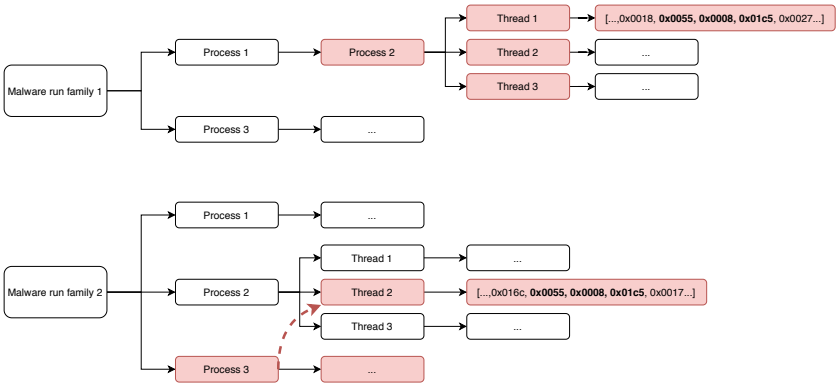


Figure 5.3: Dataset summery, red indicates malware process

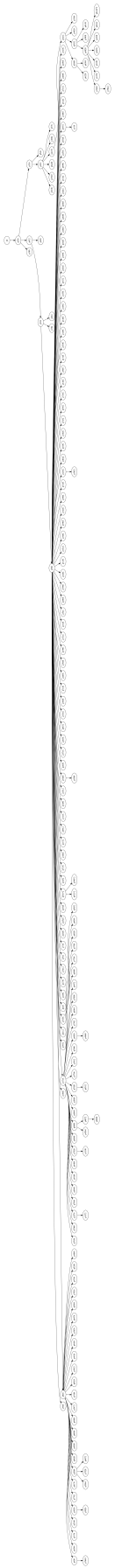


Figure 5.4: Example of full process tree.

The allowed runtime of the malware is decided following the following reasoning. Firstly we have limited analysis time, if the malware decides to stall, we will therefore not see any behavior for this malware. We expect each malware sample to finish within a minute, but also need some extra time to make sure the Windows event logger writes the system calls to file, therefore total run time can be up to 2 minutes. The second reason for keeping the analysis as short as possible is to limit the amount of data we need to process. Although generally the more data the better, it can also become too much to analyze without a large scale server. To give a general idea about the dataset size we are working with, an average system call trace of the full system contains about 8 million system calls. These are generated by around 250 unique processes and 1700 threads, depending on the malware this may be some more or less.

We can reduce the dataset size by only taking into account processes and threads that are started after the malware is launched. This is possible since the malware is running in a closed environment. The anomaly detection of injected threads is not the main research goal, we are only interested in a way of finding similarities among malware.

In Figure 5.5 the captured system calls over time are plotted for the execution of an arbitrary malware sample. Interestingly we see that there appears to be a clear maximum on the number of system calls per second. We checked what could be the cause of this since the system does not appear to be slow. We verified that the CPU is at least not the bottleneck since it is utilized less than 10 percent most of the time according to the Windows task manager.

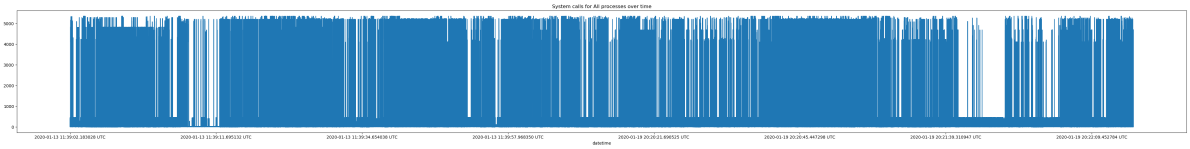


Figure 5.5: System calls over time of an analysis run of Gozi malware. (large time difference between start and finish in the graph is due to the machine being restored from a snapshot)

5.4. Feature extraction

Given infinite computation time to find the similarities among all traces of all threads of all malware runs, we would compare each piece of system call sequence of all possible lengths to all other system call sequences. This process is in practice however unfeasible, given the large time complexity.

Therefore, we propose multiple possible methods that should still provide us with the similarities we are looking for, but reduce the computation time significantly. The methods need to meet a few requirements:

- Fast enough to find similarities among large (hundreds) of unique malware samples within minutes.
- Is theoretically able to find similarities among malware families.
- Is theoretically able to find similarities within the whole system and not just the main malware process.
- Only requires system call traces of the full system as its source dataset.

To meet these requirements we propose a couple of methods that either reduce the dimensionality or or can be used to find the similarities.

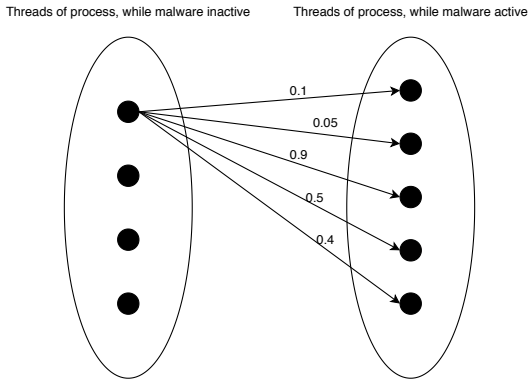
- Word2Vec (or in this case Syscall2Vec)
- Sequence alignment
- N-gram based features
- Frequent pattern mining

None of these methods however, provide the means to filter out the malware traces, that are injected into another process. Therefore, if we do not filter the malicious threads out first, we may find similarities between two completely benign threads. In an attempt to resolve this and filter the malware threads, we tried using a technique called maximum weight matching. This technique is useful since the problem can be viewed as an assignment problem.

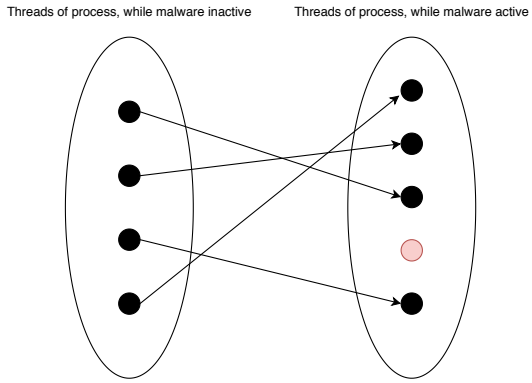
Given the traces of two of the same processes, one where the malware is active and has injected into the process, and one where the malware is not active. The threads of two processes can be represented as two disjoint sets, when we try to match the threads from one set to the threads of the other set, something that looks like a bipartite graph is formed, since almost all threads from one set has an edge to a thread of the other set. We say almost, because a property of a bipartite graph is that all nodes are connected, this is no true for our case as the malware thread their nodes should not be connected to a thread node in the benign process, since these only exist in the malicious process.

So we will try to find the threads in the right set of Figure 5.6b, that do not have a matching thread in the benign process. Which edges are formed between the two sets is determined based on the amount of similarity among the threads of the different sets.

First, the similarity is calculated between the threads of the different sets, this is illustrated in Figure 5.6a for a single thread of the benign set of threads. In the words of maximum weight matching, this similarity is the weight of the edges. After this weight is determined for all pairs of threads, maximum weight matching finds the most optimal match for which the sum of the weights of the matched edges is as large as possible. Some threads will not be matched since each thread is only allowed maximally one edge, these unmatched threads classified as the injected malware threads.



(a) Similarities for a single thread of the benign process, to all threads of the process infected witht the maliciously injected thread.



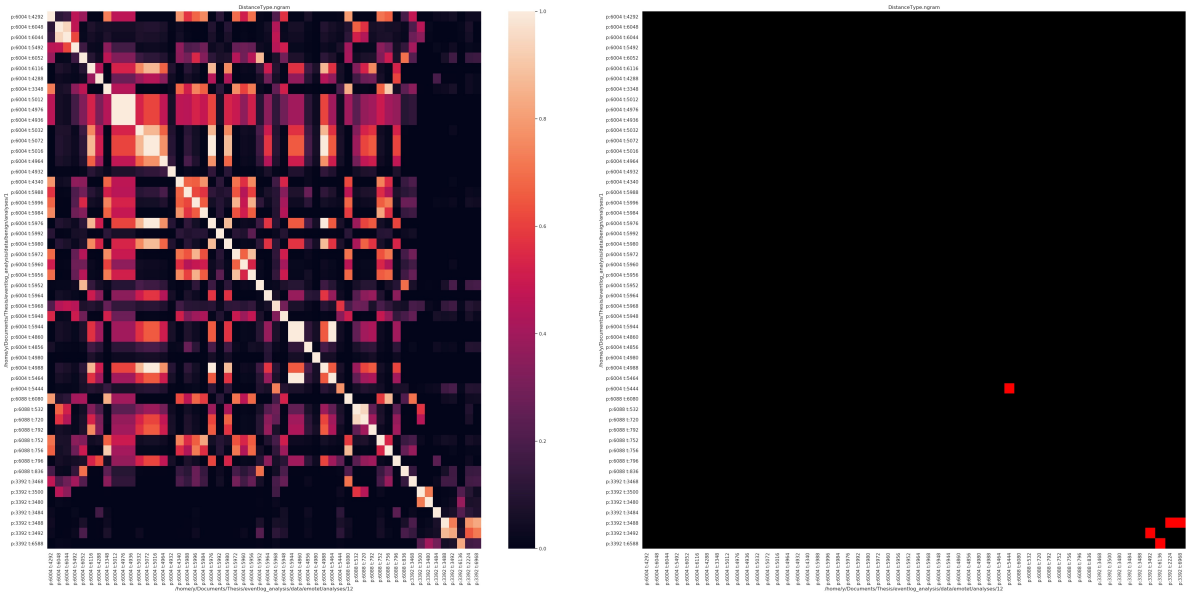
(b) Illustration of bipartite matching, each dot is a thread, the red dot represents the malicious thread we are trying to find.)

Figure 5.6

The above method works well under the assumption that the new threads of a benign process are directly caused by a malware injecting itself into that process. We can demonstrate this with a sample of *Emotet* malware which is known to inject itself in the benign process *explorer.exe* already running on the system. It injects two threads, we can visualize this by doing a cosine similarity on the bi-gram distribution of the two runs. The cosine similarities are plotted in Figure 5.7a, on the y-axis the threads of *explorer.exe* from the benign run are shown, and on the x-axis the threads of *explorer.exe* of the malicious *Emotet* run are shown. These similarities can be used to find the most likely match of each of the threads from the malicious run in the benign run. They injected threads will not match with any thread of the benign run and therefore we can put a threshold on the most the set of most similar threads. We are then left with four possibly injected threads as shown in Figure 5.7b. Maximum weighted matching identifies the two injected threads directly as in this case being thread 2224 and 6968.

Unfortunately in practice, the previously made assumption appears too strong, since a malicious process may cause the start of benign threads as a side effect of malicious behavior. When a malware process causes this, we can no longer distinguish the malicious from the benign threads, since the injected process now has benign threads that cannot be matched to any of the benign threads from a benign analysis run. Even when the same binary is run twice on the malware analysis system, it does not produce the same system call data, this is especially true for the Windows process *svchost.exe* which is responsable for 60% to 70% of all threads during the analysis of a malware sample on the system. *Svchost.exe* however, is an essential application on Windows systems as it manages process sharing of services. Possible explanations for this non-deterministic behavior are the current system time, system resource availability, and probably most importantly internet usage by various components of Windows 10.

Since we do not have a direct solution to the problem of extracting all malware traces of the whole system with our dataset besides the maximum weight matching approach, experimented whether we could limit the



(a) Cosine similarity

(b) Thresholded cosine similarity on 0.85

Figure 5.7: Threads of *explorer.exe* from a benign and malicious run

number of processes we attempt to extract malware information from, based on information from previous work. The limited process set is selected based on the work of .... this work uses a more advanced method for detecting the process injection, which is possible since they are able to follow the malware activity using taint. Their dataset consisted of 40 malware families, including the more infamous samples such as Dridex, Emotet, and Ursnif. From their work, we use the reported results of most common target processes used for injection. The most used process with 65% out of their samples used *explorer.exe* as a target for process injection. On the second place is *iexplore.exe* but with only 30%, followed by *connhost.exe*, *taskhost.exe* and *dwm.exe svchost.exe* is only used in 15% of the samples.

We will first assess how well we can extract the injected process from our dataset using an Emotet sample, a benign sample and the maximum weight matching method. We use an Emotet sample since it uses all of the above processes. We already explored *explorer.exe* successfully in Figure 5.7 which we will therefore use in the next step.

5.4.1. System call datasets

In this subsection we will describe the characteristics of the malware datasets used for the dynamic analysis based on the system call profile.

Dataset analyzed during static analysis

The same malware samples were used for the static analysis, by using this dataset we are able to compare the results from the static analysis with the results from the dynamic analysis. We used both the memory dumped samples and the packed samples. The memory dumped samples are the same as used in the static analysis of the malware. In the next sections we will refer to these two datasets as *Malpedia-a* (107 samples) (Figure 5.8) for the memory dumped set and *Malpedia-b* (Figure 5.9) for the non-memory dumped dataset. Not all samples had a non-memory dumped version available, therefore some samples that appear in the dumped analysis do not appear in the non-dumped analysis. The choice to keep these two datasets separate even though they originate from the same malware, is made because the memory dumped and non-memory dumped malware are different stages of the malware. We argue that malware from the same stage in execution is more likely to be similar to each other than malware traces from different stages.

Figure 5.8, Figure 5.9, and Figure 5.10 show some of the system call dataset characteristics for the different sets. All but sub-figures d and f are self explenatory. Sub-figure d displays the number of samples that share a single bi-gram. For example in Figure 5.8 a bi-gram with index 250 appears in about 45% of the samples of that dataset. Sub-figure f shows how many of all the non-unique bi-grams of a sequence can be explained from the top-10 bi-grams of that sample. This metric gives an indication of the number of repeated system

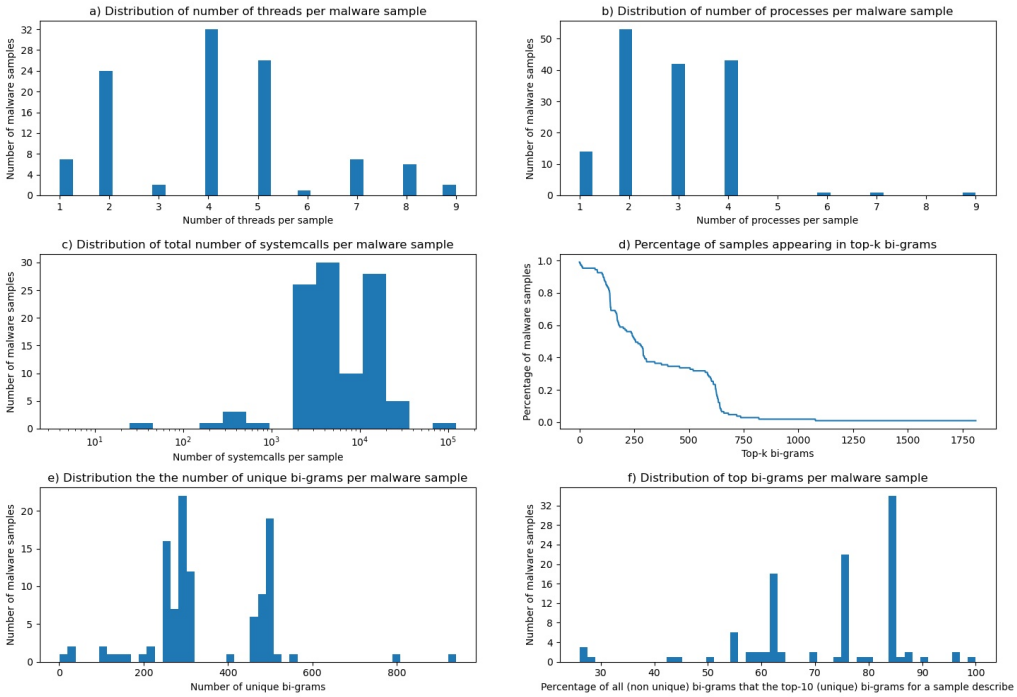


Figure 5.8: Data distributions of static memory dumped set (Malpedia-a). (Two samples were removed from this plot as they had a very large number of threads compared to the rest of the data (120) threads)

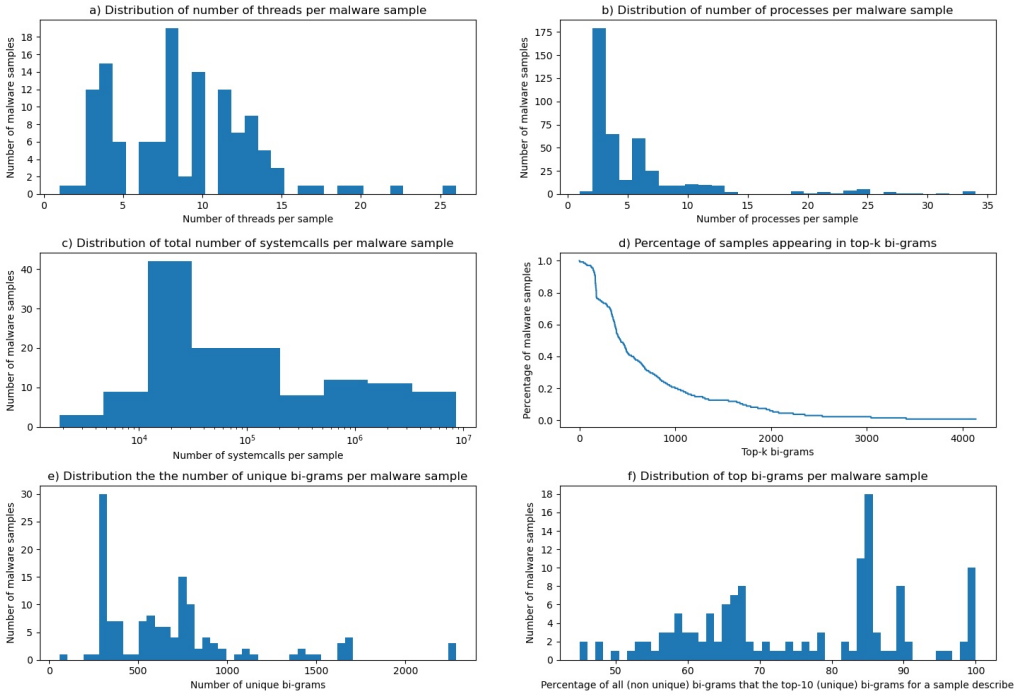


Figure 5.9: Data distributions of static non-memory dumped set Malpedia-b.

call sequences in a sequence.

Dataset of currently popular samples
This dataset is sourced from any.run[1], any.run provides a paid malware analysis service that features malware analysis reports similar to what Cuckoo provides. The samples were selected individually based on the reported malware trends the tracker to be popular around January 2020. The reason for choosing this dataset is that since the malware samples are very recent the command and control servers are still online. This allows us to see what effect this has on the malware's analysis results, the hypothesis is that these malware samples show more activity than the older malware from the previous dataset.

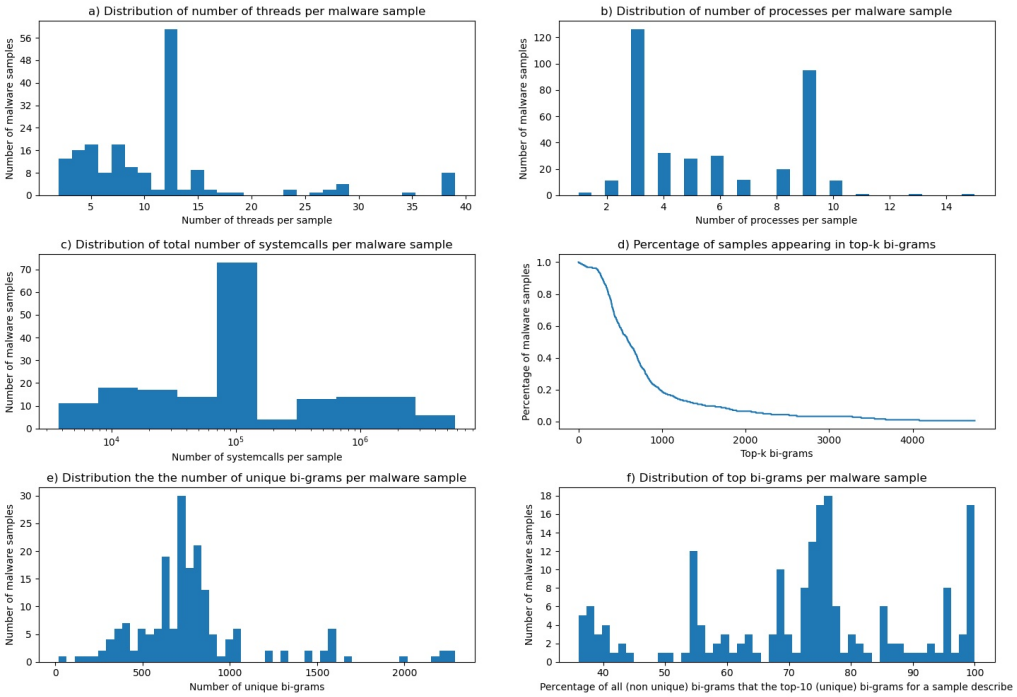


Figure 5.10: Data distributions of Any.run dataset.

Comparing the datasets to each other we see that Figure 5.8 relatively contains fewer threads and processes per sample than the samples of the other datasets. The number of unique bi-grams per sample lies mostly below 1000 for all three datasets. All datasets appear to have a lot of repeated sequences for each sample. We investigated what kind of system calls these were and found these system calls to be mostly related to memory operations, as can be seen from Figure 5.11.

5.4.2. Word2Vec

Word2Vec is an algorithm to create word embeddings, that capture the context a word appears in. Intuitively it puts words that appear in the same context often (and thus close to each other in a piece of text), closer to each other in a multidimensional vector space than words that do not often appear in the same context with each other.

The same idea can be transferred to traces of system calls, however instead of words of natural language the used words are system calls. We hypothesize that system calls such as `NtCreateFile`, `NtWriteFile` lie closer to each other than to `NtOpenThread`. After training a Word2Vec model the similarity can be visualized as in Figure 5.12. Note that system calls of the same category do not necessarily lie close to each other. For instance, the opening and closing a thread are unlikely to directly follow each other directly.

Although Word2Vec appears to give desirable results for some datasets it does not provide consistent clusters of system call categories across datasets of system call traces, as the training vocabulary grows the relation between the different system calls becomes less clear. In addition, reducing the data to a few clusters of sys-

[1] https://any.run/

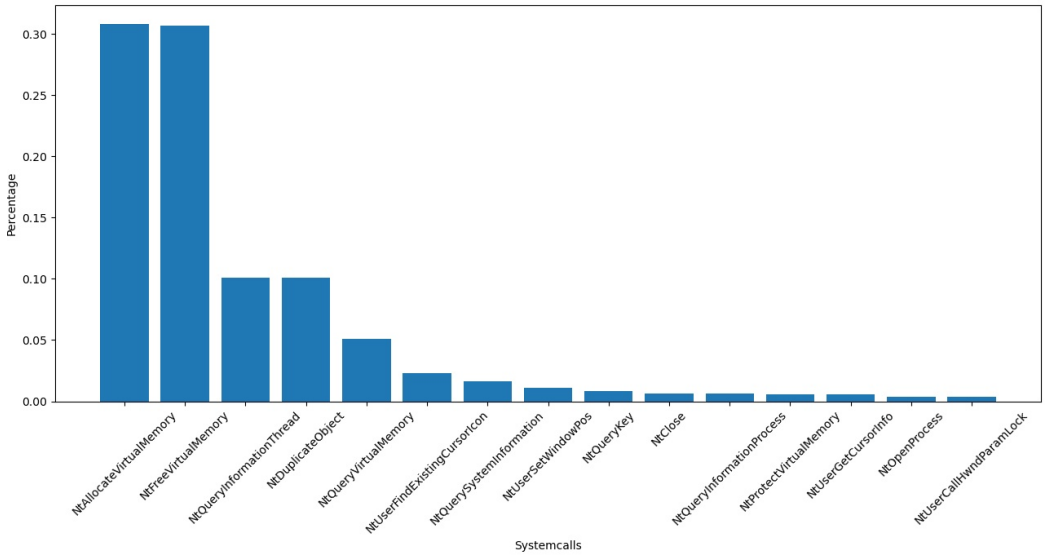


Figure 5.11: Most common system calls in any.run dataset, other datasets show the same kind of pattern.

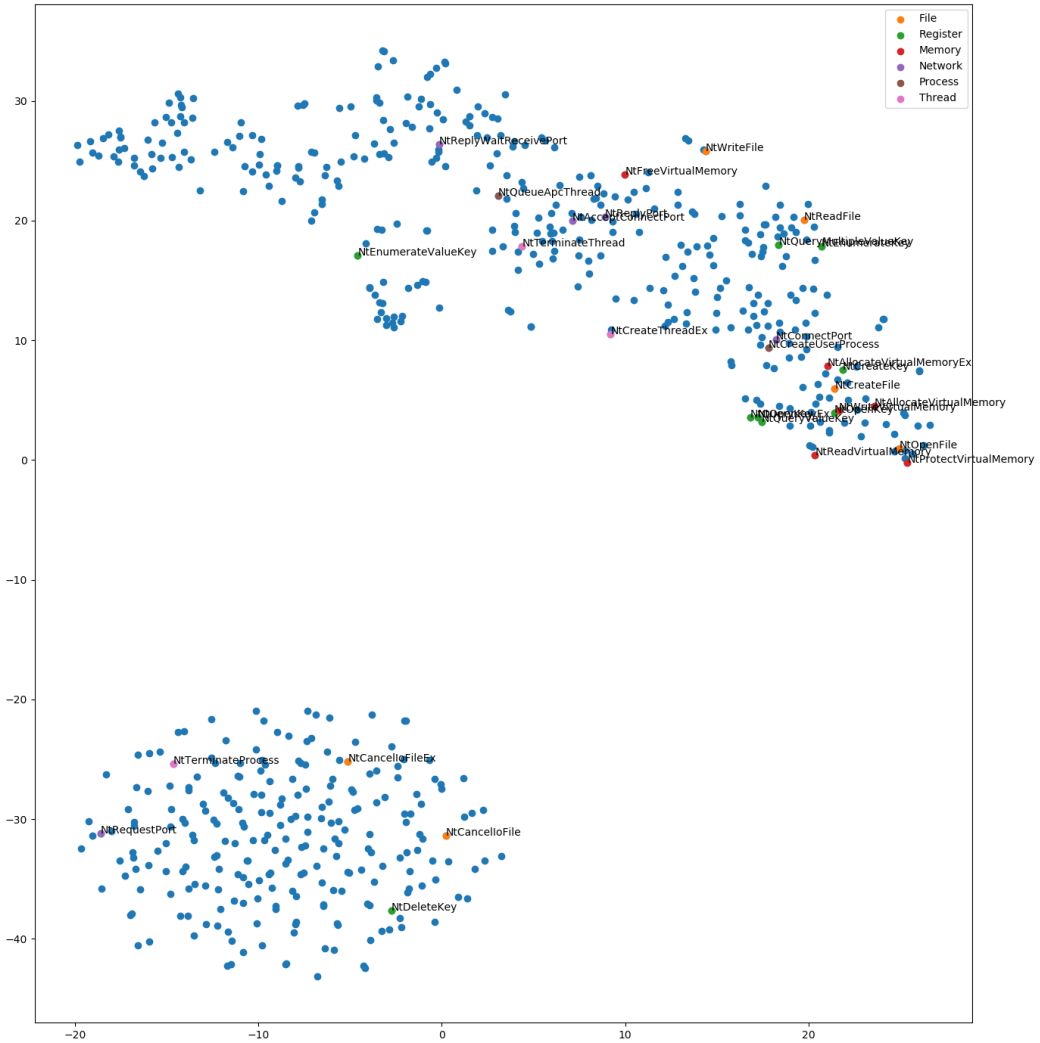


Figure 5.12: Sample scatter plot showing how the system calls relate to eachother acording to a word2vec model.

tem call groups identified by word2vec does not necessarily result in the desired data reduction as smaller patterns will evidently disappear. To compensate for this the patterns can be made longer.

5.4.3. Sequence alignment

Sequence alignment using for instance the famous Smith and Waterman algorithm which had its first application in DNA sequence alignment. Has been used previously in literature subsection 2.2.2, however it is not the best solution for our research question. Firstly, because sequence alignment is better at finding differences than finding similarities. Secondly, we argue that a system call sequences are very different from a DNA/RNA sequence. Something a biology targeted sequence alignment algorithm does not have to take into account for instance is re-occurring patterns that appear an arbitrary amount of times. This could happen when the same function is executed multiple times in a loop. In addition, system call sequences are much more complex than DNA/RNA sequences, as the vocabulary of different system calls is much larger than DNA bases which only consist of four unique elements. For these reasons, we will not use a sequence alignment approach as a possible answer to our research question.

5.4.4. n-gram based

Since the system call data is sequential data they can be vectorized using n-grams. These vectors can be used directly with cosine similarity to determine the similarity between two vectors and indirectly the malware samples. Using n-grams can however result in large vectors, since there are a total of 1920 unique system calls and the vector size grows exponentially with the size of the n-gram. For bigrams, the number of possible features is $1920^2 = 368400$ elements per vector and for trigrams a single vector can already contain $1920^3 = 7077888000$ features.

Unfortunately n-gram vectors alone do not provide a solution for the issue of finding smaller similarities than on thread level, therefore we are not really satisfied with such a solution.

5.4.5. Reduction of the sequence to a Markov chain

Another option to create features is by first transforming a thread to a Markov chain and then using the number of outgoing and incoming edges for each system call. This results in a feature vector of only 3840 elements per thread.

The Markov chain can also be viewed as a graph, then we can do similarity calculation between graphs from different malware samples based on the maximum common induced subgraph, which is a well-known problem. If two graphs then have a larger common subgraph than a different pair of graphs they are more similar than the latter. However, this problem is known to be NP-complete [21], and our graphs are too big to process.

Besides, being a way to generate features the created Markov chain as illustrated in Figure 5.13 can also provide a global overview of the activity of a thread.

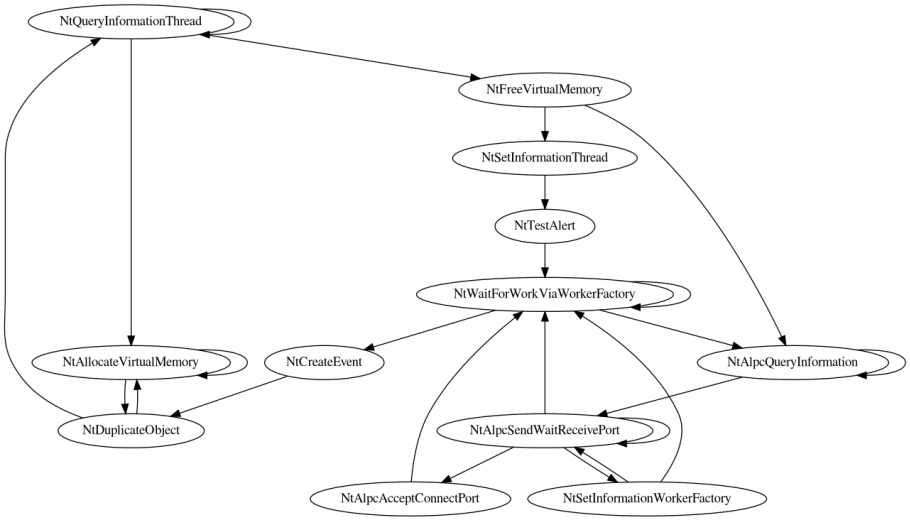


Figure 5.13: Sample graph showing the system calls displayed as graph, arrows indicating call order.

Like n-grams this works on thread level and therefore does not find similarities and does not provide a way to find similarities on a smaller granularity.

5.4.6. Frequent sequential patterns and frequent item set based

To resolve the problem of finding similarities smaller than on full thread level, we propose to use mined frequent patterns from the dataset. Extracting frequent patterns directly over the full dataset would not be ideal, since the system calls produced by the malware are only a fraction of the total amount of logged system calls. The patterns of background activity that appear in all datasets would then have the highest support, and for our goal we are more interested in the patterns that still have high support but lower than the background system calls.

We can identify two types of frequent patterns, the first is sequential frequent patterns, and the second are frequent itemsets. The difference between the two is that in the latter the order of the elements is irrelevant to the mined pattern. A frequent sequential pattern can be defined as a sequence S_a with the sequential $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_k$ which also occur in another sequence S_b with sequential elements $y_1 \rightarrow y_2 \rightarrow y_3 \rightarrow y_k$ such that $S_a \sqsubseteq S_b$. The same holds for the frequent itemset, except that the order of the elements of S_a and S_b is irrelevant. The number of times a pattern occurs in a dataset divided by the total number of found patterns is defined as the support of a pattern.

A variety of algorithms to mine frequent patterns can be found in the literature. Not all algorithms will however work given our dataset. A problem with algorithms famous algorithms such as PrefixSpan [58] or Apriori [15] is that the extracted patterns database is too large and becomes worse when the sequences are long [73]. Their example of large sequential patterns contains 20 distinct items resulting in 2^{20-1} subsequences. Compared to our dataset 20 distinct items is relatively little as our sequences contain on average 2000 items, with a standard deviation of over 7000 items and 1722 possible unique elements. In addition, they are too slow and do not scale to large input databases as they rely on dynamic programming.

Therefore, an efficient algorithm that limits the amount of extracted patterns to only the most relevant ones is necessary. Closed frequent patterns algorithms do this by only outputting patterns that are not included in another (super-)pattern that has the same support. Or more formal, the pattern S_a is closed when there is no other sequential pattern S_b, such that $S_a \sqsubseteq S_b$, with the same support. This definition however still allows for a lot of extracted patterns.

A maximal frequent pattern algorithm limits the amount of outputted frequent patterns even further than an algorithm that finds closed frequent patterns, by making the previous condition more strict and not allow super patterns regardless of the amount of support. The VMSP algorithm by Fournier-Viger[73] extracts these maximal frequent patterns and reduces the computational cost significantly by mining only maximal sequential patterns. For frequent item sets FPClose [54] for mining closed itemsets, and FPMax [29] for mining frequent maximal itemsets.

5.5. Method

We will evaluate two methods to find similar malware samples. The first is by taking cosine similarity over full threads. This method is however not able to capture the finer granularity similarities, such as when two or more threads are for the most part different from each other while only a small part matches. Therefore, we propose a second method that uses mined frequent patterns, to split the threads into smaller interesting sequences that can be used to find similarities among threads.

Both methods require that the malware specific patterns are extracted from the larger dataset of all system calls logged on a system.

5.5.1. Filtering malware system calls

To filter the relevant malware threads we will first use the process tree, all threads that have as their ancestor the initial malware thread are labeled as malware processes.

Since filtering all injected malware threads with the dataset appears to be too hard to be just an intermediate step in this research towards finding the similarities between the malware samples. We will only focus on two targets that have been identified as popular choices for process injection in previous research. We will filter injections on *iexplore.exe* and the processes spawned in or by *explorer.exe* since these can be filtered out with either maximum weight matching or by extraction based on the parent process.

5.5.2. Similarity analysis

n-gram analysis

The first method to find similarities is to directly convert the malicious threads to bi-gram vectors. The method is analog to the method used during static analysis. The difference is that instead of instructions, the sequences consist of system calls. After extracting bi-grams from all threads of a malware sample the vectors from these threads are merged together, then cosine similarity is used to find the similarity among the vectors of the different samples. The similarity can then be clustered using HDBScan and transformed to 2d using T-SNE, such that the results can be visualized. The implementation makes use of sparse matrix representation to keep the memory requirements with these large vectors within practical boundaries.

Custom clustering

The second method (Figure 5.14) that is supposed to find finer granularity similarities, involves a bit more work just taking cosine similarity over *n*-grams derived from the system call sequences. The general idea about this method is that allows clustering without satisfying *triangle inequality*, since the method does not use a geometric distance such as the Euclidean distance. Triangle inequality says that for any triangle (in our case each datapoint is a corner), the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side. For clustering, this means that given three points, two of these points can lie very far from each other, while they both lie close to the third point.

For this method, we will leverage frequent pattern extraction in two passes to find similar patterns among threads on a finer granularity than thread level. It is important to note that unlike research that is focused on finding similarities among documents we are not after clustering the documents (or in our case threads). Instead, we cluster patterns extracted from the documents, from the found clusters of frequent patterns, the malware clusters can be inferred. The found clusters cannot easily be visualized since the similarities are not given by a single similarity for the full system call pattern. This is a result of the second characteristic of the similarities identified in section 5.1. Because of this, we cannot simply put samples that share some similarity in the same cluster.

Given a database consisting of malware threads and their executed system calls, we will first extract frequent sequential patterns from the system call patterns. These sequential patterns do not contain gaps, and are exact matches of the system call sequences. The frequent sequential patterns provide an intelligent method to break up the system call sequences in smaller subsequences. When a subsequence appears more than one time it means two or more samples likely share similar behavior. Each found pattern is tagged with the threads it was found in, which leaves us with a set of all frequent sequential patterns and the threads they were found in. It is important to keep the amount of found patterns as low as possible for the next step, since the next step is more computationally expensive for larger datasets. To keep the frequent patterns as low as possible, the minimum size of the patterns needs to be as high as possible while still finding enough interesting patterns.

Now the second step is clustering multiple frequent sequential patterns together in order to make the found similarities more relevant. We can do this based on the threads the found frequent sequential patterns origin from. Once again we want to be able to find partial matches, and we do not want to penalize sequential patterns that do not match with other threads. In order to find these partial matches, we will do a second pass of pattern extraction. Except this time the order of the patterns is irrelevant since we now find frequent patterns of sets of threads. After extracting the patterns of threads we can translate the thread identifiers back to their corresponding malware applications.

After this second step, the number of resulting patterns of malware samples sharing sequential patterns is still quite large. In order to make the output more useful, the output can be pruned based on the support. In other words, the number of found malware clusters will be limited by requiring a minimal number of frequent sequential patterns that need to match between two or more malware samples. This will effectively filter the patterns that are shared between almost all compared samples and do not provide specific information to find similarities among specific malware.

In order to clarify the above steps provide an example of the workings of the method below. The colored text gives an example of how sets move between the different steps of the algorithm. Given dataset consisting of the following system call traces extracted from a total of five malware samples, as given in Table 5.2.

Only three of these malware samples resulted in shared patterns among threads, the resulting extracted frequent sequential patterns are displayed in Table 5.3.

Using the sequence IDs as the input for frequent set extraction we find the frequent items sets of Table 5.4. One found frequent item set now represents that at least two or more threads share multiple previously found

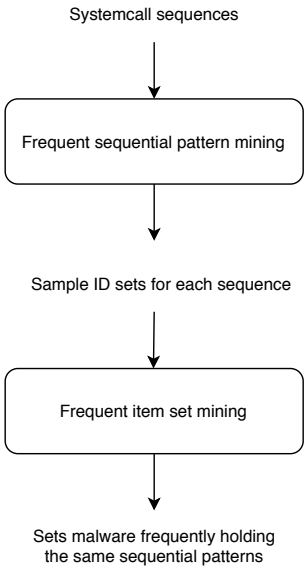


Figure 5.14: Diagram explaining the frequent pattern extraction clustering method

Sequence ID	Sequence Reference	Sequences
1	Malware sample 1, thread 1	2, 3, 15, 20, 7, 8, 9, 10, 20
2	Malware sample 1, thread 2	3, 15, 9, 10, 1, 2, 7, 8, 13
3	Malware sample 2, thread 1	9, 10, 7, 8, 1, 2, 19
4	Malware sample 2, thread 2	3, 15, 12, 16, 5, 6
5	Malware sample 3, thread 1	3, 15, 11, 5, 6, 21, 11
6	Malware sample 4, thread 1	7, 8, 4, 5, 6, 4
7	Malware sample 5, thread 1	9, 5, 22, 9

Table 5.2: Extracted malware system call traces

Pattern ID	Pattern	Support	Sequence ID
1	1, 2	2	2, 3
2	5, 6	3	4, 5, 6
3	7, 8	4	1, 2, 3, 6
4	9, 10	3	1, 2, 3
5	3, 15	4	1, 2, 4, 5

Table 5.3: Found sequential patterns

sequential patterns. We can remove the patterns that have too few occurrences (support) or are too short to be relevant, as the higher the support the more often these threads are seen to share sequential patterns together. The resulting frequent item sets can be visualized as groups of sequences in Table 5.5.

Frequent Pattern	Support
{6}	2
{4, 5}	2
{2, 3}	2
{2, 1, 3}	3
{2, 1}	3
{2}	4

Table 5.4: Frequent item sets

	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5
Sequence 1			x	x	x
Sequence 2	x		x	x	x
Sequence 3	x		x	x	
Sequence 4		x			x
Sequence 5		x			x
Sequence 6		x	x		

Table 5.5: Combinations of found patterns for the system call sequences, the colored boxes indicate the found frequent itemsets of sequences that share multiple frequent sequential patterns. (The columns refer to the first found sequential patterns and not the frequent item sets.)

We can interpret the results from Table 5.5, as malware sample 1 and sample 2 sharing multiple patterns and are therefore more similar to each other than the sequences of malware sample 3 and 4. In addition, malware sample 2 also shows similarities with malware sample 3. Although malware sample 4 shares a pattern with both samples 1 and 2 and another pattern with sample 3, it does not have enough in common with a single sequence to be labeled as similar. For malware sample 5 it was already shown during the first pattern mining step that it does not yield similarities to the other samples.

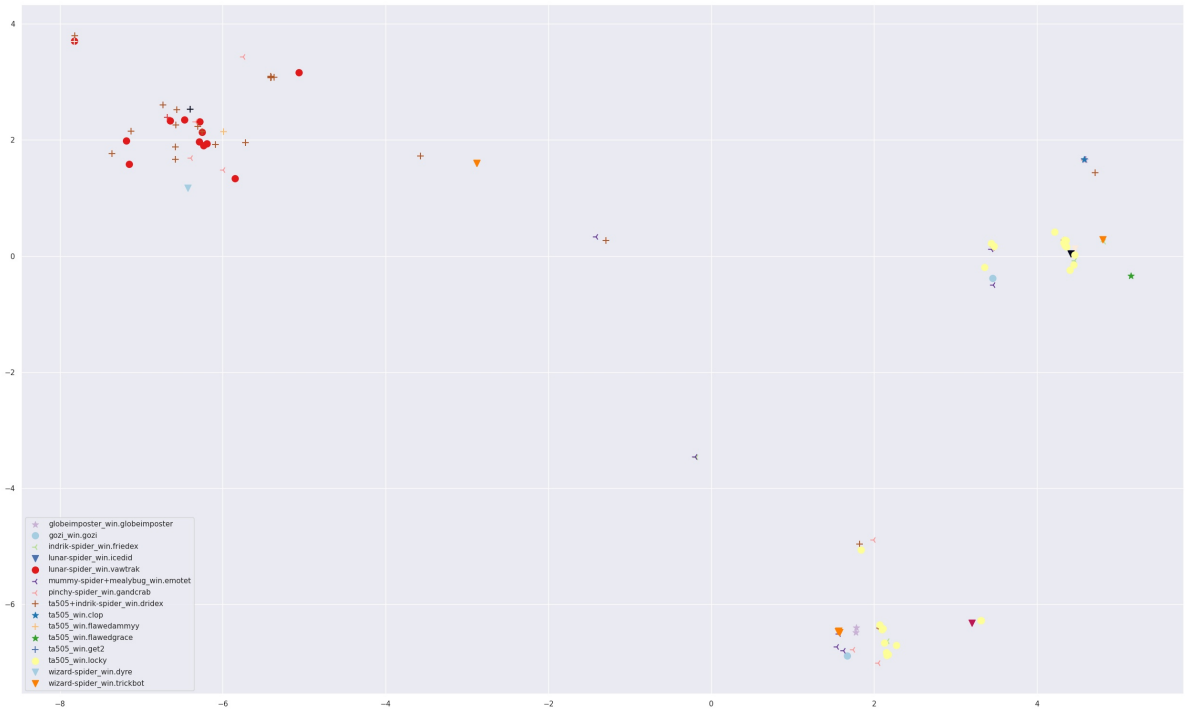
5.6. Results

First, we will show the results from the full sample n-gram analysis on both Malpedia datasets. We will discuss the results both qualitatively and quantitatively. After this, we will provide the results from the function level clustering method on the same datasets.

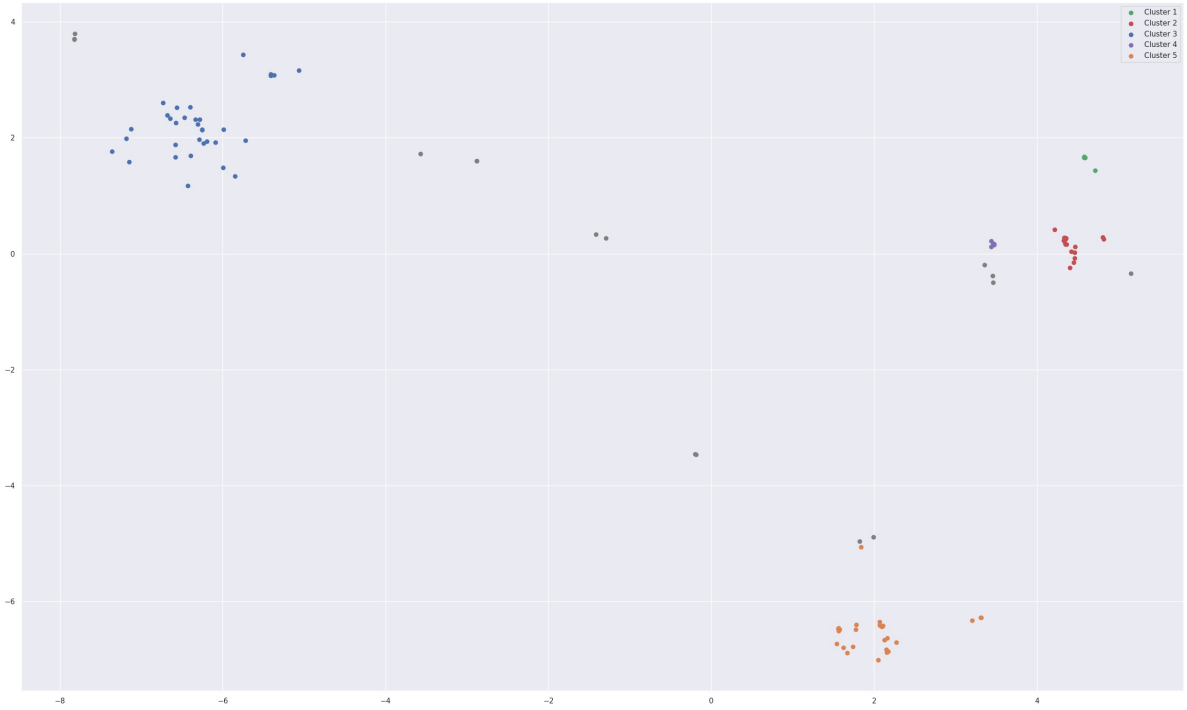
5.6.1. Full binary n-gram vector analysis

Figure 5.15 shows the result from the dataset of memory dumped malware samples. This is the same dataset as was used for the static malware analysis in chapter 4. We also use the non-memory dumped version of these binaries, the results from the clustering of this dataset are displayed in Figure 5.16. In figure (b) for both of these figures, the HDBScan algorithm is used to find clusters in the original non-projected cosine distance matrix.

The clustering results will be analyzed by inspecting the source data, to find interesting characteristics of what makes the clusters different from each other. Comparison can be done using both the used bi-grams for the vectors as well as the individual system calls among clusters. Examples of these interesting characteristics are system calls that appear only in one cluster, or when a certain type of system call such as file operations does not exist in a cluster. Since the total number of grams is too large to analyze manually, we will filter the data by removing the grams that are common to all clusters.



(a) Projection labeled with malware family name. (Legend is different from Figure 5.16a)



(b) Clustered using HDBSCAN with minimal cluster size 5 samples.

Figure 5.15: n-gram cosine similarity for dumped Malpedia-a dataset, project to 2d using T-SNE.

Memory dumped Malpedia dataset

In Figure 5.15 we see the results from the memory dumped version of the same samples. These samples are the exact same samples that were used for the static analysis. HDBScan finds five clusters, we will discuss each of these clusters.

Clusters	1	2	3	4	5
# Bigrams per cluster	348	488	546	511	324
# Bigrams reduced by common bigrams per cluster	260	409	467	476	115
# Bigrams unique to cluster	183	332	390	399	38
# Unigrams per cluster	73	111	117	113	69
# Unigrams reduced by common unigrams per cluster	23	61	67	63	19
# Unigrams unique to cluster	12	0	0	0	0

Table 5.6: *n*-gram statistics of the memory dumped Malpedia dataset per identified cluster.

In Table 5.6 the number of unqiue elements (bigrams and unigrams) for each of the clusters is displayed. As can be seen from this table, the number system calls to compare becomes reasonable to inspect manually for the unigrams with the common unigrams among all clusters removed. The full data that shows for each cluster the unigrams reduced by the common ones is provided in Appendix D. Using this data and the third party information that was previously given in subsection 4.1.2 we will now discuss each of the clusters from Figure 5.15.

- **Cluster 1** - Is a small cluster with 7 samples, containing 2 samples from *Globeimposter*, *Gandcrab* and a single sample from *Emotet*, *Clop* and *Dridex*. Interestingly no file operations except file reading are done by the samples within this cluster. Except from Emotet this cluster this cluster shows similarities with what whas found during the static analysis. Compared to the other clusters this cluster has the least amount of unique system calls.
- **Cluster 2** - Samples *Locky*, *Friedex*, *Icedid*, *Dridex*, it is the only cluster making use of the file buffer flushing api.
- **Cluster 3** - The largest cluster cluster contains 34 samples, from *Dridex*, *Vawtrack* and some *Gandcrab* samples. It is the only cluster using the *NtFsControlFile* system call, which can be used to query a system's device information. It is also the only cluster use the system call for writing a file and delays execution with *NtDelayExecution* system call.
- **Cluster 4** - Small cluster of only 5 samples in total. Two of *Locky*, two of *Dridex*, and a single *Emotet* sample.
- **Cluster 5** - Consists of *Locky*, *Globeimposter*, *Gandcrab*, *Dridex*, *Trickbot*. The first three are ransomeware malware, and the last two have been known to be both be distributed by the same malware.
- **Noise** - Not all samples are clustered, 14 samples are classified as noise. There is no clear pattern in the noise, the included samples originate from multiple malware families.

In Table 5.7 are given the quantitative results, using the author labels provided by Malpedia. Especially the F1 score is particularly low, since we also see low precision and recall.

Purity index	Rand index	Precision	Recall	F1
0.54	0.60	0.35	0.29	0.32

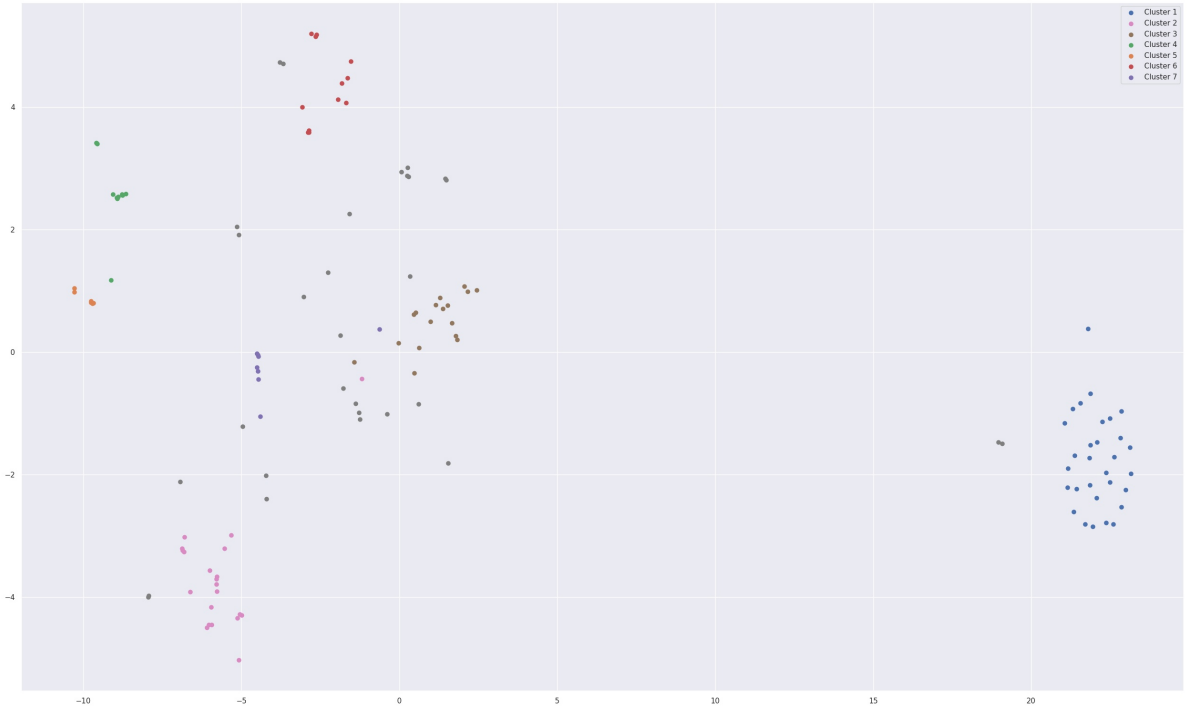
Table 5.7: Quantitative clustering evalution memory dumped malpedia (malpedia-a)

Non memory dumped Malpedia dataset (Malpedia-b)
In this subsection we will show the results of the non-memory dumped Malpedia dataset. Again using cosine similarity on bigram vectors from all the threads identified as originating from the malware, HDBScan distinguishes seven clusters as can be see in Figure 5.16.

Below we list each of the seven clusters.

- **Cluster 1** - *Dridex Vawtrak* and a *Gandcrab* sample. What is interesting is that this cluster is very similar to cluster 3 found in the dumped malware dataset. This is the only cluster using the system call *NtQueryDebugFilterState*, which is function used to detect whether the binary is currently being debugged as an anti-analysis measure.

(a) Projection labeled with malware family name.



(b) Clustered using HDBSCAN with minimal cluster size 5 samples.

Figure 5.16: *n*-gram cosine similarity for non-dumped Malpedia-b dataset, project to 2d using T-SNE.

- **Cluster 2** - *Locky* with *Gandcrab*.
- **Cluster 3** - Contains a relative large amount of different families including: *Vawtrak*, *Locky*, *Clop*, *Dridex*, *Gandcrab*, *Trickbot*. These are the same samples and families as seen in cluster 5 of the clusters from the dumped dataset.

	1	2	3	4	5	6	7
# Bigrams per cluster	375	1190	773	2859	1834	1318	1042
# Bigrams reduced by common bigrams per cluster	167	284	242	1144	887	431	430
# Bigrams unique to cluster	97	214	172	1074	817	361	360
# Unigrams per cluster	74	150	111	280	258	187	150
# Unigrams reduced by common unigrams per cluster	74	150	111	280	258	187	150
# Unigrams unique to cluster	1	5	2	16	7	9	4

Table 5.8: *n*-gram statistics of non memory dumped Malpedia-b dataset per identified cluster.

- **Cluster 4 & 6** - Consists only of *Trickbot* samples.
- **Cluster 5** - *Dridex* only, together with cluster 4, shares a relative large amount of unique system calls as can be seen in Appendix E, compared to the other clusters.
- **Cluster 7** - *Emotet* only.
- **Noise** - Contains more noise than the clustering of the dumped malware samples. Just like that clustering no clear pattern of samples appear in the noise. It are samples, that do not appear as often in the dataset, such as *Dyre*, *Gozi*, but also samples that appear more often like, *Vawtrak*.

In Table 5.9 are given the quantitative results. The precision score is better than for the memory dumped dataset, recall however is quite low.

Purity index	Rand index	Precision	Recall	F1
0.78	0.71	0.62	0.33	0.43

Table 5.9: Quantitative clustering evalution non-memory dumped Malpedia (malpedia-b)

Comparing the results of the datasets
We can compare the clusterings of both datasets. The figures from Figure 5.16 and Figure 5.15 cannot be compared based on the absolute spatial locations of the points directly, since the t-SNE algorithm has a random component. Instead, we compare the relative distance between the points and the formed clusters. The datapoints from the dumped malpedia dataset in Figure 5.15 more scattered over the space than the points of the non-dumped malpedia set Figure 5.16 and the dumped set forms less dense clusters. Even though the samples in the two dataset originate, a sample can come from different stages of the malwares execution and therefore be very different between the datasets. However, we appear to find two clusters that seem very similar in cluster content between the two datasets.

Since we used the same dataset as for the static analysis we can compare the results between this the static and dynamic method. As a reference we can use Figure 4.2 and Figure 4.15 from the static analysis chapter. The clear similarity between *Friedex* and *Dridex*, which we expected based on the literature, and which was also seen in the static analysis does not appear in this dynamic analysis. Instead of *Friedex* we see another similarity in Figure 5.18 that clusters *Dridex* together with *Emotet*, which is not entirely unexpected since *Emotet* has been used to distribute *Dridex*. As for the connections we saw with *Locky* in the static analysis we say evaluate that we also see a connection with *Gandcrab* and maybe with *Flawedammy*, as *Andromut* failed to execute dynamically.

5.6.2. Custom clustering based on frequent set finding

Now we will present the results for the custom clustering method for all datasets. First we will discuss the results on the same dataset used in the static analysis of both the memory dumped samples and the non-memory dumped samples.

The final results can be visualized using a graph in tree structure. The nodes in this tree represent the frequent item sets, and the edges are formed between nodes based on whether a malware sample is contained in sets of adjecent lengths (-1 or $+1$). The tree has as its root an empty set and as we walk down the tree each level of the tree contains the frequent item sets increasing in length. In addition, we can show how strong

the similarity connection is between samples within a set by the level of support the set has. The support of a set decreases as the frequent set is further away from the root. A prerequisite for this visualization to work is that the number of samples that are compared to each other need to be small enough such that the graph does not become too big and therefore becomes infeasible to analyze. Therefore, the minimum support of frequent itemset mining algorithm needs to be set at a sufficient high level.

Besides the raw support, each node in the graph is labeled with the fraction of the original support from the individual samples that is left in the downstream node. It is calculated by taking the individual average support of the samples contained in each cluster and dividing this by the support of the current cluster. The colors are indicicative of this value, with red meaning a high fraction is left with gradual shift to blue meaning the fraction is relatively low.

For each of the datasets we will first provide the full tree, however since these are too large to be readable, the trees are pruned to a minimum level of support in order to keep the tree small enough to visually analyze. This pruning is quite important since it can affect the interpretation of the final clusters.

We will also merge the different samples from the same family together as we tree will become unreadable when we keep the malware samples separate. We argue this is better than taking a single random sample from each family since we saw from previous analysis that depending on the version of the malware it may be more or less similar to other malware.

Malpedia dataset

Figure 5.17 and Figure 5.18 show the results of the custom clustering for two Malpedia datasets. To keep the trees condensed each node only contains an index that references a malware family in Table 5.10. For the Malpedia-b Figure 5.17dataset the minimum support of the frequent item set mining was set at 600, so at least 600 or more sequential patterns need to be shared among one or more malware samples.

Starting on the left, we see on level 2, two clusters containing *Flawedammy* (8), one containing *Emotet* (5) and another with *Flawedgrace* (9). Seeing *Flawedammy* and *Flawedgrace* together is not unexpected given their known common purpose. To see *Emotet*, however, is unexpected. Following these clusters to level 3, we still see relatively high similarity with a fraction of 0.7, indicating the same functionality is shared among all three samples.

To the right on the second level, we see that *Locky* (11) and *Trickbot* (14) share the largest amount of sequences. Following the paths downstream from *Locky* (11), *Trickbot* (14) we see that it is joined by *Dridex* (12) on level 3, with a relatively high fraction of 0.72 for level 3. Substituting any of the samples in this cluster for another sample drops the support significantly.

Now where we see this method of similarity comparison shine is if we look at the similarity *Emotet* (5) *Gandcrab* (6), *Locky* (11), *Dridex* (12), *Trickbot* (14), on level 5. Although the relative support is not particularly high we do see that the absolute support with 830 is high. Adding any extra sample to the cluster such as *Globeimposter* (0) or *IcedId* causes the support to drop significantly. This would not have been caught by a conventional clustering algorithm as it would have either disappeared in smaller clusters, or the dataset would appear as one large cluster.

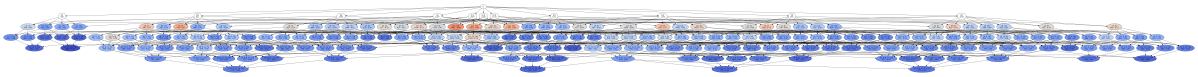


Figure 5.17: Visualization of the found patterns for the non-dumped Malpedia-b set.

In Figure 5.18 the tree of the Malpedia-a malpedia dataset is displayed. This graph is harder to analyze than the graph extracted from the non-memory dumped dataset, as the supports of the different clusters lie much closer to each other. Some samples share almost exactly all of its patterns with another samples, such as *Friedex* (2) and *Lock* (11), only missing two patterns from *Friedex*. Another example is *Vawtrack* (4) with *Dridex* (12), missing only 3 patterns from *Vawtrack*. No clear clusters based on labeled authors can be observed, however in general we see that the ransomeware apears to cluster together in this dataset. These are *Globeimposter* (0), *Friedex* (2), *Gandcrab* (6), *Clop* (7), *Locky* (11).

Similarity between the two sets is not necessarily something we would expect since both parts of the malware may be very different. The non-memory dumped version may only do the unpacking, and could be broken after that stage because of a broken command and control server.

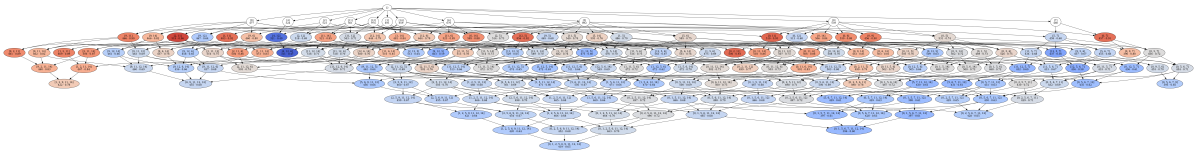


Figure 5.18: Visualization of the found patterns for the dumped Malpedia-a set.

Index	Author	Malware
0	-	Globeimposter
1	-	Gozi
2	Indrik-spider	Friedex
3	Lunar-spider	Icedid
4	Lunar-spider	Vawtrak
5	Mummy-spider, mealybug	Emotet
6	Pinchy-spider	Gandcrab
7	ta505	Clop
8	ta505	Flawedammy
9	ta505	Flawedgrace
10	ta505	Get2
11	ta505	Locky
12	ta505+indrik-spider	Dridex
13	Wizard-spider	Dyre
14	Wizard-spider	Trickbot

Table 5.10: Index reference to malware name

Any.run dataset

Given the results of the Any.run dataset in Figure 5.19, we see only two clear patterns. This is the pattern, Emotet, Ramnit, Dridex. Since most of these malware are much newer than the malware used in the previous dataset, not as much is known about them yet.

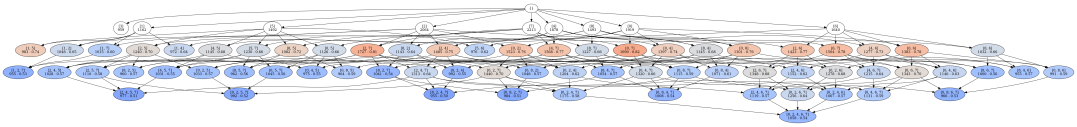


Figure 5.19: Visualization of the found patterns for the any.run dataset. Pruned at a maximum of 900 support.

Index	Author	Malware
0	ta505+indrik-spider	Dridex
1	Wizard-spider	Dyre
2	Mummy-spider, mealybug	Emotet
3	-	Gozi
4	-	Hawkeye
5	Lunar-spider	Icedid
6	-	Qbot
7	-	Ramnit
8	Wizard-spider	Trickbot

Table 5.11: Index reference to malware name

5.7. Discussion and conclusion

The cosine similarity over n-gram vectors of the system call sequences show Comparing the difference in system calls among clusters proofs that certain type of malwares are found, as not all behevior is present in all clusters. The quality of the clusters could be improved, since the clusters are not as clean as we saw with the static analysis. We argue that the granularity of n-gram vectors of the full system call sequences among malware is too large. As can be seen from the dataset distribution of Figure 5.8,Figure 5.9, and Figure 5.10, a large part of the data can be described by only a small set of possible n-grams, which consist for a majority part of memory operations.

We used the same quantitative clustering evaluation metrics as for the static clustering. The results of the dynamic analysis of the two datasets are comparable, and both score worse on all tests than the results from the static datasets. We note again that the labels provided by Malpedia are not to be trusted blindly, as it is even possible authors with different labels are actually the same author. In addition, *globeimposter* and *gozi* malware do not have any author attached.

It is even harder to evaluate the results of the custom clustering algorithm for the dynamic similarity analysis. In order to make the results visible of this unconventional cluster method, the tree structure was introduced, however this method is not optimal as it results in a large amount of clusters and therefore becomes harder to analyze.

We do not nescecarily see clusters based on author. From the memory dumped dataset from Malpedia, the closes we see to similarity is based on functionality. The malware that shows similarity is the malware from the ransomware type.

In order to further analyze the difference between the different clusters on systemcall level, the best we can do is find the differences in sequential patterns between the different clusters. However, this results in a long list of patterns of system calls from which still nothing can be concluded. Unlike the dataset used for the static analysis this dynamic dataset is simply too limited to do deeper analysis on what type of matches were found. Acquiring the dynamic dataset is much more difficult than acquiring data statically, since a separate system for capturing data is necessary. Besides the data acquiring system, there is a dependence on the malware actually working as the author of the malware intedended in to work on a regular system such that data can be captured.

We argued the reasons for developing a new method over choosing one of the conventional methods for finding differences between sequential data. We are still convinced system call sequences are a good source of data for doing similarity analysis, but additional data besides the raw system calls is necessary. However, we cannot really quantify how well the method works given the difficult dataset and all the possible problems with dynamic analysis previously discussed in section 5.1.

5.8. Limitations

We will discuss the limitations on two fields. First we will discuss the limitations of the dataset that was generated. Second we will discuss the limitations of technique used to analyze the malware system call traces.

5.8.1. Dataset

There are four limitations we found that have to do with our captured dataset.

The first is that the time of the capture analysis is currently limited to a maximum of 2 minutes. Giving the malware more time to execute may produce more relevant malware data. The malware may continue to run in the background to wait for instructions from a command and control server. Another possibility is that the malware stalls on purpose to avoid being analyzed on high throughput malware analysis systems.

The second is that the smallest categorization of the data is on thread level. We do not have any information about the code structure within a thread such basic block boundaries or even function boundaries. This means we cannot easily cut the system call traces to smaller pieces as was possible using static analysis, because we do not know where to cut them. We want to do this such that these individual pieces can be attributed to different authors.

The third issue is that we do not have the process name of all processes. The process name is unfortunately only logged if the thread creator reports to the *PsSetCreateProcessNotifyRoutine* windows functionality. About a third of the process IDs present in the system call log do not appear in list of created processes. These processes can therefore not be filtered on using their process name.

The fourth and fifth issue are not directly caused by this dataset, but by the problem that the malware process activities can be scattered through the system traces. Since it has been identified that malware may

use techniques that make following the full execution of the malware from the starting point of execution is hard by injecting itself into other benign processes (subsection 3.1.4). Previous literature did not yet give this issue any attention while searching for similarities among malware, since their goal was to identify the malware family. They did not have to gather all malware behavior, the behavior of the main malware process appeared to be enough. However, since our goal is to find similarities among all malware, including among malware families, we want to gather as much information from the malware execution as possible.

All recorded system call traces were recorded on a system that does not have any applications installed besides the default applications that come with Windows 10. Some malware might do process injection in non-windows applications, to increase the detection surface of possible process injections. Therefore, it would be good to prepare the test machine with other popular applications, to try and catch these injections.

5.8.2. Technique

Regarding the frequent itemset clustering technique, the solution may suffer from the problem of sample length discrepancy. Malware samples that produce a higher number of different system call sequences are more likely to appear in the final clustering analysis than samples with only a few frequent sequential system call patterns. However, we argue it is not as big of a problem is it is for full document similarity comparison. If a malware sample only has a few unique frequent sequential patterns and all of these samples match with another sample, we cannot yet conclude these samples are likely from the same author. This is because unless these patterns are very long they are likely not unique enough to characterize as work from a specific author. This brings us to another limitation as to the longer the possible sequential patterns are the more expensive the sequential pattern-finding becomes. In our research, we had to limit the maximal length of the patterns to 5, since the current state of the art sequential pattern finding algorithms could not handle higher numbers as the algorithms did not complete on our dataset of relatively long sequences.

As often the case with clustering it is hard to evaluate the quality of the results. There is no previous work that does a similar analysis and there are no ground truth labels to validate the found similarities are the similarities we are actually looking for. However, we are able to compare the results with the results that were achieved from the static analysis though, because we used the same dataset for both analyses.

An issue in general with frequent pattern mining is that the output data is not necessarily smaller than the input set [14]. Even when using maximal patterns the output is still too large to draw useful conclusions from directly.

Regarding the visualization of the results, using the graph-based approach is not perfect since it requires pruning to make the graph interpretable. The problem with pruning based on support is that it may seem like the bottom nodes in the graph are reasonable clusters, while in fact, the pruning can cut off other clusters that lie just below the set pruning threshold.

5.9. Summary

First, we explained the problem we are trying to solve and why we cannot do naive clustering or methods that find differences instead of similarities in sequences. Then we proceeded to explain how the data gathering took place, a solution on top of Cuckoo sandbox was built in order to gather system-wide system calls. Unfortunately, the gathered data did not provide sufficient information to find all process injections as there is too much arbitrary background noise in most processes to differentiate between benign threads and maliciously injected threads. We discussed multiple options to extract useful features from the system call traces, we decided on a custom approach that involving frequent pattern extraction. To represent the results we used a custom graph representation to make it possible to visualize overlapping clusters the clustering method makes possible.

6

Conclusion and future work

6.1. Conclusion

In this thesis, we wanted to find an answer on how to attribute different malware to the same author. According to previous research on author attribution coding style should remain after compilation. We however, have strong indications these past studies are flawed, because of dataset issues regarding copied source code among an author their programs. Therefore, we argue that the conclusion that coding style survives compilation is not justified. With coding style not surviving compilation, we therefore redefined similarity with the goal of doing author attribution that focusses on the functionality of the binary programs instead of style.

Previous clustering and classification research regarding malware is concerned with whether malware samples belong to the same family or classifying whether a binary sample is malicious or not. This research took a different approach, we wanted to find out how to cluster malware from different malware families together based on the author who wrote the malware. This required us to get a good understanding of the difference between finding similarities and finding differences in malware. We approached the problem using both static and dynamic analysis techniques, both requiring different feature extraction techniques and slightly different clustering methods. Evaluation of the results was done by looking at what is generally known about the malware families and their authors, deeper inspection of some of the clusters through case studies, and quantitative cluster evaluation metrics.

In this section, we will first discuss the answer to our sub-research questions. The answers together will eventually form the answer to our main research question.

6.2. Research questions

The answers to the sub-questions and main research question are given below.

Sub-question 1

Is it possible to find similarities among malware binaries using an authors coding style information?

Answer 1: We argue that is not possible to find similarities in the form of code style, there is simply not enough information about the coding style of the author left after compilation. Even if we assume variables such as the compiler, library versions, and an authors style remains the same over time, different source code with the same function can simply compile to the same machine code. Previous work declared it as possible to learn an author their style using mostly n-gram vectors and random forest and SVM machine learning algorithms. However, we claim that instead of learning the author their style, the author their use of certain custom general functions copied among an author their different programs was learned. We supported this claim by comparing the literal source code similarity on the dataset of the previous work, this resulted in high similarity, too high to assert only the author their programming style is learned.

Sub-question 2

What features that help attribute authorship can be extracted from recent malware binaries using static analysis?

Answer 2: Malware samples are almost always packed to make static analysis harder, without unpacking static analysis will not yield any useful results. Therefore, we can only answer this question given that the malware is in an unpacked state acquired by manual or automatic memory dumps. Realizing that the previous work on malware attribution has not been measuring binary style but binary similarity. We see that n-gram vectors from the assembly of decompiled binaries can be a good measure for finding similarities and therefore cluster authorship.

Sub-question 3

How can similarities among malware binaries be found using features statically derived from recent malware binaries?

Answer 3: The simplest method given the decision to use n-gram vectors from decompiled binaries is to create a single vector from the full binary. We have shown that this does not result in a desirable result, since we see no clear clusters, while we expected them to be there. We argue that this is because the granularity of creating a single vector per binary is too large. Therefore, we proposed to split the binaries in smaller parts, and compare these parts individually among binaries. The number of parts a malware sample can be disassembled can however be quite large, this results in a lot of data points. Merging all these points into a single large dataset and running an arbitrary clustering algorithm on it will not yield useful results. This is because of the diversity of the data and the number of points, therefore there are no clear cluster boundaries. Besides, the clusters that cover the similar instruction sets are relatively small and therefore if they exist disappear in the noise of non-similar code. To solve this we proposed to do the clustering pairwise and focus only on the points that lie very close to each other. By using a malware dataset of limited samples that already have suspicions of code sharing, we visualized how well the method works in practice. The found clusters can be explained from public reports of the clustered malware, which already provides some evidence that the method works. In addition, a couple of case studies were done to further prove the usefulness of the developed method.

Sub-question 4

What features that help attribute authorship can be extracted from recent malware binaries using dynamic analysis?

Answer 4: We argue that system calls are a logical solution to find similarities among malware samples. Firstly, because it provides enough unique information to a specific malware sample to find distinguishable information to where malware samples show similarities. Secondly, since system calls can be gathered from kernel space, malware that runs on userspace is unlikely to detect that it is being analyzed. Thirdly, system calls can be captured system-wide and are not dependent on a hook into a specific process, in theory, this should help capture all malware activity instead of just the initial malware process.

Sub-question 5

How can similarities among malware binaries be found using features dynamically derived from recent malware binaries?

Answer 5: We proposed two clustering methods for similarity analysis on malware using system calls, the first attempted using n-grams for the clustering, which is a well-researched method used for natural text clustering. We have assessed the made clusters both visually and quantitatively using the available author labels. However, we argued this method is too coarse-grained, therefore we developed a second method that has two advantages. Firstly it allows us to find the similarities on a finer-grained level, and secondly, the method is able to cluster a single sample into multiple different clusters. The evaluation of this second method is unfortunately slightly problematic, it showed that we are able to find similarities, some matched the expectation, but others did not unfortunately, because of limitations with the dataset we were unable to evaluate the quality of these similarities further.

Main research question

How can we automatically attribute authorship among recent malware binaries using either static or dynamic analysis?

Answer main: The problem of finding similarities instead of finding differences among malware samples is not a well-researched topic yet, as most research focussed on finding differences instead of similarities. We engineered multiple methods to try and figure out the best method to find similarities among malware samples, both statically and dynamically. For the static method, we were able to find some new probable similarities and confirm some other interesting similarities. Regarding the dynamic technique for finding similarities, we are less convinced of it being the best possible method. Dynamic analysis is more difficult than static analysis, because of the additional data gathering system. Although the currently developed method in combination with the used dataset could be improved, we are convinced the reasoning for choosing such a method is valid.

All methods resulted in slightly different clusters for the "same"[1] dataset, since we do not have a ground truth value for which parts of each malware are copied between each other, the clusters cannot be evaluated on precision.

6.3. Limitations

In the process of solving the problem of finding similarities among malware families, we have shown why a naive solution does not work as well as we desired, and therefore proceeded to propose a newly engineered solution for both static malware analysis and dynamic malware analysis. For both solutions we had to acquire all data ourselves, given that doing this for malware is quite difficult, chapter 3 is dedicated to the difficulties of doing malware analysis and what to take into account when doing data science on malware. For static analysis capturing data is easier than for dynamic analysis, because for dynamic analysis more practical problems arose and it required a custom kernel driver to log the system calls of the operating system while a malware sample executes.

This research proves that it is possible to do similarity clustering on smaller granularity than on full binaries. We show that we can get the data to cluster for both the static and dynamic analysis that go beyond clustering malware of the same family. Finding both known similarities, but also probable new similarities among malware families. We used publicly available malware reports, case studies on groups of malware, and quantitative clustering metrics for evaluation. However, evaluating clustering remains hard, evaluating clustering in general is already difficult, and being a cybersecurity-related clusters makes it even harder.

Clustering is hard in general, because of its unsupervised nature. The data used for clustering does not necessarily need to have a ground truth value which we can evaluate against, as is the case with the datasets used in this thesis.

When clustering we have to deal with parameters such as the number of clusters and the problem of what to do with outliers. The used clustering algorithm during this thesis (HDBSCAN) deals with both these parameters, however even with this algorithm it is possible to come to different clusters by changing the parameter of the minimum number of samples per cluster.

Data analysis in cybersecurity is a constantly changing field, because of technological progress. This causes uncertainty with the dataset, since malware keeps evolving to use new techniques to actively counter

[1] dynamic or static versions

analysis, some of these techniques we discussed in chapter 3. It remains difficult to determine if all malware data is captured during the analysis. Another cause of uncertainty has to do with the source of the data, malware is not easy to gather as it has to be caught in the wild. The information about malware and malware binaries available in cybersecurity repositories are gathered by companies and individual researchers without academic review. Although measures are taken to increase the trust in the malware data repositories such as by using a peer review system to add new information, incomplete or false information can still slip in.

In summary, a conclusion regarding the results of a clustering boils down to something that appears to make sense when observed visually or on a case by case basis, however it remains difficult to express the quality of the clustering quantitatively.

6.4. Future work

In this section we will list the possibilities with more work and better data, in addition we will list practical improvements to some of the discussed techniques.

Improved data

We believe this research could benefit from better tagged data. Although the main source of data that was used during this research (Malpedia) already lists some authors for Malware there are a couple of shortcomings with these author tags. First, a single malware sample can be tagged with multiple author names if there exists doubt on who the actual author is of the malware. The second issue is that both malware and author names are not standardized, every anti-virus vendor or researcher can create their own names. This practice causes a single malware sample to be tagged with multiple names and multiple author names, which makes it hard to use the tags as ground truth for author attribution.

Ideally, a dataset for author attribution does not tag malware by arbitrarily named author tags. Instead, malware samples should be tagged by the individual authors their real name, such that even when the composition of a group changes the group can be tracked. These author tags can be established through law enforcement investigation.

Use of function context in function matching

Currently, no structural information outside of the function is used for finding function similarity. To improve the quality of matching similar functions, the greater context in which a function appears could be used as a heuristic. An example of such a feature is to weight the match by whether the found matching functions have the same calling and returning functions, as was illustrated during the analysis in subsection 4.4.4.

Better post processing for frequent pattern clustering

The postprocessing after finding the patterns of malware samples that have mutual system call sequences could be improved. Currently, we show the results as a graph, however this does not scale to larger datasets. Additionally, such evaluation requires a manual inspection to interpret the results which makes automation of the solution for large scale datasets an issue.

Improved system call tracing

The current system call tracing solution that is used for the dynamic analysis only keeps track of the process and thread ID the system call appeared in. Although this makes the dataset very simple, it requires more post processing work to make intelligent splits to lower the granularity in which similarities can be found. This process could be simplified by keeping track the stack such that splits can be made based on the current stackframe. Another option that would provide more information about the system call would be to log the arguments made to each system call.

Improved tracing of process injection

A pure system call based approach turned out to be insufficient to find the malware threads among all other threads that run on a system. Even though the analysis system is mostly deterministic, external influences can cause non-deterministic behavior among analysis runs. Because of this, it is not possible to distill just the malware threads from the system. Additional data is necessary such that the malware can be traced on the system. If the arguments of each system call could be captured the flow of data between system calls could potentially be followed and the inter-thread interaction exposed more easily. Another method to extract the malware traces could be in the form of anomaly detection. This method does not require other additional data, however for this approach are large amounts of benign data necessary.

Syscall2Vec

Improve the Word2Vec/Syscall2Vec model, the model could probably be improved such that it provides a better model. In this research, we experimented with it shortly, but it did not provide the desired results in the limited available time. In case larger datasets will be used in the future, it may also become more relevant to reduce the amount of data further, and therefore worth the additional effort. Although using Word2Vec appears like an interesting application for system calls we have not found any work previous work that does system call analysis and makes use of this or any other state of the art word embedding techniques.

A

Google Code Jam Dataset - A possible issue

It is curious that at least for the classification task the Google Code Jam (GCJ) dataset performs so much better than the Malpedia dataset according to our results. Therefore we investigated what the cause of this behavior could be.

The problem we identified has to do with the used GCJ dataset itself. Theoretically, this dataset is optimal for the task, each programmer in this dataset solves the same programming problem, therefore, we avoid learning the program behavior instead of the programmer's style. However, a major issue with this dataset not addressed in previous literature can be identified. Different programs from the same programmer have a high likelihood of containing duplicate code. For instance, if a programmer wants to use some big-integer implementation, but this is not available in the standard library they might copy and paste this implementation from another source into his submission file. This same snippet of code is then duplicated in each submission again. The programmer can then be identified purely on the usage of this snippet instead of his programming style. To confirm this behavior the literal similarity of the source code is analyzed using `SSDeep`[1]

SSDeep works by creating a so-called fuzzy hash for an input and comparing this hash to other hashes to generate a similarity score. A hash can efficiently be compared to another hash, much more efficient than when a full file comparison has to be done. SSDeep is a popular tool to do this to compare binaries. The computed hash is called fuzzy because a similarity match is independent of the location where it appears in the two files. Besides the hash, SSDeep generates a score ranging from 0.0 (no match) to 1.0 (exact match).

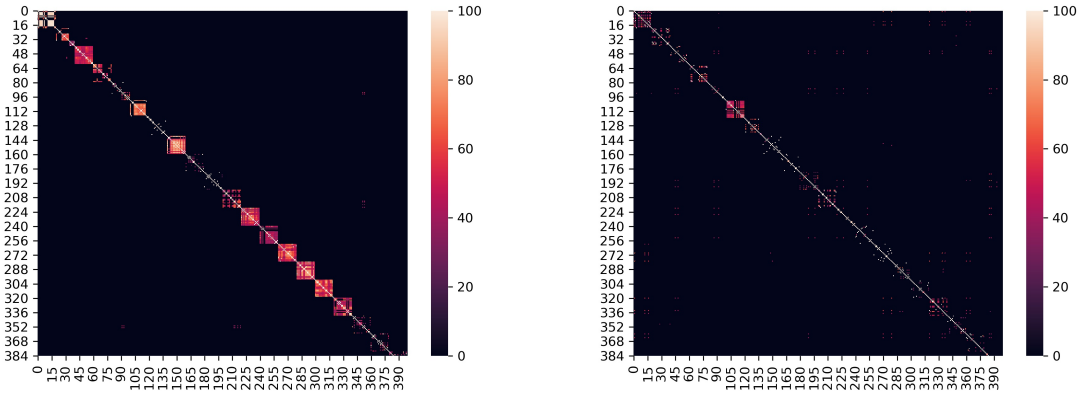


Figure A.1: Heatmaps displaying SSDeep scores of 58 participants with each 20 submissions, the left figure shows the source code, right figure the compiled version of the source.

As can be seen from this example, even though we are looking for literal similarities the source code already show very large similarity scores. Therefore, we can no longer distinguish between the coding style of a

[1] `https://ssdeep-project.github.io/ssdeep/index.html`

programmer and simple code re-use.

B

Table comparison *Friedex* versions

Comparison table of different *Friedex* versions to *Dridex*, from which the function offsets are used.

Dridex	friedex__2017-06-08	friedex__2018-01-26	friedex__2019-03-12	Dridex	friedex__2017-06-08	friedex__2018-01-26	friedex__2019-03-12
0x10025fcb	x	x	x	0x10026f59	x	x	x
0x10027e93	x	x	x	0x10025f3a	x	x	
0x10024bbd	x			0x10027908	x		
0x10028221	x	x		0x10026495	x	x	x
0x100283ae	x			0x10027719	x	x	x
0x1002826b	x	x		0x1002695f	x	x	x
0x10001c00	x	x		0x1002bacc	x	x	x
0x10020776	x	x	x	0x100279a8	x	x	x
0x100287a3	x	x		0x100279d3	x	x	x
0x10025753	x	x	x	0x10024f17	x	x	
0x10026232	x	x		0x10026755	x	x	x
0x10025abf	x	x		0x100272fe	x	x	x
0x1002b860	x	x		0x10026b21	x	x	
0x1002b8db	x	x		0x100163bd	x	x	x
0x1002b302	x	x	x	0x1002ca51	x		
0x1002b274	x	x	x	0x1002c673	x	x	
0x1002bc0e	x	x		0x1002baa5	x	x	x
0x10026035	x	x		0x100147ac	x	x	x
0x1002cc64	x	x	x	0x1002c575	x	x	
0x10025701	x	x	x	0x1002c5c5	x	x	
0x10025c38	x	x	x	0x1002929b	x	x	x
0x1002ccf5	x	x		0x10026bfe	x	x	x
0x10025cf8	x	x	x	0x1002c7e1	x	x	x
0x10026544	x	x	x	0x1002c7a0	x	x	
0x10025e25	x	x	x	0x10026c8a	x	x	x
0x100274b8	x	x	x	0x10026e9d	x	x	x
0x10027257	x	x	x	0x10015cc9	x	x	x
0x10025e71	x	x	x	0x1002c33f	x	x	
0x1002727c	x	x	x	0x1002c7ef	x	x	x
0x10026590	x	x	x	0x10027018	x	x	x
0x10026122	x	x		0x10026a63	x	x	x
0x10025927	x	x	x	0x10026a00	x	x	x
0x10020764	x	x	x	0x100279c5	x	x	
0x10028725	x	x	x	0x1002747c	x	x	x
0x10029e97	x			0x1002744d	x	x	x
0x10028b67	x	x		0x10026684	x	x	
0x10025efd	x	x		0x1001b1ab	x	x	x
0x10026324	x	x	x	0x1002b34f	x	x	x
0x10026e15	x	x	x	0x10026b03	x	x	x
0x10028339	x	x		0x100271ad	x	x	x
0x10025649	x			0x10027122	x	x	x
0x1002870d	x	x	x	0x10026949	x		
0x1002d179	x	x	x	0x10027650	x	x	x
0x1002b1d7	x	x	x	0x1002bbc3	x	x	x
0x1002b27e	x			0x10029614	x	x	x
0x10020740	x	x	x	0x100259b0	x	x	x
0x10026ac2	x	x		0x100257b5	x	x	
0x10020752	x	x	x	0x10025860	x	x	
0x1002758e	x	x	x	0x10025834	x	x	
0x10026763	x	x	x	0x100275ea	x	x	x
0x10025deb	x	x	x	0x10027a24	x	x	
0x10026e5b	x	x	x	0x10027a06	x	x	x
0x100275b5	x	x	x	0x100280a4	x		
0x100275c3	x	x	x	0x10027de0	x	x	x
0x100273cb	x	x	x	0x1002816c	x	x	x
0x100267bc	x	x	x	0x10029d9b	x	x	x
0x10025ebe	x	x	x	0x100298a3	x	x	x
0x10027a9c	x	x	x	0x10029bf2	x	x	
0x10025fdf	x			0x10029b84	x	x	x
0x1002e2ce	x	x		0x10029027	x	x	x
0x1002d193	x			0x100286de	x	x	x
0x1002d29e	x			0x100279e9	x	x	x
0x1002705a	x	x	x	0x1002c7c0	x		
0x100268ff	x	x	x	0x1002cc7e	x	x	x
0x10026ad0	x	x	x	0x1002d8eb	x	x	
0x10026815	x	x	x	0x1003099a	x	x	
0x100272b2	x	x	x	0x100423f1	x		x
0x10026002	x			0x10028f79		x	x
0x100265dd	x	x		0x10027b36		x	
0x1002c374	x	x		0x10028119		x	
0x1002be2a	x	x		0x100281bc		x	
0x1002779d	x	x	x	0x100282aa		x	
0x10026747	x	x	x	0x10028801		x	
0x10027538	x	x	x	0x10026d8c		x	x

Dridex	friedex__2017-06-08	friedex__2018-01-26	friedex__2019-03-12
0x1002ec6e		x	x
0x100258e4		x	
0x10009485		x	x
0x10029e1d		x	x
0x1000ab9d		x	x
0x1002b212		x	x
0x1002bfff		x	
0x1002ba8b		x	
0x100258f5		x	x
0x10027b22		x	x
0x100274d0		x	x
0x1002c746		x	
0x1002c85a		x	x
0x1002ca84		x	
0x10026739		x	x
0x1002c715		x	
0x100256e5		x	
0x10027e7f		x	
0x10028260		x	
0x10003a69		x	x
0x1002d11b		x	
0x100256a3		x	
0x10025676		x	
0x1002b685		x	
0x1002b768		x	
0x1002c6e9		x	x
0x1002cbaf		x	x
0x10025f89		x	x
0x1002b8c6		x	x
0x1002b91a		x	x
0x1002518e		x	
0x1002e33b		x	
0x10025412		x	x
0x1002ff25		x	x
0x1002fd19		x	
0x1002ff4a		x	x
0x1000ae8b		x	x
0x1000aed9		x	x
0x1002b9e8		x	x
0x1002c3bd		x	
0x1002590e		x	x
0x10025ff1		x	x
0x1002f4ef		x	x
0x1002b5ef		x	
0x1002e574		x	
0x1002b823		x	x
0x1002f721		x	x
0x1002d247		x	
0x10025737		x	
0x1002b931		x	
0x10030938		x	
0x10030b4f		x	
0x100309e1		x	x
0x10030c4e		x	
0x100283ed		x	x
0x100128e9		x	x
0x10029f34		x	x
0x1002cc05		x	
0x1002b98c		x	x
0x10025b2a		x	
0x10013184		x	x
0x1001312f		x	
0x1002ba44		x	x

Dridex	friedex__2017-06-08	friedex__2018-01-26	friedex__2019-03-12
0x10014520		x	
0x10014542		x	
0x100292c2		x	x
0x10028535		x	x
0x1002ba16		x	x
0x10026f9d		x	x
0x10030d59		x	
0x10030b21		x	
0x10030a61		x	x
0x1002baf3		x	x
0x100324c5		x	
0x10032355		x	
0x100323ba		x	
0x1001e95b		x	
0x10031c81		x	
0x10031911		x	
0x1001e9cf		x	
0x1001e98e		x	
0x1002549d		x	x
0x1002b411		x	x
0x10030ab7		x	x
0x10025528		x	x
0x1002818d		x	
0x10028d00		x	x
0x1002e484		x	
0x100255d2		x	x
0x1002a572		x	
0x1002bd5d		x	
0x1002c7fd		x	x
0x1002e463		x	x
0x1002e1fe		x	
0x1002e314		x	x
0x1002f959		x	
0x1002f8aa		x	
0x1002fa51		x	x
0x1002fa2c		x	x
0x1002e425		x	x
0x10029198		x	x
0x10029227		x	x
0x10030fd9		x	x
0x10030fec		x	x
0x10031b67		x	
0x1003237c		x	
0x10031c1a		x	
0x10031869		x	
0x10031b95		x	
0x10025165			x
0x1002aeeb			x
0x100071ec			x
0x10026d18			x
0x10025c2a			x
0x10026ee1			x
0x100033f1			x
0x100038ca			x
0x10003718			x
0x10007177			x
0x1002561a			x
0x1002e3f4			x
0x1002650c			x
0x1001689e			x
0x1002865e			x
0x1002f7a3			x

C

Table comparison *TA505* comparison

Comparison of TA505 similarities found during static analysis, all four samples are compared to the *Locky* malware family.

Locky	Andromut	Clop	Flawedgrace	Flawedammyy	Globeimposter
0x00402bc9	x				
0x0040cd69	x			x	x
0x0040e7c0	x	x	x	x	x
0x0040e1b1	x			x	x
0x0040e7c9	x	x	x	x	x
0x0040db7d	x	x	x	x	x
0x0040e3d5	x	x	x	x	x
0x0040ce4d	x				
0x0040e500	x			x	x
0x0040d5fd	x				
0x0040d7d0	x				
0x0040de86	x				
0x0040deed	x	x	x	x	x
0x0040e383	x	x	x	x	x
0x0040e356	x			x	x
0x0040e3e8	x	x	x	x	x
0x004108e2	x			x	x
0x004110f6	x			x	x
0x0040f7a1	x	x	x	x	x
0x0040f7aa	x	x	x	x	x
0x0040fd42	x				
0x0040ea2c	x	x	x	x	x
0x0040ea3b	x	x	x	x	x
0x004104c0	x		x	x	x
0x00410fc5	x			x	x
0x004111d0	x	x	x	x	x
0x0040e805	x			x	x
0x00410882	x	x	x	x	x
0x004108b2	x	x	x	x	x
0x004108cb	x	x	x	x	x
0x00410790	x	x	x	x	x
0x00410899	x	x	x	x	x
0x0040ef1d	x	x	x	x	x
0x0040f2ec	x	x	x	x	x
0x004119c0	x	x	x	x	x
0x00411957	x	x	x	x	x
0x004115e0	x			x	x
0x0040f5b6	x			x	x
0x0040f960	x	x	x	x	x
0x0040f96c	x	x	x	x	x
0x0040f325	x			x	x
0x0041074a	x	x	x	x	x
0x004121d0	x	x	x	x	x
0x00410e61	x	x		x	x
0x00411ff5	x	x	x	x	x
0x0041289e	x				
0x004110be	x				
0x004138e0	x		x		
0x00412b8c	x			x	x

	Andromut	Clop	Flawedgrace	Flawedammyy	Globeimposter
0x0041495f	x	x	x	x	x
0x00412da7	x				
0x004127da	x		x		
0x004110ab	x				
0x00412a87	x		x		
0x00412ada	x				
0x00413042	x		x		
0x004127a8	x		x		
0x004144ee	x			x	x
0x00413d07	x	x	x	x	x
0x00414940	x			x	x
0x00414860	x		x	x	x
0x0041434e	x			x	x
0x0040cc35	x				
0x0040ce58	x				
0x004124e0	x				
0x0041286b	x		x		
0x00412975	x		x		
0x00412c10	x				
0x004143d6	x	x	x	x	x
0x0040227b	x				
0x0040229b	x				
0x00402531	x			x	x
0x0040254f	x				
0x0040256f	x			x	x
0x00402685	x				
0x00402c05	x		x		
0x00402c20	x				
0x0040a22f	x			x	x
0x00411f90	x	x	x	x	x
0x00412835	x		x		
0x0040cd52		x	x		
0x0040ea16		x	x		
0x0040dde5		x	x		
0x0040ddf8		x	x		
0x0040de03		x	x		
0x0040e1f9		x	x		
0x0040e1ea		x	x		
0x00411107		x	x		
0x0040ea00		x	x		
0x00411f53		x	x		
0x004120a5		x		x	x
0x00410aab		x	x		
0x00413a80		x	x		
0x00413ab9		x	x		
0x00413acc		x	x		
0x00411ec0		x	x		
0x00412380		x	x	x	x
0x0040cb97			x		
0x0040e7d2				x	x

D

Dynamic analysis - cosine similarity - Malpedia dumped dataset

System calls	1	2	3	4	5
NtAlertThreadByThreadId			x		
NtContinue		x	x	x	x
NtCreateFile		x	x	x	x
NtDelayExecution			x		
NtDeviceIoControlFile			x		
NtFsControlFile			x		
NtGetMUIRegistryInfo			x		
NtNotifyChangeKey	x	x	x	x	
NtOpenThreadToken		x	x	x	
NtQueryDebugFilterState	x		x		
NtQueryDirectoryFileEx		x	x	x	x
NtQueryWnfStateData		x	x	x	
NtRaiseHardError	x	x	x		x
NtReadFile	x		x		
NtReleaseWorkerFactoryWorker			x		
NtRequestWaitReplyPort	x	x	x	x	
NtSetInformationFile		x	x	x	x
NtSetWnfProcessNotificationEvent	x	x	x	x	
NtSubscribeWnfStateChange		x	x	x	
NtWaitForAlertByThreadId	x		x		
NtWaitForSingleObject			x		
NtWorkerFactoryWorkerReady			x		
NtWriteFile			x		
NtAlpcConnectPort		x		x	x
NtAlpcSendWaitReceivePort		x		x	x
NtCallbackReturn		x		x	
NtCompareObjects		x			
NtCreateMutant		x		x	
NtCreateSemaphore		x		x	
NtDuplicateToken	x	x		x	
NtEnumerateKey		x		x	x
NtEnumerateValueKey	x	x		x	x
NtIsUILanguageComitted		x		x	x
NtMapViewOfSectionEx	x	x		x	x
NtOpenMutant		x		x	x
NtOpenProcess	x	x		x	x
NtOpenProcessTokenEx	x	x		x	
NtOpenSemaphore	x	x		x	
NtOpenThreadTokenEx	x	x		x	
NtQuerySection	x	x		x	x

System calls	1	2	3	4	5
NtQuerySecurityObject	x	x		x	x
NtQueryWnfStateNameInformation	x	x		x	x
NtReadVirtualMemory	x	x			
NtSetInformationThread		x		x	x
NtSetTimer2		x		x	
NtTraceEvent		x		x	x
NtUnmapViewOfSectionEx	x	x		x	
NtUpdateWnfStateData		x		x	x
NtUserBuildHwndList	x	x		x	
NtUserCallNoParam		x		x	
NtUserCallOneParam	x	x		x	
NtUserCallTwoParam		x		x	
NtUserCreateWindowEx	x	x		x	
NtUserFindExistingCursorIcon		x		x	
NtUserGetAtomName	x	x		x	
NtUserGetClassInfoEx		x		x	
NtUserGetClassName		x		x	
NtUserGetDpiForCurrentProcess		x		x	
NtUserGetGUIThreadInfo	x	x		x	
NtUserGetImeInfoEx		x		x	
NtUserGetObjectInformation	x	x		x	
NtUserGetProcessUIContextInformation	x	x		x	
NtUserGetProp	x	x		x	
NtUserGetWindowCompositionAttribute	x	x		x	
NtUserIsNonClientDpiScalingEnabled	x	x		x	
NtUserIsTopLevelWindow	x	x		x	
NtUserMessageCall	x	x		x	
NtUserRegisterClassExWOW	x	x		x	
NtUserRegisterWindowMessage	x	x		x	
NtUserReleaseDC	x	x		x	
NtUserSetProcessDpiAwarenessContext	x	x		x	
NtUserSetWindowLongPtr	x	x		x	
NtUserSetWindowPos	x	x		x	
NtUserSystemParametersInfo	x	x		x	
NtUserUnregisterClass	x	x		x	
NtUserUpdateInputContext	x	x		x	
NtUserWin32PoolAllocationStats	x	x		x	
NtWaitForMultipleObjects32	x	x		x	
NtWriteVirtualMemory	x	x			

E

Dynamic analysis - cosine similarity - Malpedia non dumped dataset

System calls	1	2	3	4	5	6	7
NtClearEvent	x			x	x	x	
NtDelayExecution	x			x	x	x	
NtDeviceIoControlFile	x			x	x		
NtFsControlFile	x			x		x	
NtGetMUIRegistryInfo	x			x	x		
NtQueryDebugFilterState	x						
NtReadFile	x			x	x	x	
NtReleaseWorkerFactoryWorker	x			x	x	x	
NtSignalAndWaitForSingleObject	x			x		x	
NtWaitForAlertByThreadId	x	x		x	x	x	x
NtWaitForSingleObject	x			x	x	x	
NtWorkerFactoryWorkerReady	x			x	x	x	
NtWriteFile	x			x		x	
NtAccessCheckByType		x		x	x	x	x
NtAlertThreadByThreadId		x		x	x	x	x
NtAlpcAcceptConnectPort		x		x	x	x	x
NtAlpcConnectPort		x	x	x	x	x	x
NtAlpcConnectPortEx		x	x	x	x	x	x
NtAlpcCreatePort		x		x	x	x	x
NtAlpcCreateResourceReserve		x		x	x		
NtAlpcCreateSecurityContext		x		x	x		x
NtAlpcDeleteSecurityContext		x		x	x		
NtAlpcImpersonateClientOfPort		x		x		x	
NtAlpcQueryInformation		x	x	x	x	x	x
NtAlpcQueryInformationMessage		x		x	x		
NtAlpcSendWaitReceivePort		x	x	x	x	x	x
NtAlpcSetInformation		x		x	x	x	x
NtCallbackReturn		x	x	x	x	x	x
NtCancelIoFile		x					
NtCancelTimer2		x	x	x	x	x	
NtCancelWaitCompletionPacket		x		x	x		
NtCompareObjects		x	x	x	x	x	
NtCreateKey		x	x	x	x	x	x
NtCreateMutant		x	x	x	x	x	x
NtCreateSemaphore		x	x	x	x	x	x
NtCreateThreadEx		x	x	x	x	x	x
NtCreateTimer		x		x	x	x	x
NtCreateUserProcess		x		x	x	x	x
NtDuplicateToken		x	x	x	x	x	x
NtEnumerateKey		x	x	x	x	x	x
NtEnumerateValueKey		x	x	x	x	x	x
NtFlushBuffersFile		x					
NtGdiBitBlt		x		x	x	x	
NtGdiCreateBitmap		x		x	x	x	x
NtGdiCreateDIBitmapInternal		x		x	x	x	
NtGdiExtGetObjectW		x		x	x	x	
NtGdiSetDIBitsToDeviceInternal		x		x	x	x	
NtGetCompleteWnfStateSubscription		x		x	x		
NtGetNlsSectionPtr		x					
NtIsUILanguageComitted		x	x	x	x	x	x
NtLockVirtualMemory		x					
NtMapViewOfSectionEx		x	x	x	x	x	x
NtNotifyChangeMultipleKeys		x	x				
NtOpenMutant		x	x	x	x	x	x
NtOpenProcess		x	x	x	x	x	x
NtOpenProcessTokenEx		x		x	x	x	x
NtOpenSemaphore		x	x	x	x	x	x
NtOpenThread		x	x	x	x		x
NtOpenThreadTokenEx		x		x	x	x	x
NtOpenTimer		x					
NtQueryDirectoryFile		x		x			x
NtQueryEvent		x	x	x	x		x
NtQueryFullAttributesFile		x	x	x			x
NtQuerySection		x	x	x	x	x	x
NtQuerySecurityObject		x	x	x	x	x	x
NtQueueApcThread		x		x	x		
NtReadVirtualMemory		x	x	x	x	x	x
NtResumeThread		x	x	x	x	x	x
NtSetInformationObject		x		x	x	x	x
NtSetInformationThread		x	x	x	x	x	x
NtSetInformationToken		x		x			
NtSetTimerEx		x		x	x	x	x
NtSetValueKey		x	x	x	x		x

System calls	1	2	3	4	5	6	7
NtSuspendThread		x		x	x	x	
NtTraceEvent		x		x	x		
NtUnlockVirtualMemory		x		x			
NtUnmapViewOfSectionEx		x	x	x	x	x	x
NtUnsubscribeWnfStateChange		x		x	x		
NtUserBuildHwndList		x		x	x	x	x
NtUserCallHwndLock		x	x				
NtUserCallNoParam		x	x	x	x	x	x
NtUserCallOneParam		x	x	x	x	x	x
NtUserCreateEmptyCursorObject		x		x	x	x	
NtUserCreateWindowEx		x		x	x	x	x
NtUserFindExistingCursorIcon		x	x	x	x	x	x
NtUserGetAtomName		x		x	x	x	x
NtUserGetClassInfoEx		x		x	x	x	x
NtUserGetGUIThreadInfo		x	x	x	x	x	x
NtUserGetKeyboardLayout		x	x	x	x	x	x
NtUserGetObjectInformation		x	x	x	x	x	x
NtUserGetProcessUIContextInformation		x	x	x	x	x	x
NtUserMessageCall		x		x	x	x	x
NtUserRegisterClassExWOW		x	x	x	x	x	x
NtUserRegisterWindowMessage		x	x	x	x	x	x
NtUserReleaseDC		x	x	x	x	x	x
NtUserSetCursorIconData		x		x	x	x	
NtUserSetWindowLongPtr		x		x	x	x	x
NtUserSystemParametersInfo		x	x	x	x	x	x
NtUserWin32PoolAllocationStats		x	x	x	x	x	x
NtWaitForMultipleObjects32		x		x	x	x	x
NtWriteVirtualMemory		x		x	x	x	x
NtUserCallHwnd			x				x
NtUserCallHwndParamLock			x				
NtUserCallHwndParamLockSafe			x			x	
NtUserCallTwoParam			x	x	x	x	x
NtUserGetCursor			x				
NtUserThunkedMenuItemInfo			x	x	x	x	
NtUserWindowFromPoint			x	x		x	
NtAddAtomEx				x	x		
NtAdjustPrivilegesToken				x	x		
NtAllocateLocallyUniqueId				x			
NtAlpcCreatePortSection				x			x
NtAlpcCreateSectionView				x			x
NtAlpcDeletePortSection				x			x
NtAlpcDeleteSectionView				x			x
NtCancelIoFileEx				x			
NtCreateNamedPipeFile				x			
NtCreatePrivateNamespace				x			
NtFlushProcessWriteBuffers				x			
NtGdiAnyLinkedFonts				x	x		
NtGdiCreateCompatibleBitmap				x	x		
NtGdiCreateCompatibleDC				x	x		
NtGdiCreateRectRgn				x	x		
NtGdiCreateRoundRectRgn				x			
NtGdiCreateSolidBrush				x	x		
NtGdiDeleteObjectApp				x	x		
NtGdiDoPalette				x	x	x	
NtGdiDrawStream				x	x		
NtGdiExtTextOutW				x	x		
NtGdiFlush				x	x		
NtGdiFontIsLinked				x	x		
NtGdiGetCharABCWidthsW				x	x	x	
NtGdiGetDCDword				x	x		
NtGdiGetDCObject				x	x		
NtGdiGetDCforBitmap				x	x		
NtGdiGetDIBitsInternal				x	x	x	
NtGdiGetDeviceCaps				x	x		
NtGdiGetEntry				x	x		
NtGdiGetFontData				x	x	x	
NtGdiGetGlyphIndicesW				x	x	x	
NtGdiGetOutlineTextMetricsInternalW				x	x	x	
NtGdiGetRandomRgn				x	x		
NtGdiGetRealizationInfo				x	x		
NtGdiGetTextExtentExW				x		x	
NtGdiGetTextFaceW				x	x		
NtGdiGetTextMetricsW				x	x		

System calls	1	2	3	4	5	6	7
NtGdiGetWidthTable				x	x	x	
NtGdiHfontCreate				x	x	x	
NtGdiIntersectClipRect				x	x	x	
NtGdiPatBlt				x	x		
NtGdiPolyPatBlt				x	x		
NtGdiPolyTextOutW				x	x		
NtGdiQueryFontAssocInfo				x	x		
NtGdiRestoreDC				x	x		
NtGdiSaveDC				x	x		
NtGdiSelectBitmap				x	x		
NtGdiSetLayout				x	x		
NtGdiStretchDIBitsInternal				x	x	x	
NtGetContextThread				x			
NtGetWriteWatch				x			
NtImpersonateAnonymousToken				x			
NtQueryDefaultLocale				x	x		
NtReleaseMutant				x	x		
NtReleaseSemaphore				x	x		
NtRemoveIoCompletion				x			
NtResetWriteWatch				x			
NtTerminateProcess				x	x		
NtTerminateThread				x	x		
NtUpdateWnfStateData				x			x
NtUserAssociateInputContext				x		x	
NtUserBeginPaint				x	x		
NtUserCalcMenuBar				x	x	x	
NtUserCallHwndParam				x		x	
NtUserConsoleControl				x	x		
NtUserCreateInputContext				x			
NtUserDestroyWindow				x			
NtUserDispatchMessage				x	x		
NtUserDrawIconEx				x	x		
NtUserEnableChildWindowDpiMessage				x	x		
NtUserEnableScrollBar				x	x		
NtUserEndPaint				x	x		
NtUserGetAncestor				x	x		
NtUserGetCaretBlinkTime				x	x		
NtUserGetClassName				x	x	x	x
NtUserGetDC				x	x		
NtUserGetDCEx				x	x		
NtUserGetDpiForCurrentProcess				x	x		
NtUserGetDpiForMonitor				x	x		
NtUserGetIconInfo				x	x	x	
NtUserGetIconSize				x	x		
NtUserGetImeInfoEx				x	x	x	x
NtUserGetKeyboardLayoutList				x		x	
NtUserGetMessage				x	x		x
NtUserGetProcessWindowStation				x	x		
NtUserGetProp				x	x	x	x
NtUserGetScrollBarInfo				x	x		
NtUserGetSystemMenu				x	x		
NtUserGetThreadDesktop				x	x		
NtUserGetThreadState				x	x		
NtUserGetTitleBarInfo				x	x		
NtUserGetWindowBand				x			
NtUserGetWindowCompositionAttribute				x	x	x	x

System calls	1	2	3	4	5	6	7
NtUserGetWindowDC				x	x		
NtUserInternalGetWindowText				x	x		
NtUserIsChildWindowDpiMessageEnabled				x	x	x	
NtUserIsNonClientDpiScalingEnabled				x	x	x	x
NtUserIsTopLevelWindow				x	x	x	x
NtUserMsgWaitForMultipleObjectsEx				x			x
NtUserPeekMessage				x		x	x
NtUserPostMessage				x	x	x	
NtUserPostThreadMessage				x			
NtUserQueryInputContext				x	x	x	
NtUserQueryWindow				x	x		
NtUserRemoveProp				x	x		
NtUserSBGetParms				x	x		
NtUserSelectPalette				x	x		
NtUserSetLayeredWindowAttributes				x	x		
NtUserSetProcessDpiAwarenessContext				x	x		
NtUserSetProp				x	x		
NtUserSetScrollInfo				x	x		
NtUserSetWindowCompositionAttribute				x	x	x	
NtUserSetWindowFNID				x	x		
NtUserSetWindowLong				x	x	x	
NtUserSetWindowPos				x	x	x	x
NtUserSetWindowsHookEx				x	x	x	
NtUserShowWindow				x	x		
NtUserUnhookWindowsHookEx				x		x	
NtUserUnregisterClass				x	x	x	x
NtUserUpdateInputContext				x	x	x	x
NtWaitForMultipleObjects				x	x		
NtYieldExecution				x			
NtGdiExcludeClipRect					x	x	
NtGdiExtSelectClipRgn					x		
NtGdiRectVisible					x		
NtPowerInformation					x		x
NtQueryInformationAtom					x		x
NtSetSecurityObject					x		x
NtUserCallHwndLockSafe					x	x	
NtUserChangeWindowMessageFilterEx					x	x	
NtUserGetKeyState					x		
NtUserKillTimer					x		
NtUserNotifyIMEStatus					x		
NtUserSetActiveWindow					x		
NtUserSetImeOwnerWindow					x		
NtUserSetTimer					x	x	x
NtUserSetWinEventHook					x	x	
NtGdiOpenDCW						x	
NtLockFile						x	
NtUnlockFile						x	
NtUserBuildHimcList						x	
NtUserFindWindowEx						x	
NtUserGetRequiredCursorSizes						x	
NtUserInheritWindowMonitor						x	
NtUserMoveWindow						x	
NtUserUnhookWinEvent						x	
NtCancelSynchronousIoFile							x
NtGdiCreateDIBSection							x
NtUserGetClipboardData							x
NtUserMenuItemFromPoint							x

Bibliography

[1] any.run - interactive online malware sandbox. URL https://any.run/.

[2] "sin"-ful spiders: Wizard spider and lunar spider sharing the same web. https://www.crowdstrike.com/blog/sin-ful-spiders-wizard-spider-and-lunar-spider-sharing-the-same-web/. Accessed: 2020-03-12.

[3] Friedex: Bitpaymer ransomware the work of dridex authors. https://www.welivesecurity.com/2018/01/26/friedex-bitpaymer-ransomware-work-dridex-authors/. Accessed: 2020-03-12.

[4] Is 'revil' the new gandcrab ransomware? https://krebsonsecurity.com/2019/07/is-revil-the-new-gandcrab-ransomware/. Accessed: 2020-03-12.

[5] 2020 state of malware report. https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf, . Accessed: 2020-03-12.

[6] Gandcrab. https://www.malwarebytes.com/gandcrab/, . Accessed: 2020-03-12.

[7] Malpedia - gozi. https://malpedia.caad.fkie.fraunhofer.de/details/win.gozi, . Accessed: 2020-03-12.

[8] Threat actor profile: Ta505, from dridex to globeimposter. https://www.proofpoint.com/us/threat-insight/post/threat-actor-profile-ta505-dridex-globeimposter. Accessed: 2020-03-12.

[9] Cyber threats 2019:a year in retrospect. https://www.pwc.co.uk/cyber-security/assets/cyber-threats-2019-retrospect.pdf. Accessed: 2020-03-12.

[10] A connection between the sodinokibi and gandcrab ransomware families? https://www.tesorion.nl/aconnection-between-the-sodinokibi-and-gandcrab-ransomware-families/. Accessed: 2020-03-12.

[11] Dissecting emotet – part 1. https://www.telekom.com/en/blog/group/article/cybersecurity-dissecting-emotet-part-one-592612. Accessed: 2020-03-12.

[12] Ursnif, emotet, dridex and bitpaymer gangs linked by a similar loader. https://blog.trendmicro.com/trendlabs-security-intelligence/ursnif-emotet-dridex-and-bitpaymer-gangs-linked-by-a-similar-loader/. Accessed: 2020-03-12.

[13] Trickbot - malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/win.trickbot. Accessed: 2020-03-12.

[14] Charu C. Aggarwal. Applications of frequent pattern mining. In *Frequent Pattern Mining*, pages 443–467. 2014. doi: 10.1007/978-3-319-07821-2_18. URL https://doi.org/10.1007/978-3-319-07821-2_18.

[15] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499, 1994. URL http://www.vldb.org/conf/1994/P487.PDF.

[16] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. Chapter eight - A taxonomy of software integrity protection techniques. *Advances in Computers*, 112:413–486, 2019. doi: 10.1016/bs.adcom.2017.12.007. URL https://doi.org/10.1016/bs.adcom.2017.12.007.

[17] Martin Apel, Christian Bockermann, and Michael Meier. Measuring similarity of malware behavior. In *The 34th Annual IEEE Conference on Local Computer Networks, LCN 2009, 20-23 October 2009, Zurich, Switzerland, Proceedings*, pages 891–898, 2009. doi: 10.1109/LCN.2009.5355037. URL https://doi.org/10.1109/LCN.2009.5355037.

[18] Sebastian Banescu and Alexander Pretschner. Chapter five - A tutorial on software obfuscation. *Advances in Computers*, 108:283–353, 2018. doi: 10.1016/bs.adcom.2017.09.004. URL https://doi.org/10.1016/bs.adcom.2017.09.004.

[19] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 189–200, 2016. URL http://dl.acm.org/citation.cfm?id=2991114.

[20] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. *TTAnalyze: A tool for analyzing malware.* na, 2006.

[21] Horst Bunke, Pasquale Foggia, C. Guidobaldi, Carlo Sansone, and Mario Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops SSPR 2002 and SPR 2002, Windsor, Ontario, Canada, August 6-9, 2002, Proceedings*, pages 123–132, 2002. doi: 10.1007/3-540-70659-3_12. URL https://doi.org/10.1007/3-540-70659-3_12.

[22] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard E. Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018. URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_06B-2_Caliskan_paper.pdf.

[23] Ricardo J. G. B. Campello, Davoud Moulavi, and J ö rg Sander. Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part II*, pages 160–172, 2013. doi: 10.1007/978-3-642-37456-2_14. URL https://doi.org/10.1007/978-3-642-37456-2_14.

[24] Raymond Canzanese, Spiros Mancoridis, and Moshe Kam. Run-time classification of malicious processes using system call analysis. In *10th International Conference on Malicious and Unwanted Software, MALWARE 2015, Fajardo, PR, USA, October 20-22, 2015*, pages 21–28, 2015. doi: 10.1109/MALWARE.2015.7413681. URL https://doi.org/10.1109/MALWARE.2015.7413681.

[25] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. The tigress c diversifier/obfuscator. *Retrieved August*, 14:2015, 2015.

[26] Gideon Creech and Jiankun Hu. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Trans. Computers*, 63(4):807–819, 2014. doi: 10.1109/TC.2013.13. URL https://doi.org/10.1109/TC.2013.13.

[27] KDD Cup. Dataset. *available at the following website http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html*, 72, 1999.

[28] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 51–62, 2008. doi: 10.1145/1455770.1455779. URL https://doi.org/10.1145/1455770.1455779.

[29] G ö sta Grahne and Jianfei Zhu. High performance mining of maximal frequent itemsets. In *6th International Workshop on High Performance Data Mining*, volume 16, page 34, 2003.

[30] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. 2016.

[31] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 255–270, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/caliskan-islam.

[32] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 309–320, 2011. doi: 10.1145/2046707.2046742. URL https://doi.org/10.1145/2046707.2046742.

[33] Joonhyouk Jang, Sanghoon Choi, and Jiman Hong. A method for resilient graph-based comparison of executable objects. In *Research in Applied Computation Symposium, RACS '12, San Antonio, TX, USA, October 23-26, 2012*, pages 288–289, 2012. doi: 10.1145/2401603.2401666. URL https://doi.org/10.1145/2401603.2401666.

[34] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based" out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, 2007.

[35] Wesley Jin, Sagar Chaki, Cory F. Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary function clustering using semantic hashes. In *11th International Conference on Machine Learning and Applications, ICMLA, Boca Raton, FL, USA, December 12-15, 2012. Volume 1*, pages 386–391, 2012. doi: 10.1109/ICMLA.2012.70. URL https://doi.org/10.1109/ICMLA.2012.70.

[36] Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura, and Jun Miyoshi. API chaser: Taint-assisted sandbox for evasive malware analysis. *JIP*, 27:297–314, 2019. doi: 10.2197/ipsjjip.27.297. URL https://doi.org/10.2197/ipsjjip.27.297.

[37] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. A novel approach to detect malware based on API call sequence analysis. *IJDSN*, 11:659101:1–659101:9, 2015. doi: 10.1155/2015/659101. URL https://doi.org/10.1155/2015/659101.

[38] Hyunjoo Kim, Jonghyun Kim, Youngsoo Kim, Ikkyun Kim, Kuinam J. Kim, and Hyuncheol Kim. Improvement of malware detection and classification using API call sequence alignment and visualization. *Cluster Computing*, 22(Suppl 1):921–929, 2019. doi: 10.1007/s10586-017-1110-2. URL https://doi.org/10.1007/s10586-017-1110-2.

[39] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 769–780, 2015. doi: 10.1145/2810103.2813642. URL https://doi.org/10.1145/2810103.2813642.

[40] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 287–301, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat.

[41] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao - yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 351–366, 2009. URL http://www.usenix.org/events/sec09/tech/full_papers/kolbitsch.pdf.

[42] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1691–1708, 2017. doi: 10.1145/3133956.3134099. URL https://doi.org/10.1145/3133956.3134099.

[43] Itzik Kotler and Amit Klein. Pinjectra. https://github.com/SafeBreach-Labs/pinjectra, 2013.

[44] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*, pages 5:1–5:6, 2013. doi: 10.1145/2430553.2430558. URL https://doi.org/10.1145/2430553.2430558.

[45] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[46] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[47] McAfee. Mcafee labs threats report 2018 q4. 2019.

[48] Xiaozhu Meng, Barton P. Miller, and Kwang - Sung Jun. Identifying multiple authors in a binary program. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II* , pages 286–304, 2017. doi: 10.1007/978-3-319-66399-9_16. URL https://doi.org/10.1007/978-3-319-66399-9_16.

[49] Jiang Ming, Zhi Xin, Pengwei Lan, Dinghao Wu, Peng Liu, and Bing Mao. Replacement attacks: Automatically impeding behavior-based malware specifications. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, pages 497–517, 2015. doi: 10.1007/978-3-319-28166-7_24. URL https://doi.org/10.1007/978-3-319-28166-7_24.

[50] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 253–270, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming.

[51] Tempestt Neal, Kalaivani Sundararajan, Aneez Fatima, Yiming Yan, Yingfei Xiang, and Damon Woodard. Surveying stylometry techniques and applications. *ACM Computing Surveys (CSUR)*, 50(6):1–36, 2017.

[52] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[53] Stavros D. Nikolopoulos and Iosif Polenakis. A graph-based model for malware detection and classification using system-call groups. *J. Computer Virology and Hacking Techniques*, 13(1):29–46, 2017. doi: 10.1007/s11416-016-0267-1. URL https://doi.org/10.1007/s11416-016-0267-1.

[54] G ö sta Grahne and Jianfei Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. Knowl. Data Eng.*, 17(10):1347–1362, 2005. doi: 10.1109/TKDE.2005.166. URL https://doi.org/10.1109/TKDE.2005.166.

[55] Yoshihiro Oyama. Investigation of the diverse sleep behavior of malware. *JIP* , 26:461–476, 2018. doi: 10.2197/ipsjjip.26.461. URL https://doi.org/10.2197/ipsjjip.26.461.

[56] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.

[57] Young Hee Park, Douglas S. Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop, CSIIRW 2010, Oak Ridge, TN, USA, April 21-23, 2010*, page 45, 2010. doi: 10.1145/1852666.1852716. URL https://doi.org/10.1145/1852666.1852716.

[58] Jian Pei, Jiawei Han, Behzad Mortazavi - Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. Knowl. Data Eng.*, 16(11):1424–1440, 2004. doi: 10.1109/TKDE.2004.77. URL https://doi.org/10.1109/TKDE.2004.77.

[59] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. Malpedia: a collaborative effort to inventorize the malware landscape, 2017. URL https://malpedia.caad.fkie.fraunhofer.de/.

[60] Yong Qiao, Yuexiang Yang, Lin Ji, and Jie He. Analyzing malware by abstracting the frequent itemsets in API call sequences. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, pages 265–270, 2013. doi: 10.1109/TrustCom.2013.36. URL https://doi.org/10.1109/TrustCom.2013.36.

[61] Paulo E. Rauber, Alexandre X. Falc ã o, and Alexandru C. Telea. Visualizing time-dependent data using dynamic t-sne. In *Eurographics Conference on Visualization, EuroVis 2016, Short Papers, Groningen, The Netherlands, 6-10 June 2016*, pages 73–77, 2016. doi: 10.2312/eurovisshort.20161164. URL https://doi.org/10.2312/eurovisshort.20161164.

[62] Nathan E. Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pages 172–189, 2011. doi: 10.1007/978-3-642-23822-2_10. URL https://doi.org/10.1007/978-3-642-23822-2_10.

[63] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, volume 10885 of *Lecture Notes in Computer Science*, pages 372–392. Springer, 2018. doi: 10.1007/978-3-319-93411-2_17. URL https://doi.org/10.1007/978-3-319-93411-2_17.

[64] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamzeh. Malware detection based on mining API calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 1020–1025, 2010. doi: 10.1145/1774088.1774303. URL https://doi.org/10.1145/1774088.1774303.

[65] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, 2016. doi: 10.1145/2886012. URL https://doi.org/10.1145/2886012.

[66] Jun Sese and Shinichi Morishita. Itemset classified clustering. In *Knowledge Discovery in Databases: PKDD 2004, 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy, September 20-24, 2004, Proceedings*, pages 398–409, 2004. doi: 10.1007/978-3-540-30116-5_37. URL https://doi.org/10.1007/978-3-540-30116-5_37.

[67] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[68] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, pages 1–25, 2008. doi: 10.1007/978-3-540-89862-7_1. URL https://doi.org/10.1007/978-3-540-89862-7_1.

[69] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[70] Symantec. Internet security threat report. 2019.

[71] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.

[72] Jorge Maestre Vidal, Marco Antonio Sotelo Monge, and L. Javier Garc í a - Villalba. A novel pattern recognition system for detecting android malware by analyzing suspicious boot sequences. *Knowl. Based Syst.*, 150:198–217, 2018. doi: 10.1016/j.knosys.2018.03.018. URL https://doi.org/10.1016/j.knosys.2018.03.018.

[73] Philippe Fournier - Viger, Cheng - Wei Wu, Antonio Gomariz, and Vincent S. Tseng. VMSP: efficient vertical mining of maximal sequential patterns. In *Advances in Artificial Intelligence - 27th Canadian Conference on Artificial Intelligence, Canadian AI 2014, Montr é al, QC, Canada, May 6-9, 2014. Proceedings*, pages 83–94, 2014. doi: 10.1007/978-3-319-06483-3_8. URL https://doi.org/10.1007/978-3-319-06483-3_8.

[74] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9-12, 1999*, pages 133–145, 1999. doi: 10.1109/SECPRI.1999.766910. URL https://doi.org/10.1109/SECPRI.1999.766910.

[75] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 732–744, 2015. doi: 10.1145/2810103.2813663. URL https://doi.org/10.1145/2810103.2813663.

[76] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. An intelligent pe-malware detection system based on association mining. *Journal in Computer Virology*, 4(4):323–334, 2008. doi: 10.1007/s11416-008-0082-4. URL https://doi.org/10.1007/s11416-008-0082-4.

[77] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[78] Wen Zhang, Taketoshi Yoshida, Xijin Tang, and Qing Wang. Text clustering using frequent itemsets. *Knowl. Based Syst.*, 23(5):379–388, 2010. doi: 10.1016/j.knosys.2010.01.011. URL https://doi.org/10.1016/j.knosys.2010.01.011.