

Thilo Denzer

**NTNU**  
Norwegian University of  
Science and Technology  
Faculty of Information Technology and Electrical  
Engineering  
Department of Information Security and Communication  
Technology

Thilo Denzer

# Similarity-based Intelligent Malware Type Detection through Multiple Sources of Dynamic Characteristics

June 2019





Norwegian University of  
Science and Technology

# Similarity-based Intelligent Malware Type Detection through Multiple Sources of Dynamic Characteristics

Information Security

Submission date: June 2019

Supervisor: Assoc. Prof. Dr. Geir Olav Dyrkolbotn

Co-supervisor: Dr. Andrii Shalaginov

Norwegian University of Science and Technology  
Department of Information Security and Communication  
Technology



## Preface

The following work deals with multinomial malware classification of ten malware families with machine learning algorithms. It is carried out in the context of a Master's thesis in MIS (Information Security) at NTNU. It was conducted during the spring semester 2019. The broad idea of the topic was brought up by my supervisors, Geir Olav Dyrkolbotn and Andrii Shalaginov, before it was specified in more detail and finalised in a direct discussion between us. It is still a relevant topic today since the malware landscape is constantly growing and evolving and therefore, further research has to be conducted in the area of this topic. Nonetheless, the thesis is targeted at an audience from the field of information technology with the focus on forensics. However, no expert knowledge of the malware landscape is needed to understand the elaboration since all used terms and methods are explained.

01-06-2019



## Acknowledgements

I want to give my sincere gratitude to my supervisors Geir Olav Dyrkolbotn and Andrii Shalaginov for giving me the opportunity to write my Master's thesis. Throughout the whole work they always provided extensive support and guidance, professional advice and constant feedback. In addition, they provided me with the malware samples and the static features used in the experiment. I want to thank NTNU for the provision of the resources for my research. I want to acknowledge my parents, Michael and Nikola Denzer, for the financial support and therefore, the opportunity to study in Norway as well as the personal encouragement. A big thanks goes to my sister, Vera Denzer, for proofreading my thesis report. Moreover, I want to thank my friend Mahesh Thapa for providing me with additional hardware for testing purposes.

T.D.





## Abstract

Malware analysts face challenges related to increasing number of malware variants emerging every year. Conventional classification of Windows PE32 executables into benign and malicious is no longer sufficient and needs refinement when it comes to detecting similar functionality malware samples belonging to the same category. Thus, it is important to explore sources of multiple dynamic characteristics that can substantially improve similarity-based malware detection through indicators of compromise from disk, network and memory. The goal of this thesis is to explore a way to improve multinomial malware classification by exploiting available dynamic characteristics.

In this work dynamic features were extracted with the help of the automated malware analysis system Cuckoo Sandbox and classified into their ten respective families with the machine learning library Weka. It has been analysed which dynamic features contribute the most for multinomial malware classification and what the performance gain is compared to static feature-based malware classification. An overall classification result of 87.5% could be achieved with the best performing dynamic features being the modified and opened registry keys, the created and modified files, the loaded DLLs and the resolved hosts. The best performing classifier was Random Forest. This result, however, can be improved by adding more dynamic features or combine them with selected static features in the future.

**Keywords:** Malware classification, Malware Family, Machine Learning, Random Forest, Dynamic Features, Static Features, Performance Comparison



## Contents

<b>Preface</b> . . . . .	<b>i</b>
<b>Acknowledgements</b> . . . . .	<b>iii</b>
<b>Abstract</b> . . . . .	<b>v</b>
<b>Contents</b> . . . . .	<b>vii</b>
<b>List of Figures</b> . . . . .	<b>ix</b>
<b>List of Tables</b> . . . . .	<b>xi</b>
<b>List of Algorithms</b> . . . . .	<b>xiii</b>
<b>Acronyms</b> . . . . .	<b>xv</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Description . . . . .	2
1.3 Research Questions . . . . .	3
1.4 Proposed Contribution . . . . .	3
1.5 Structural Outline . . . . .	4
<b>2 Related Work</b> . . . . .	<b>5</b>
<b>3 Background Literature Study and State of the Art in Malware Analysis</b> . . . . .	<b>13</b>
3.1 Static vs. Dynamic . . . . .	13
3.2 Malware Classification . . . . .	14
3.3 Malware Landscape . . . . .	14
3.3.1 Malware Types . . . . .	15
3.3.2 Malware Families . . . . .	15
3.3.3 Malware Platforms . . . . .	15
3.4 Evasion Techniques . . . . .	15
3.4.1 Obfuscation . . . . .	16
3.4.2 Anti-* . . . . .	16
3.4.3 Polymorphic & Metamorphic Malware . . . . .	17
3.5 Malware Naming Issue . . . . .	17
3.6 Machine Learning . . . . .	18
<b>4 Methodology</b> . . . . .	<b>21</b>
4.1 Data Set . . . . .	21
4.2 Data Set processing and analysis . . . . .	22
4.3 Feature Extraction . . . . .	23
4.4 Feature Selection . . . . .	24
4.5 ML-aided Malware Classification . . . . .	25

<b>5</b>	<b>Experimental Setup</b>	<b>27</b>
5.1	Data Set Collection	27
5.2	Software Versions	27
5.3	Hardware Specifications	27
5.4	Experimental Design and Implementation	27
5.4.1	Malware Pre-processing	28
5.4.2	Sandbox-related Configuration Details	29
5.4.3	Dynamic Malware Analysis and Feature Extraction	31
5.4.4	Feature preparation for Weka	34
5.4.5	Application of Machine Learning	37
<b>6</b>	<b>Results &amp; Analysis</b>	<b>39</b>
<b>7</b>	<b>Discussion</b>	<b>47</b>
7.1	Implications	52
7.1.1	Theoretical Implications	52
7.1.2	Practical Implications	52
7.2	Limitations of the Study	52
<b>8</b>	<b>Conclusion</b>	<b>55</b>
	<b>Bibliography</b>	<b>57</b>
<b>A</b>	<b>Appendix</b>	<b>1</b>
A.1	PEframe Python code	1
A.2	Powershell script to extract CPU usage and memory usage during malware execution	1
A.3	Python code for pre-processing task. Find errors, copy memory features into report and group all reports based on family.	2
A.4	Python code to extract all dynamic features and create Weka file	3
A.5	Python code to extract all entries from static feature dataset	5
A.6	Confusion Matrix for Random Forest Classifier and oneR-based Feature Selection	6

## List of Figures

1	Statistic for the total amount of malware and the amount of new malware in the last ten years [1, 2] . . . . .	2
2	Screenshot from VirusTotal on the hash value of the WannaCry Decryptor [3] . . . . .	3
3	Malware analysis pyramid based on Pektaş [4] . . . . .	14
4	Computer Antivirus Research Organization malware naming scheme used by Microsoft [5] . . . . .	18
5	Process flowchart of the malware analysis process during the experiment based on Banin et al. [6] . . . . .	23
6	Different Steps of the Experimental Setup . . . . .	28
7	Cuckoo's main architecture [7] . . . . .	31
8	Correlation between the dynamic features and the three used feature selection methods. Colour key: green = 1-10; orange = 11-20; red = 21-31 . . . . .	41
9	Detailed performance evaluation of each malware family for Random Forest classifier and Correlation-based feature selection of dynamic features . . . . .	43
10	Confusion matrix for Random Forest classifier and Correlation-based feature selection of dynamic features . . . . .	43
11	Confusion matrix for Random Forest classifier of the seven best performing families of dynamic features . . . . .	44
12	Comparison of TP rates from Correlation-based feature selection of static against dynamic features by class . . . . .	46
13	Data distribution of the worst performing dynamic features . . . . .	49
14	Data distribution of the best performing dynamic features . . . . .	49
15	Data distribution of static features with obfuscator as red . . . . .	52



## List of Tables

1	Most important references of multinomial malware classification regarding RQ2 and RQ3 . . . . .	10
2	Software used in the Experiment . . . . .	28
3	In the experiment used malware families with accurate amount distribution . . . . .	34
4	All extracted dynamic features . . . . .	35
5	Used feature selection methods with connection to Weka module names for reproducibility . . . . .	37
6	Weighted average of different classifiers with binary approach classification of dynamic features . . . . .	40
7	Weighted average of different classifiers with amount-based approach classification of dynamic features . . . . .	40
8	Feature selection methods of dynamic features . . . . .	42
9	Feature selection methods of static features . . . . .	45
10	Data points of the peak of the curve based on Fig. 13 and Fig. 14 . . . . .	50





## List of Algorithms

1	Pseudocode for static analysis of malware samples with PEframe . . . . .	29
2	Modified default Python script for analysing Windows executables used by Cuckoo . . . . .	30
3	Pseudocode to remove errors, copy memory features into report and group all reports based on family . . . . .	33
4	Pseudocode to extract all dynamic features and create Weka file . . . . .	36



## Acronyms

- ANN** Artificial Neural Network. [19](#)
- API** Application Programming Interface. [5–9](#), [13](#), [16](#), [51](#), [55](#)
- ARFF** Attribute-Relation File Format. [34](#)
- AUC** Area under the Curve. [9](#), [55](#)
- AV** Antivirus. [13](#)
- BIOS** Basic Input/Output System. [7](#)
- CARO** Computer Antivirus Research Organization. [17](#), [22](#)
- CPU** Central Processing Unit. [24](#), [27](#), [29](#), [31](#), [35](#)
- CSV** Comma-separated Values. [34](#), [36](#)
- DAME** Dark Avenger Mutation Engine. [17](#)
- DLL** Dynamic-link Library. [24](#), [48](#), [55](#)
- DNS** Domain Name System. [8](#), [24](#)
- FP** False Positive. [39](#)
- FS** Feature Selection. [24](#)
- FTP** File Transfer Protocol. [8](#)
- HMM** Hidden Markov Model. [9](#)
- HTTP** Hypertext Transfer Protocol. [8](#), [24](#)
- HTTPS** Hypertext Transfer Protocol Secure. [24](#), [30](#)
- ID** Identification. [31](#), [32](#)
- IP** Internet Protocol. [24](#), [48](#)

**IRC** Internet Relay Chat. [24](#)

**JSON** JavaScript Object Notation. [31](#), [32](#), [34](#)

**kNN** k-nearest Neighbours. [19](#)

**LCS** Longest Common Subsequence. [5](#), [6](#)

**LR** Logistic Regression. [19](#)

**MAC** Media Access Control. [7](#), [16](#)

**MD5** Message-Digest Algorithm 5. [2](#), [36](#)

**ML** Machine Learning. [18](#)

**NB** Naive Bayes. [18](#)

**NOP** No Operation. [16](#)

**OS** Operating System. [15](#), [24](#), [29](#)

**pcap** packet capture. [8](#), [24](#)

**PDF** Portable Document Format. [53](#)

**PE32** Portable Executable - 32bit. [2](#), [3](#), [9](#), [27](#), [55](#)

**PUP** Potentially Unwanted Program. [9](#), [50](#)

**RAM** Random-Access Memory. [27](#), [31](#), [35](#)

**RAT** Remote Access Tool. [21](#)

**RF** Random Forest. [19](#), [40](#), [47](#)

**ROC** Receiver Operating Characteristic. [9](#), [39](#)

**ROP** Return-oriented Programming. [7](#)

**SCADA** Supervisory Control and Data Acquisition. [1](#)

**SMTP** Simple Mail Transfer Protocol. [8](#), [24](#)

**SRM** Structural Risk Minimization. [19](#)

**SSL** Secure Sockets Layer. [24](#), [29](#)

**SVM** Support Vector Machine. [7](#), [9](#), [18](#)

**TCP** Transmission Control Protocol. [8](#)

**TLS** Transport Layer Security. [24](#)

**TP** True Positive. [39](#), [41](#), [44](#), [55](#)

**TPE** Trident Polymorphic Engine. [17](#)

**UDP** User Datagram Protocol. [8](#)

**UPX** Ultimate Packer for Executables. [7](#), [16](#)

**URL** Uniform Resource Locator. [13](#)

**VM** Virtual Machine. [7](#), [16](#), [22](#), [29](#), [30](#), [53](#), [55](#)

**WMI** Windows Management Instrumentation. [48](#), [55](#)

**XMLRPC** Extensible Markup Language Remote Procedure Call. [31](#)



# 1 Introduction<sup>1</sup>

Today's society is more and more connected. Technology is ubiquitous and we are depending on a working IT infrastructure like never before. Malfunctioning of some parts of this infrastructure could be devastating and would lead to considerable damage. A big proportion of our daily activities are based on the internet and similar network-based technology. Working from home, buying necessities online or just sharing information on social media, the internet is not as secure as it used to be [2]. Malware, short for malicious software, makes up a huge amount of the internet traffic and it is pretty easy to infect oneself. A malware is considered to be *a sequence of instructions that performs malicious activity on a computer* [8]. The extend of the malicious activity depends on various factors but mostly on the intent of the malware author and the level of protection of the victim system. Usually, such activities include but are not limited to stealing credentials or other useful data, downloading the actual malware payload, disrupting the system, installing a backdoor, elevating existing privileges and more. Nowadays it becomes almost trivial to attack systems with pre-crafted malware from the internet. With do-it-yourself malware development kits, novices with little to none coding skills or technical know-how can create their own powerful malware. The amount of those tool kits is growing rapidly. However, malware is not just found on private users but also on company systems and even more severe on critical infrastructure or governmental systems as shown in the past by well known malware like Stuxnet or cases in which authorities got compromised. Stuxnet was a malware attacking SCADA systems from the manufacturer Siemens - Simatic S7 - heavily deployed in the Iranian nuclear power program causing substantial physical damage.

## 1.1 Motivation

With the rise of the internet the distribution of malware is simpler as ever before. Thus, the malware landscape is constantly evolving and malware analysts face the challenge of increasing number of malware every year. The statistics diverge from source to source but they have an increase in numbers in common as summarised in Fig. 1 for the total amount of malware and the amount of new malware in the last ten years by the independent IT-Security institute AV-TEST [1]. According to the statistics from AV-TEST, there has been 121.67 million new malware samples found from a total amount of 719.15 million in 2017. This means that 16.9% of the malware found in 2017 are considered to be new malware samples. *The number of new malicious files processed by Kaspersky Lab's in-lab detection technologies reached 360,000 a day in 2017, which is 11.5% more than the*

---

<sup>1</sup>**Disclaimer:** The thesis is built upon the course - 'IMT4205 Research Project Planning', submitted in December 2018, since it is meant to be a continuation of the work done from this course. Therefore, some parts from this chapter might show strong similarities to the work done in the course.

previous year [9]. As the malicious data increases, it is only natural that malware analysts are overwhelmed with the sheer amount of malware samples at some point. It is too cumbersome to analyse every single malware. Therefore, there is a big demand for automatic solutions which don't require the analysts to go through every single malware manually.

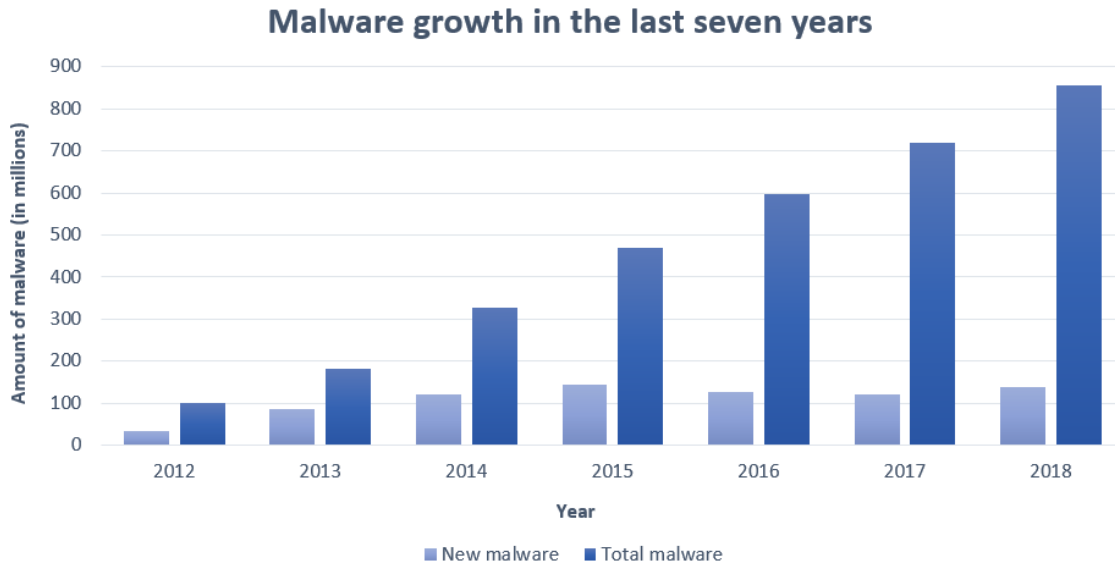


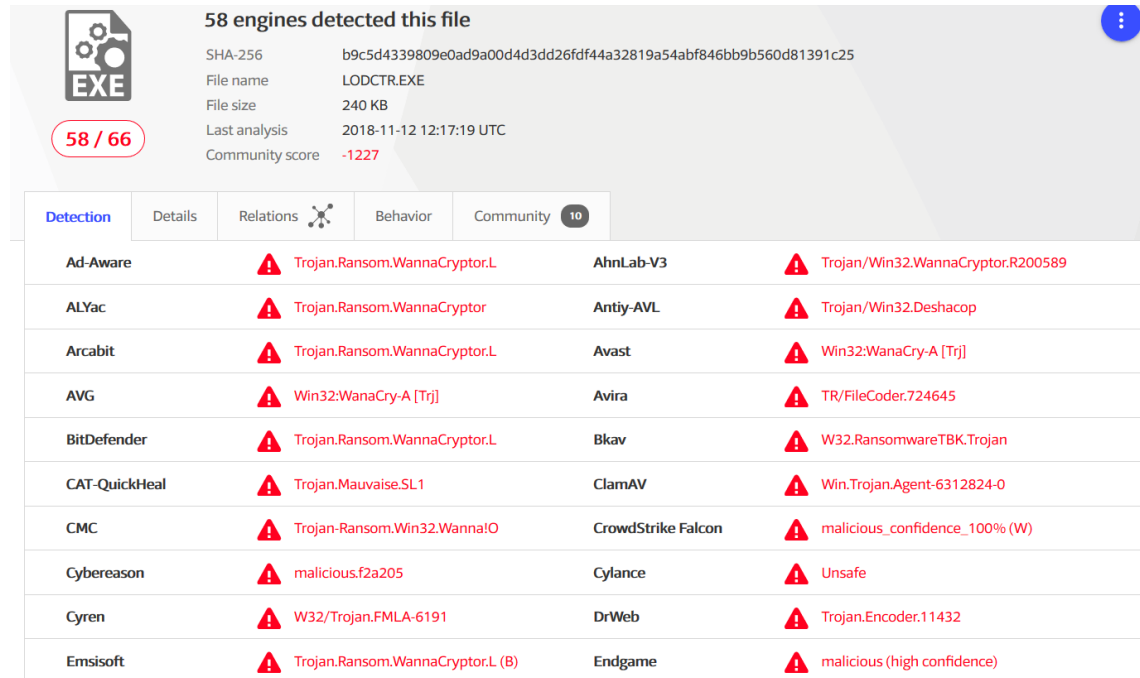
Figure 1: Statistic for the total amount of malware and the amount of new malware in the last ten years [1, 2]

## 1.2 Problem Description

A signature-based detection approach is the main technique used for malware detection by anti-virus programs [10, 11] with a conventional classification of Windows PE32 executables into malicious and benign. Binary classification is usually done with signatures, partial matching, regular expressions or heuristics [12]. This clustering approach is no longer sufficient because e.g. of malware diversification [12] which focuses on avoiding similarity-based matching of malware by randomly diversifying code and data regions to reduce the similarity between malware mutants. Therefore, malware classification needs refinement when it comes to detecting similar functionality malware samples belonging to the same category. Moreover, static signature-based detection of malware is obsolete and becomes less relevant every year with growing malware threats. Multinomial malware detection and classification based on dynamic indicators of compromise from memory, disk and network, which could substantially improve anti-malware solutions, need to be explored and enhanced. Another challenge for the anti-malware infrastructure is the absence of agreement between anti-virus vendors on how the malware should be named. E.g. uploading a hash value from the WannaCry Decryptor (MD5: 7bf2b57f2a205768755c07f238fb32cc) to the online scan engine



VirusTotal, which combines many anti-virus products, results in many different naming conventions of the various anti-virus software as seen in Fig. 2.



58 engines detected this file

SHA-256: b9c5d4339809e0ad9a00d4d3dd26fdf44a32819a54abf846bb9b560d81391c25  
 File name: LODCTR.EXE  
 File size: 240 KB  
 Last analysis: 2018-11-12 12:17:19 UTC  
 Community score: -1227

58 / 66

Detection	Details	Relations	Behavior	Community
Ad-Aware	Trojan.Ransom.WannaCryptor.L			AhnLab-V3
ALYac	Trojan.Ransom.WannaCryptor			Antiy-AVL
Arcabit	Trojan.Ransom.WannaCryptor.L			Avast
AVG	Win32:WanaCry-A [Trj]			Avira
BitDefender	Trojan.Ransom.WannaCryptor.L			Bkav
CAT-QuickHeal	Trojan.Mauvaise.SL1			ClamAV
CMC	Trojan-Ransom.Win32.WannaIO			CrowdStrike Falcon
Cybereason	malicious.f2a205			Cylance
Cyren	W32/Trojan.FMLA-6191			DrWeb
Emsisoft	Trojan.Ransom.WannaCryptor.L (B)			Endgame
				Trojan/Win32.WannaCryptor.R200589
				Trojan/Win32.Deshacop
				Win32:WanaCry-A [Trj]
				TR/FileCoder.724645
				W32.RansomwareTBK.Trojan
				Win.Trojan.Agent-6312824-0
				malicious_confidence_100% (W)
				Unsafe
				Trojan.Encoder.11432
				malicious (high confidence)

Figure 2: Screenshot from VirusTotal on the hash value of the WannaCry Decryptor [3]

### 1.3 Research Questions

To summarise the explained issues in a more scientific manner the following research questions have been developed:

- RQ1:** What are the cases and particular scenarios, in which conventional signature-based malware detection and ML-aided detection with the help of static features may fail?
- RQ2:** What are the dynamic behavioural features in contemporary Windows malware analysis that can be successfully used in multinomial classification?
- RQ3:** What is the performance gain of comprehensive dynamic features used for intelligent malware category detection in relation to static features extracted from PE32 headers?

### 1.4 Proposed Contribution

This project will cover an important aspect of this problem by addressing the issue of malware classification. The goal of this thesis is to explore a way to improve multi-class malware classification by exploiting available dynamic characteristics. Instead of doing a binary malware classification into malicious and benign, malware is classified by its respective group based on its functionality. An

experiment based on existing open source tools will be conducted throughout this thesis in which malware is dynamically analysed and based on dynamic features from memory, disk and network classified into its respective family.

## 1.5 Structural Outline

The whole thesis is divided into eight chapters. In Chapter 2 an overview of the current state-of-the-art is given with the most important literature for multinomial malware classification which functions as foundation for this thesis. Chapter 3 sums up all necessary background information and definitions concerning malware analysis to explain the used terms, techniques and methods within this paper. Afterwards, Chapter 4 introduces the methodology on which the experiment, presented in Chapter 5, is being build on. The experiment lists all practical work conducted during the thesis, explaining what has been done, which algorithms, software and hardware has been used and how it was performed. In Chapter 6 the results of the experiment from the previous chapter are presented. Afterwards, those results are analysed, interpreted and evaluated in Chapter 7 alongside an insight into the limitations of the overall employed methodology. Lastly, Chapter 8 concludes the elaboration and mentions what has to be done in the near future to improve multinomial malware classification with dynamic features.

## 2 Related Work<sup>1</sup>

Static malware analysis is mostly used for common malware detection like in anti virus software. Although this is a quick and easy way to identify malware, it is also pretty unreliable. Malware authors often obfuscate or dynamically change their malware so that signature-based or static analysis as a whole is no longer sufficient. This is shown by Payer et al. [12] and their proposed approach of malware diversification, which randomly diversifies code and data regions. It reduces the similarity between the same instances of malware enough to aggravate or even disable direct, similarity-based matching. This makes static similarity-based matching no longer effective. However, a lot of recent research deals with the topic of finding similarities between malware based on dynamic methods. The approach by Park et al. [13] is about similarity-based malware detection by analysing assembly instruction sequences in executables found on the hard disk. Yi et al. [14] are proposing an approach using DepSim to find semantic matches between malicious software based on control and data dependency graphs *achieved by identifying the maximum common subgraph*. DepSim, therefore, uses dynamic taint analysis and backtracking techniques. The experiment showed that DepSim can successfully find semantic similarities and can even deal with obfuscated or packed malware. Liu et al. [10] are calculating the level of similarity by analysing function-call graphs based on the graph similarity flooding algorithm. Alkhateeb [11] is detecting malware using similarities in API calls.

Moreover, clustering malware into the two categories benign and malicious is no longer sufficient as well. A new approach is to classify malware based on its functionality, meaning the type of malware family they are derived from. An approach of malware classification is proposed by Han et al. [15] and is based on sequence characteristics of API calls. Another approach is based on dynamic API call counts in which Kim et al. [16] make a similarity analysis on the results of a frequency API call investigation. The authors are utilising the open source tool Cuckoo Sandbox to extract the API calls and align them based on the count of calls. The API calls are then classified into nine types based on the malware variant. The ten APIs with the highest frequencies of calls are used to define the malware type. In a different paper Kim et al. [17] present an approach of malware detection and classification based on API call sequence alignment and visualisation. The authors propose a system which is *composed of five functional steps: (1) Data collection and Sequence extraction; (2) Feature extraction and preprocessing; (3) Clustering; (4) Behavioural sequence chain extraction; and (5) Detection and classification*. 1790 malware samples and 1138 benign API call sequences were used to produce a F-measure of 94.3% with the similarity-LCS (longest common subsequence) method. An approach that is not based on static nor on dynamic malware analysis is presented by

---

<sup>1</sup>**Disclaimer:** The thesis is built upon the course - 'IMT4205 Research Project Planning', submitted in December 2018, since it is meant to be a continuation of the work done from this course. Therefore, some parts from this chapter might show strong similarities to the work done in the course.

Narayanan et al. [18]. The malware is visualised as a binary pattern, re-shaped as 2D matrix and then visualised as image. The authors state *that visualizing the malware programs as images opens up the path for broader spectrum analysis*. Chia-mei and Gu-hsin [19] present an approach of malware classification based on content similarity and directory structure similarity. The authors are capturing malware with the help of a honeypot system and a decompressor to extract file structure and content in the feature extraction phase. During their work they only focus on source code files and will ignore all binary files. Lastly, in the Cluster system the malware will be clustered and classified into a database. The accuracy of the proposed system is stated to be 96.25%. Grégio et al. [20] present an approach in which the [Longest Common Subsequence](#) between two malware traces is calculated based on appearing bigrams. Two same malware variants should share specific features or attributes particular to its malware family. The authors can then make assumptions about which malware code generated similar memory content. Based on this malware trace similarity technique they built a clustering application on top to group similar malware samples and another application to find cases of code reuse. They analysed 16,248 malware samples in their system and produced an average precision value of 0.843 to the reference clustering sets (static and dynamic). Liu et al. [21] are using *Opcode n-gram, grey-scale images and the import function to extract malware features*. In the decision-making phase classifiers are trained with machine learning algorithms in order to group the suspicious malware samples in its corresponding family within the clustering phase. The authors are doing the malware classification by using Python's machine learning module SCikit-learn containing the classification algorithms. 21,740 malware samples from nine families are used for their experiment, which achieved a best accuracy for clustering malware of 0.853 when  $n$  is equal to 3 and all seven classifiers are combined. In terms of detecting new malware, the authors used 900 malware samples, 810 from known samples and 90 from new ones. 78 of the 90 were correctly assigned to its category while 12 were assigned wrongly to different categories which results in an accuracy of 0.867. Lee et al. [22] are using the Cuckoo Sandbox as a dynamic method to extract the [API](#) behaviour data, group them using an  $n$ -gram model and calculate the similarities to group the malware mutants in a database. The extracted [API](#) sequences are then compared by the means of the cosine similarity method and with the local cluster coefficient the malware codes are categorised into groups. The experiment showed the following results; as the similarity threshold increases *the number of members in each group decreases, but the accuracy of the members of the malware group increases*. E.g. for the threshold of 95% 213 groups with 2065 members were created, which means that 78.25% of malicious codes could be grouped by a total number of 2639 analysed malware samples. Islam et al. [23] propose an integrated method of static and dynamic features for malware classification. They use the trace tool HookMe to analyse [API](#) functions and a collection of machine learning algorithms called Weka library to classify the malware. They achieved an accuracy of 97.055% while also providing a list of similar existing techniques for comparison.

Concerning the research questions stated above in Chapter 1, there is some literature dealing specifically with those issues. The already mentioned work from Payer et al. [12] addresses the issue of malware diversification which is a particular method of malware authors to avoid conventional signature-based malware detection. Sathyanarayan et al. [24] are using static analysis to

extract [API](#) calls to construct malware class signatures. To detect malicious programs the authors statistically compare the [API](#) calls of the malware with the class signatures. However, there are cases in which their approach fails, as stated by the authors: *a limitation of our approach is that it does not work for packed malware*. Another case in which conventional malware detection with the help of static features might fail is presented by Biondi et al. [25] and their tutorial for malware detection and evasion techniques. The authors introduce malware obfuscation techniques used to hide information with the example of *XOR-obfuscated strings to avoid string-based detection* and a complete code obfuscation by statically compiling the binary and packing it with [UPX](#) to change the structure of the binary. Static-feature-based malware detection can easily be bypassed by the means of obfuscation methods [26]. Even more obfuscation techniques are displayed by Preda et al. [27] who divide it into two areas, conservative and non-conservative obfuscation techniques. Moreover, anti-sandboxing techniques are presented in the white paper of lastline [28], an American cyber security company. The authors state that advanced malware can detect the presence of a [VM](#) by the means of registry keys, special [VM](#) tools, particular processes and services, *identifying the BIOS serial number or MAC address of the virtual network adapt*, specific structure of system memory or certain hardware parameters. All those techniques might indicate the malware being executed in a [VM](#), which then usually either hides its malicious intent or deletes itself [25]. Lastline also elaborates advanced evasion techniques in detail. They present six particular malware evasion techniques for which malware detection might fail. They cover *Stalling Delays, User Action Required Delays, Intelligent Suspension of Malicious Activity, Fragmentation, Return-Oriented Programming (ROP) Evasion and Rootkits*. Sharma [29] expounds that machine learning approaches are computational demanding which is not suited for ordinary end users. Islam and Altas [26] are using a comparative approach of malware classification by using string information as static features, [API](#) calls and parameters as dynamic features and machine learning techniques. The authors conclude that dynamic features are *a necessary complement to static techniques* and that their approach works better for old malware samples instead of the latest ones. Consequently, the used features are not good enough to properly classify malware. Griffin et al. [30] propose an automatic system for malware detection using string signatures but the system fails for packed or metamorphosed malware. Their system is also limited if it cannot generate good signatures which happens when the average number of malware family variants is too low. Mohamed and Ithnin [31] present major drawbacks of traditional signature-based malware detection systems based on data mining, machine learning, [SVM](#) and [API](#) call graph techniques. The requirement of an up-to-date and maintained signature database, the impossibility of detecting new attacks, so called zero-day attacks, and the fact that simple obfuscation techniques can evade detection are listed. Nataraj et al. [32] mention that visualisation techniques to detect and classify malware has its limitations. Malware binaries are visualised as grey-scale images and a k-nearest neighbour approach with the Euclidean distance is used. They state that malware authors could relocate sections or add redundant data in a binary to avoid detection. Furthermore, machine learning techniques can easily produce many false positives diminishing users' trust into the machine-learning-based approaches [33]. Regarding dynamic behavioural features used for successful multinomial classification, it has been proven

that memory access patterns can not only be used to successfully detect malware [6] but also to classify malware into ten malware families respectively types as shown by Banin and Dyrkolbotn [34]. In their work they extracted 29 best features to create relatively accurate models with the emphasis on a *less accurate but more understandable model*. They achieved an accuracy between 0.56 to 0.99 for malware families and 0.43 to 0.86 for malware types. A popular dynamic feature for malware classification are API calls [15, 16, 17, 22, 23] as listed in detail above. Bounouh et al. [35] are using a hybrid approach by taking static and dynamic features in considerations to improve classification accuracy. Files (create, modify, read, delete, memory-mapped), registries (create key, delete key, monitor key, modify value, read value, delete value), processes (create process, delete process, create thread, read shared memory, write shared memory) and network features (TCP, SMTP, UDP, HTTP, FTP, ping requests, DNS queries, data) are used as dynamic features. Ying et al. [36] are capturing malware variants execution traces. From those *dynamic execution traces*, *API calls*, *return value[s]* and *module name[s]* are extracted as features and their number of occurrences are saved in a trace frequency information table. Tian et al. [37] are extracting behavioural features from API system calls by collecting run-time trace reports with the help of the trace tool 'HookMe'. They then use the collection of machine learning algorithms from the Weka library to classify the malware. Shalaginov and Franke [38] use dynamic characteristics and machine learning for their approach of multinomial malware classification. They analyse disk activities with two sub-domains *low-level access by the application that includes modification, deletion and writing to the file on a disk storage* and registry modifications. Moreover, network traffic is analysed as malware usually try to download payloads, communicate with the attacker or upload sensitive user information over the network. Last, the authors mention memory footprints as possible option for dynamic characteristics in multinomial malware classification. However, they do not elaborate it any further since the collection process hardly yields any acceptable results and is significantly more cumbersome than the other two methods. Rieck et al. [39] propose a malware instruction set approach in which they extract dynamic behavioural features as sequence of instructions. Therefore, *individual execution flows of threads and processes are sequentially appended to a single report* to perform multinomial classification with machine learning techniques. The authors achieve an F-measure of over 0.96% for their method of multinomial malware classification. Nari et al. [40] propose a multinomial malware classification system based on network behaviour of the malware. The malware are classified into their respective family by extracting network flows from network traces gathered in pcap files during malware execution. A behaviour graph to represent network activity and dependencies between network flows is generated. Last, important features are extracted such as *graph size*, *root out-degree*, *average out-degree*, *maximum out-degree [and] number of specific nodes* [41] from the behaviour graphs in order to classify the malware with machine learning algorithms from the Weka library. Pircoveanu et al. [42] are conducting an experiment on classifying malware by their respective type. The authors use Cuckoo Sandbox to execute the malware and derive behavioural features from it and machine learning algorithm Random Forest from the Weka library. DNS requests, accessed files, mutexes, registry keys but mainly API calls from a total sample size of 42,000 malware are used as dynamic features. Random Forest is then classifying the malware into the four

groups of Trojan, PUP, Adware and Rootkit. A weighted average AUC value of 0.98 is achieved.

*Dynamic analysis is more effective as compared to static analysis and does not require the executable to be disassembled* [41]. The following literature presents performance gains of dynamic features in relation to static features for malware category detection. In the above mentioned work from Banin and Dyrkolbotn [34] the authors compare their own results with the results of Shalaginov et al. [43], who are using a static feature extraction method including features from PE32 headers to extract 35 total features from 'PEframe' and 'VirusTotal' as well as a Neuro-Fuzzy method for malware classification. Banin and Dyrkolbotn [34] were able to achieve an overall classification accuracy of 78.4% by using dynamic analysis of memory access patterns while Shalaginov et al. [43] could only reach 39.6% with static features. In the above mentioned paper from Tian et al. [37] they compare their model against other existing work based on a static feature extraction method. For malware detection they achieved an overall accuracy of 97.3% while the compared works only obtain an accuracy of 88%, 93.71% and 95%. In terms of malware classification the authors attained an accuracy of 97.4% compared to similar existing techniques with only 87% and 97% accuracy. Damodaran et al. [44] present a comparison approach based on static and dynamic features as well as a hybrid analysis for multinomial malware family classification. The authors extract opcode sequences and API calls by using a static and a dynamic method. They then train a Hidden Markov Model (HMM) to classify the analysed malware into six families. The resulting scores are plotted as ROC curves and the values of the area under the ROC curve (AUC) are compared. Regarding API calls, the authors achieve an average result of 0.9847 for the dynamic features and 0.924 with their static approach. In terms of opcodes, they obtain average results of 0.905 with dynamic features and 0.7067 by using a static extraction method. The values are AUC-ROC results, which means that the True Positive Rate is plotted against the False Positive Rate and the area under the curve is considered. The higher the value, the higher the True Positive Rate and the lower the False Positive Rate and vice versa. However, the authors mention that it has to be taken into account that obfuscation techniques could have influenced the static feature extraction method. Shijo and Salim [45] examine three different methods for malware classification; static, dynamic and an integrated approach of both. In doing so, they used the two machine learning techniques Support Vector Machine (SVM) and Random Forest to classify the malware into malicious and benign. As static features they extracted printable string information and eliminated the meaningless strings identified by a low occurrence frequency. API call sequences are used as dynamic features and analysed by the n-gram method. The authors analysed 997 malware and 490 clean files for their experiment. Both SVM and Random Forest achieved similar results for its respective feature extraction method which is why the average from both will be presented here. Obtained results with the static approach are 0.9535 True Positive Rate, 0.114 False Positive Rate and an accuracy of 95.36% while dynamic features reached an average accuracy of 96.905% with a True Positive Rate of 0.969 and a False Positive Rate of 0.0995.

To summarise this chapter, Table 1 has been created. It lists the most relevant references for multinomial malware classification with the used features and their extraction technique, the utilised sample size, the applied classification method and the obtained performance.

Reference	Feature Extraction	Sample Size	Features	Classification Method	Performance
Shalaginov et al. [43]	static	400	Static PE32 information	Neuro-Fuzzy	Accuracy: 39.6%
Liu et al. [21]	static	21,740	Opcode n-gram, grey-scale images, import function	Machine learning	Accuracy: 85.3%
Tian et al. [46]	static	1,367	Printable strings	Machine learning	Accuracy: 97%
Tian et al. [47]	static	721	Function length patterns	Machine learning	Accuracy: 87%
Kim et al. [17]	dynamic	2,928	API call sequence alignment and visualisation	Multiple sequence alignment	F-measure: 0.94
Grégio et al. [20]	dynamic	16,248	Instruction sequences	Longest common subsequence	Precision: 0.843
Lee et al. [22]	dynamic	2,639	API behaviour data	Cosine similarity method, n-gram model	Accuracy: 78.25%
Rieck et al. [39]	dynamic	3,133	Instruction sequences	Machine learning	F-measure: 0.96
Banin and Dyrkolbotn [34]	dynamic	983	Memory access patterns	Machine learning	Accuracy: 78.4%
Tian et al. [37]	dynamic	1,824	API call sequences	Data mining, machine learning	Accuracy: 97.4%
Zhao et al. [48]	dynamic	13,223	API calls	Machine learning	Accuracy: 83.3%
Ahmed et al. [49]	dynamic	516	Spatiotemporal information in API calls	Machine learning	Accuracy: 96.3%
Pircoveanu et al. [42]	dynamic	42,000	DNS requests, accessed files, mutexes, registry keys, API calls	Machine learning	AUC-ROC: 0.98
Hansen and Larsen [2]	dynamic	31,295	API calls	Machine learning	F-measure: 0.864; AUC: 0.978

Table 1: Most important references of multinomial malware classification regarding RQ2 and RQ3



In conclusion, most of the recent literature deals with either static features, which can fail (RQ1), or dynamic features from disk and network. Memory features are often not included in multinomial malware classification due to their volatile nature which makes them difficult to obtain. There is also no distinct clarification of which dynamic features are useful for multi-class malware classification (RQ2). Moreover, there is no clear assessment of the performance difference between static and dynamic features (RQ3). This paper works towards a possible solution of those issues in the future. Specific dynamic features, including some selected memory-based features, are extracted and used for classification purposes while static features from the same malware samples are used for a reliable performance comparison.



## 3 Background Literature Study and State of the Art in Malware Analysis

For the following paper it is important that there is a clear understanding of the idea of malware analysis being the foundation of knowledge. Therefore, in this chapter, the most important terms, techniques and methods used in this work are explained in detail including all necessary background information and definitions concerning malware analysis.

### 3.1 Static vs. Dynamic

Current literature describes two approaches to perform malware analysis, static and dynamic [34, 50, 51]. Both types roughly accomplish the same goal of describing how the analysed malware works as well as the needed time and skill. However, the analysing tools used to achieve this goal are quite different from each other [51]. Static malware analysis is an examination method for malicious software without any execution [50]. To accomplish this, static properties are collected such as *bytes, opcodes and API n-grams frequencies, properties of Portable Executable header [and] strings (e.g. commandline commands, URLs etc)* [34]. Moreover, a code analysis can be performed by actually viewing the malicious code with the help of disassemblers and decompilers to gain a better comprehension of the malware functionalities [51]. Static malware analysis is often commonly referred to as signature-based malware detection in which a cryptographic hash value or checksum is calculated and compared to existing data, an approach used by AV-vendors [2]. Dynamic analysis, also called behavioural analysis, describes the process of executing the malware in a safe and controlled environment, like a virtual machine or a specialised sandbox such as Cuckoo. While executing the malware, the malicious activities are being captured which include *patterns of a registry, network and disk usage, monitoring of API-calls, tracing of executed instructions, investigation of memory layout and so on* [34]. Thus, any addition, deletion and modification of files, services, processes, registries and system settings can be identified as well as unusual network traffic. Furthermore, the lab environment should never be connected to any other network during dynamic malware analysis and files should be transferred with read-only access [51].

In real life scenarios static and dynamic malware analysis are often both used in combination, a so-called hybrid technique [41]. However, static and dynamic analysis can also be done fully automatic to generate information about the analysed malware. In real case scenarios a human analyst will use the automatically gathered data to perform a manual analysis on top. This method and some example tools are shown in Fig. 3. [4]

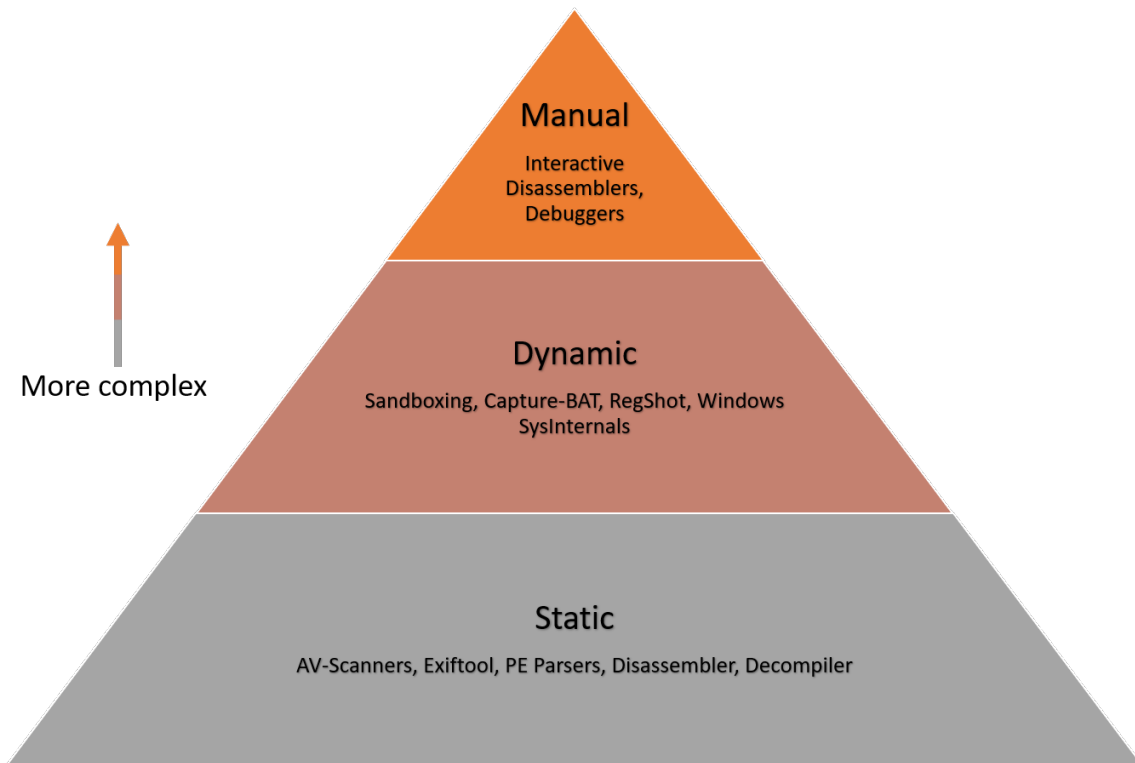


Figure 3: Malware analysis pyramid based on Pektaş [4]

### 3.2 Malware Classification

Malware evolves and increases in number each year of both new malware and variants of already known malware. This makes former state-of-the-art binary classification method, in which malware is classified into benign and malicious, obsolete. Moreover, the large variety of cyberthreats also increases the number of complex and encapsulated malware. To cope with the challenge of growing and more complex and demanding malware, multinomial malware classification is presently used. Malware is classified in more than two categories, often malware types and malware families are used as reference model. [38]

### 3.3 Malware Landscape

*In order to provide better understanding of malware capabilities, describe vulnerabilities of systems and operations as well as to use appropriate protection and post-attack actions [34], it is important to not only detect malware but also to classify them based on their functionalities. In past research articles malware classification has often been an issue for the authors due to wrongly used terminology. Commonly in use are two widely accepted malware categorisation approaches; malware types and malware families. However, proper definitions are rarely provided by authors as literature studies*

indicate. *This can lead to the various misunderstandings and non-valid comparisons* [34]. Banin and Dyrkolbotn [34] present three scientific papers from 2008, 2009 and 2013 in which the authors either wrongly or inconsistently use the terminologies as well as making a non-valid comparison as stated above by comparing malware types to malware families and vice versa. In this work Banin's and Dyrkolbotn's [34] definition of malware types and malware families is used. It is based on well known vendors such as Microsoft and Symantec and their description of malware categories.

### 3.3.1 Malware Types

Malware types, also called malware categories in some literature, can be seen as the generic term. It describes the general functionality of a malware or simply explains what malware does and what goals it pursues. *Worms, viruses, trojans, backdoors, and ransomware are some of the most common types of malware* [5].

### 3.3.2 Malware Families

Malware families is a grouping based on its particular functionality and their common characteristics or simply described how malware acts and which methods are used to achieve its goals. For example, a malware could be of type backdoor, which means that the overall goal of the malware is to create a backdoor in the target system. The way it achieves that goal, the actual code it uses to do that, is considered in the malware family. This can be similar across different malware because either code snippets are being reused by malware authors but also *as malware evolves, some semantics of the original malware are preserved as these semantics are necessary for the effectiveness of the malware* [24].

### 3.3.3 Malware Platforms

Malware can be written in different scripting/programming languages for various operating systems [5]. The most common operating system malware is produced for, is the Windows 32-bit platform. Windows is the most widespread OS and with implemented backwards compatibility it is a familiar target for malware authors to attack.

## 3.4 Evasion Techniques

Even the most powerful malware in terms of functionality is useless if it gets detected the very first moment it executes its malicious actions or even before it can copy itself to the system. Therefore, malware authors try to make their malware as unnoticeable to the victim as possible. Thus, the malware can perform its malicious behaviour undetected for a long period of time. Such a method is called evasion technique because the malware tries to evade detection. For automated malware detection and classification systems it can be a great challenge to cope with such evasion techniques. Especially because it is also possible for malware to use different evasion techniques concurrently [27]. The specific techniques used to hide the malicious nature of a malware to avoid detection can differ depending on the malware author and his intentions, the victim's system and the malware functionalities.

### 3.4.1 Obfuscation

Obfuscation techniques aim to change the malware code in a way that its either not possible anymore to determine a correlation to other malware or to make conventional detection methods unable to find malicious indicators. There are a lot of different obfuscation techniques which can not all be covered in this chapter but amongst the most known ones are approaches to change the order of instructions, insert garbage code like `NOP`-commands or fake instructions, replace commands with equivalent ones or rename variables and registers [27, 28, 52] in order to make basic signature-based detection nearly impossible or to increase analysis time. A *static feature approach can be easily bypassed by obfuscation methods* [26]. Alternate approaches of obfuscation are the usage of encryption, encoding or compressing/packing techniques with examples such as XOR encryption, base64 encoding and the use of packers like `UPX`. Thereby, the intention of the authors isn't long term security but rather to remain undetected [25] through transmutation of the code by removing any readable strings.

### 3.4.2 Anti-\*

As already stated in Section 3.1, dynamic analysis is done by executing the malicious files in a controlled environment like a virtual machine. Moreover, malware analysts use particularly designed tools and software to work with. This is of course a commonly known fact, also to malware authors. Special evasion techniques have been developed to detect such proof of ongoing analysis. The malware checks for certain indicators which suggest that the malware is being examined by an analyst. The malware then tries to either hide its malicious behaviour by not executing the particular code or even erases itself from the hard disk. Anti-VM techniques are used to detect registry keys, installed tools, processes and services, serial numbers or `MAC` addresses, system memory structure and hardware parameter related to virtual environments indicating that the malware is being executed in a sandbox [28, 53]. The malware could also try to crash [25] or infect [54] the sandbox once recognised. This, however, is not realistic because a malware's goal is to remain hidden [25]. Another popular evasion technique among malware authors is anti-tools. The malware checks for indication that commonly used analysis software such as the monitoring tools Wireshark or Process Explorer are installed on the system or running as process [53]. The ransomware/cryptominer Rakhni for instance, *has a list of more than [sic] 150 names of tools used for process monitoring and analysis; if one of the running processes is in that list, Rakhni will hide its malicious behavior* [25]; [Supposed to be "than"]. Anti-Debugging techniques are used for detecting present Debuggers, a software to diagnose and locate errors in computer systems, used to give full control to the malware analyst over the run-time behaviour of the analysed malware. Malware can detect if it is executed in debug-mode in different ways. An easy approach is to check if the `Windows API IsDebuggerPresent()` is invoked [55]. Malware can also detect set flags in the Process Environment Block fields or the Heap fields to detect debuggers [56]. A third commonly used method is to check for execution time [56]. While being debugged the malware reaches certain functions much later than expected.

### 3.4.3 Polymorphic & Metamorphic Malware

As already stated in Subsection 3.4.1, encryption is a viable evasion technique used by malware authors to avoid detection. The malware typically consists of the encrypted payload and the decryptor recovering the payload during run time. By using a different key for each infection, the malware ensures a different payload signature. However, the constant decryptor makes this approach unsuitable on the long run. Polymorphic malware, on the contrary, uses the obfuscation techniques mentioned above in Subsection 3.4.1 to create innumerable amount of various decryptors. Thus, it addresses the issue of the encryption approach being unable to avoid signature-based detection because of its constant decryptor. Moreover, tools such as The Mutation Engine, DAME and TPE [57] exist to help malware authors to transform a non-obfuscated malware into a polymorph without any considerable expenditure. [52, 58]

Metamorphic malware uses obfuscation techniques to mutate itself in order to produce malware variants [59] without sacrificing functionality [58]. While Polymorphic malware have similar memory indicators and use traditional malware elements for encryption, metamorphic malware vary in memory for each variant and use different encryption elements [58].

## 3.5 Malware Naming Issue

A huge challenge concerning protection against malware is the absence of an agreement on naming conventions for malware between different anti-virus vendors, analysts and researchers. This issue can lead to confusion as well as causing some major difficulties when security analysts have to rely on them to simply compare or correlate viruses or to build reference datasets [60, 61]. Kelchner [62] declares *historical reasons and vendor-specific policies [as causes why] malware naming has never followed any conventions* [60]. Although there have been approaches of proposing standardised naming conventions, they mostly failed to be widely adopted among groups of interest in the past [60, 61, 63].

CARO, which is short for Computer Antivirus Research Organization, is a group of individuals to study malware since 1990 [63]. In 1991, CARO published their first approach of a malware naming convention to reduce the naming confusion [64]. This first approach by Fridrik Skulason (Virus Bulletin's technical editor), Alan Solomon (S&S International) and Vesselin Bontchev (University of Hamburg) was ignored by the malware community and anti-virus vendors until it got some subsequent revisions over time [65]. This concept recently gained an increase in popularity as it is used by companies such as Microsoft [5] and Trend Micro [66] as shown in Fig. 4. The Type and Family attributes are used to describe malware functionalities as described in previous sections. The Platform attribute indicates the operating system, the programming language and the file formats. The Variant letter is *used sequentially for every distinct version of a malware family* and the Suffixes provide optional information about the malware [5]. However, the delimiters can be selected individually from the set of available symbols: [!#.@/:] [65] and can therefore differ between different practitioners.

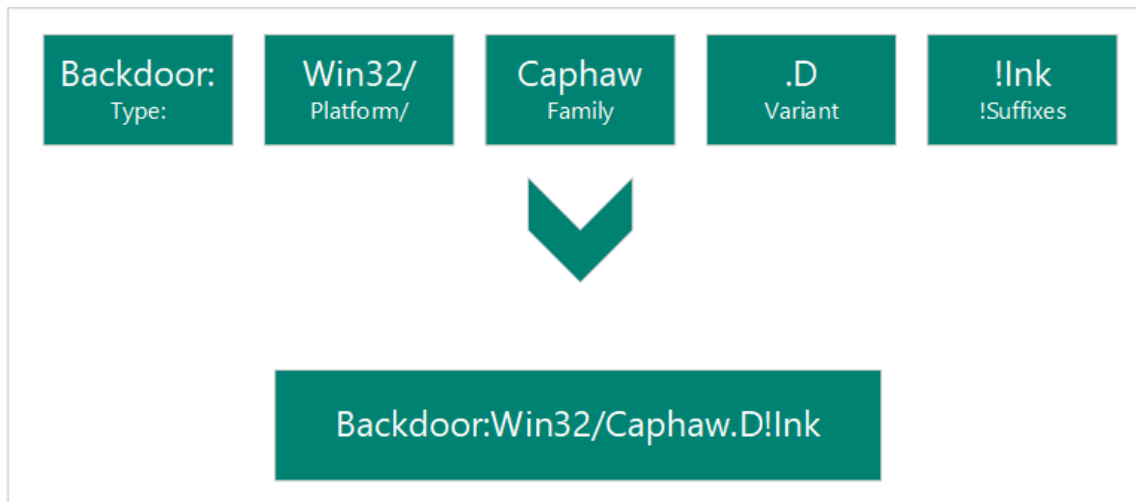


Figure 4: Computer Antivirus Research Organization malware naming scheme used by Microsoft [5]

### 3.6 Machine Learning

Machine learning (ML) is used today in many different scientific areas. It can be seen as small subset of artificial intelligence. It is the generic term for artificial generation of knowledge from experience. ML algorithms are used to learn from examples of sample data, also known as training data, to build a mathematical model. After completion of the training phase the model can make predictions or decisions on the test set by recognising patterns and regularities without ever learning the sample data by heart. This allows the system to also evaluate previously unknown data. *The types of machine learning algorithms differ in their approach, the type of data they input and output, and the type of task or problem that they are intended to solve.* The most common ones are supervised and unsupervised learning and the main difference is that unsupervised learning algorithms learn from unlabelled, unclassified or uncategorised test data, data that contains only inputs. In contrast, supervised learning uses *data that contains both the inputs and the desired outputs*. Moreover, various classifiers can be applied to machine learning algorithms to build a classification model. [67]

The most common classifiers, indicated by Wikipedia [67] and used by Banin and Dyrkolbotn [34], for malware classification are:

- **Naive Bayes (NB)** is a probabilistic classifier inspired by the Bayes theorem under a simple assumption which is the attributes are conditionally independent [68]. The Bayes's theorem is a mathematical proposition from probability theory that describes the calculation of conditional probabilities [69]. Naive Bayes is a rather simple algorithm but since it takes linear time, it can be scalable to large datasets as well. However, it can not give valid prediction when the conditional probability is zero [68].
- **Support Vector Machine (SVM)** is a powerful classifier based on Vapnik's theory, which belongs to the computational learning theory trying to explain the learning process with statis-



tics [70]. It has strong data regularisation properties and can handle big data sets. It is based on the Structural Risk Minimization (SRM) principle, to find an optimal hyperplane by maximizing the margins that can guarantee the lowest true error due to increasing the generalization capabilities. [46]

- **Artificial Neural Networks (ANN)** is a set of connected input/output units where each connection has a weight associated with it [...]. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples [68]. There are different network architectures available such as Feed-forward, Recurrent and Convolutional. Which architecture to use depends on the model. There can be several hidden layers in the model which will increase the mapped complexity but also the time performance. ANN are tolerant of noisy data and can classify untrained patterns. [68]
- **k-Nearest Neighbours (kNN)** belongs to the lazy learning algorithms. All instances corresponding to training data points are stored in a n-dimensional space. When an unknown discrete data is received, it analyzes the closest k number of instances saved (nearest neighbors) and returns the most common class as the prediction and for real-valued data it returns the mean of k nearest neighbors [68]. kNN is usually resistant to noisy data because of the averaging of k-nearest neighbours. [68]
- **Logistic Regression (LR)** is a classifier from the field of statistics and is commonly used for binary classification. At the core of Logistic Regression it is based on the logistic function which is used to describe the properties of population growth ( $f(x) = \frac{1}{1+e^{-x}}$ ). Input values are linearly combined by the usage of coefficient values or weights. The output value is then being modeled as binary value. [71]
- **J48** is a java implementation of the C4.5 algorithm in Weka used to build decision trees from a set of training data by using the concept on information entropy. At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other [72]. The normalised information gain is used as splitting criterion. The decision is made by choosing the attribute possessing the highest normalised information gain. [72]
- **Random Forest (RF)** is based on a combination of many decision tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [46]. It is efficient for large data sets and it maintains accuracy even if data is missing. It generates estimations of errors during the forest building process and of the important variables for the classification. Random Forest is also known to have a good accuracy among the current classifiers. [46]



## 4 Methodology

In the following chapter the used methods of the experiment are described in detail. As shown by the references in Chapter 2 concerning RQ1 there are cases, in which conventional signature-based malware detection and ML-aided detection with the help of static features fail. Thus, dynamic features from disk, memory and network are used to overcome some of those limitations of static malware analysis. As long as the overall functionalities of the malware stay the same, the dynamic indicators of compromise observed on the system and network are highly alike. This means that dynamic features are resistant to evasion techniques, presented in Section 3.4, to a certain point. As long as the malware can be executed in a sandbox, most of the known obfuscation and metamorphism techniques do not hinder the analysis because the behaviour of malware will be nearly consistent even if the source code is obfuscated.

### 4.1 Data Set

To have a useful data set is one of the most important aspects in the experiment. If the data set is too small or not distinct enough the outcome of the experiment will not be valid. In order to test the classification algorithms later on, the malware samples already have to be classified in their respective families. The top 10 most frequent labelled malware categories, as indicated by Shalaginov and Franke [38], are being used. This includes malware from the following families:

- **Agent** is a large malware family usually associated with the installation of Adware but it can also download additional malware and even change some Windows configuration settings. [73]
- **Hupigon** is a malware family commonly known as Remote Access Tool (RAT) opening a backdoor, by registering itself as a service, the perpetrator can use to control the compromised machine. [2]
- **Obfuscator** is a generic term for malware trying to obfuscate itself to avoid detection as explained in Subsection 3.4.1.
- **Onlinegames** belongs to the publicly known Trojans family. It usually downloads and drops additional malware on the infected machine as well as collects online game key strokes. [2]
- **Renos** is a malware family that mostly shows fake security warnings to *trick a computer user to download third-party cleaning utilities* [73].
- **Small** is a malware family also of the Trojan-type connecting to servers to download additional unwanted software without the users consent. [2]
- **Vb** is a generic term for malware written in Visual Basic. [73]
- **Vbinject** is a malware family also written in Visual Basic but it conceals malware inside.
- **Vundo** is another family of Trojans also associated with Adware but especially for pop-up

advertisements. [73]

- **Zlob** includes multiple components such as *modification of Internet Explorer's settings, altering of the user's default Internet search page and home page and also tries to download and run other malicious programs* [2].

## 4.2 Data Set processing and analysis

A lot of recent research concerning malware classification are doing a binary classification into malicious and benign to detect malware [2, 17, 22] before actual conducting a multi-class classification approach. Such a malware detection approach will not be done in this work because the scope of this elaboration is multinomial malware classification. Therefore, every analysed malware is assumed to already be identified as malware by some kind of detection system. The family labels mentioned above in Section 4.1 were retrieved from Microsoft using the **CARO** naming scheme, explained in Subsection 3.5, for malware which were positively identified by VirusTotal. Moreover, malware with anti-VM or anti-debug features will be removed from the actual dynamic analysis in a pre-processing phase because those samples could heavily skew the outcome of the experiment.

Behavioural indicators of compromise are extracted, generated by a dynamic malware analysis. Based on those characteristics, the malware samples are being classified into groups concerning to their similarities. The experiment will be conducted with the help of the existing open source tools Cuckoo Sandbox as dynamic analysis system and machine learning algorithms from Weka library. Cuckoo Sandbox offers a controlled environment, also referred to as sandbox, in which the malware can safely be executed. After each execution of malware the features will be extracted, as explained in more detail in Section 4.3, and the virtual machine will get reverted to a clean state, a snapshot. According to this methodology, every malware sample will be analysed and all the features mentioned in Section 4.3 will be extracted. Afterwards, the results have to be prepared for the classification. In this phase, the malware will be classified based on the extracted characteristics with the help of machine learning algorithms. To achieve this, the machine learning model is trained with a part of the behavioural features obtained from the dynamic analysis. Afterwards, the remaining data is used to test the accuracy of the algorithm. More detailed information about the classification method and machine learning algorithms are found in Section 4.5. A basic process flowchart of the method of the experiment is visualised in Fig. 5.

In addition to the actual dynamic experiment, a second experiment is performed concerning RQ3. It has to be analysed if the dynamic features actually achieve better results and if so, the performance gain of the dynamic features compared to the static features has to be examined. Therefore, static features, extracted from PEframe and Linux tools, are used for classification with the same machine learning algorithms. The used static features are:

```
pe_api, pe_debug, pe_packer, pe_library, pe_autogen, pe_object,
pe_executable, pe_text, pe_binary, pe_temporary, pe_database,
pe_log, pe_webpage, pe_backup, pe_cabinet, pe_data, pe_registry,
pe_directories, pe_dll, pe_detected, size_TEXT, size_DATA,
size_OBJ, size_TOT, filesize
```

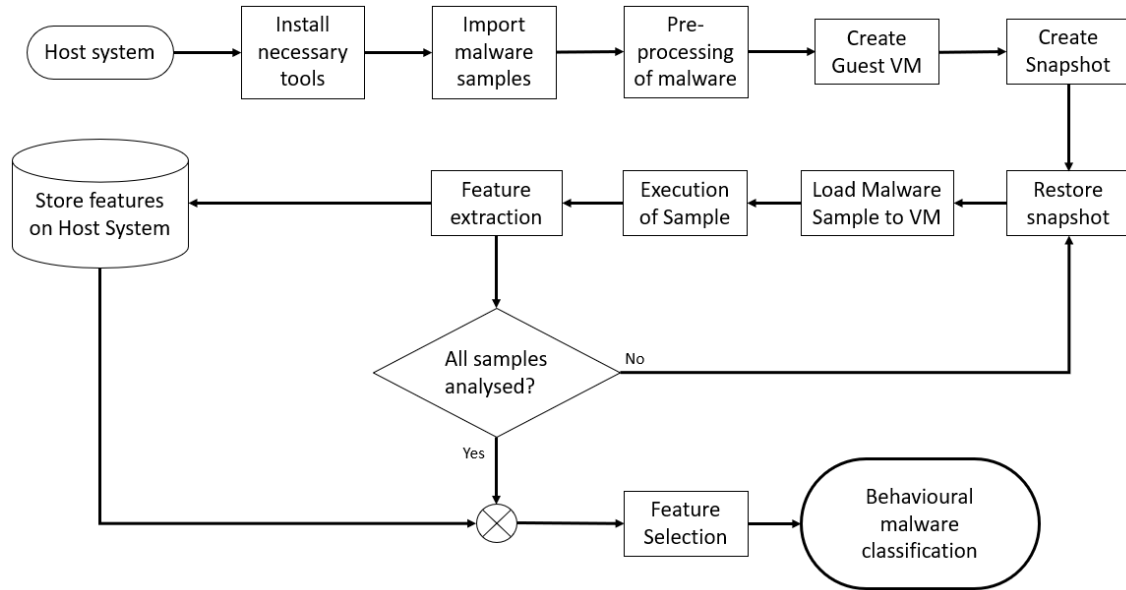


Figure 5: Process flowchart of the malware analysis process during the experiment based on Banin et al. [6]

The detailed description of those static features is given by Shalaginov et al. [43] and Grini et al. [74]. In their study the authors used features from PEframe and Linux-based command line tools but also static-based features extracted from VirusTotal. Since it can not be ruled out that the features from VirusTotal might have been created with additional intelligent pre-processing, they are excluded from the dataset used in the second experiment. This is done in order to guarantee an unbiased comparison between static and dynamic features extracted in an experiment of the same tool-based level.

### 4.3 Feature Extraction

The extracted features will influence the outcome of the experiment the most. Thus, the fact about which feature to extract is one of the most important question to answer. This work will only look at behavioural malware features extracted by dynamic analysis. This is done in order to mitigate code obfuscation and other static-based evasion techniques as presented in Section 3.4 and because dynamic features provide a complete picture of the whole execution process of a malware. Therefore, dynamic features from disk activities and network traffic as well as memory footprints are obtained. Which individual features and which combinations of features will produce the best results will be analysed in the experiment. The reasons behind the chosen features are the already implemented support of extraction by the used malware analysis system, the fact that they are based on malware functionality and also previous literature successfully conducting experiments based on those features. However, in contrary to what was planned at the beginning, it was not further pursued

to extract full memory dumps of the analysis machine. Even though it would produce more reliable results, the results would also be more vague and the analysis itself would take much longer. The following dynamic behavioural features are extracted in a contemporary Windows malware analysis in order to answer the question of which features can be successfully used in multinomial classification (RQ2):

- **Disk activities** are probably the largest category but also the easiest obtainable features. Low-level file operations are extracted, which includes any kind of file modification on the accessible disk storage such as reading, writing, deletion or other modifying actions done by the malware. This also involves new files dropped to the system by the malware. Furthermore, registry patterns, describing changes of the operating system (OS) configuration database [38], are examined. This includes access of specific registries, registry keys read, modified or deleted and new keys or values added to the registry. Moreover, dynamic-link libraries (DLL), loaded by the malware processes, are extracted. In addition, mutexes on the file system are analysed and recorded since this is a common approach by malware to lock access on specific resources but also to avoid reinfecting the same host again.
- **Network traffic** includes all data flowing through the network. The whole network traffic produced by the malware is monitored, stored as `pcap` file and the relevant network information, such as DNS traffic, IRC and SMTP traffic, domains, IPs and HTTP as well as SSL/TLS encrypted HTTPS requests are extracted. [7].
- **Memory footprints** are the hardest to obtain and to analyse because of their volatile nature. Moreover, a lot of literature indicates that memory analysis is not very trustworthy without ground-truth and it often shows reduced accuracy, as indicated by Shalaginov and Franke [38], which makes memory patterns a possibly unreliable feature for malware classification. To examine whether memory footprints can still be used as dynamic feature for multinomial malware classification they are included in this work nonetheless. Therefore, behavioural metrics, such as average CPU and memory usage as well as the peak usage of those resources, are measured.

#### 4.4 Feature Selection

All the behavioural features extracted in Section 4.3 are then used to classify the malware into their respective families. Therefore, the extracted features are fed to machine learning algorithms. The main work of the classification is done by such machine learning algorithms and will be conducted with the library called Weka providing machine learning implementation. But Weka also provides the ability to rank features and find feature combinations. The most common metrics for feature selection (FS) methods are:

- **Correlation** calculates a correlation between each attribute and the output variable.
- **Information Gain**, typically referred to as Entropy, calculates the information gain for each attribute for the output variable.
- **Learner** calculates the feature subset producing the best performance which is then taken as

subset.

With feature selection it can be determined which features and which feature combinations are the most suitable for multinomial malware classification, discussed on the basis of RQ2. [75]

## 4.5 ML-aided Malware Classification

After the feature selection is completed, different classifiers suitable for malware classification can be tested with Weka. The commonly used classifiers for malware analysis are listed in detail in Section 3.6. Those six classifiers will also be used in the experimental setup of this thesis. As mentioned in Section 4.1, the malware was already pre-classified. Therefore, supervised learning can be used to conveniently classify the malware samples. As test method of the data, a 10 fold cross-validation approach is utilised. With cross-validation the original sample size is divided into  $k$  subsamples of equal size, referred to as folds. From those randomly partitioned subsamples one subsample will be used as testing data, while  $k-1$  samples are taken to train the machine learning algorithm. This process is then repeated  $k$  times with a different subsample as test data each iteration. With this method the whole data set is being used for both training and validation and each subsample is used exactly once for testing. The  $k$  results can then be averaged to produce a single estimation. Moreover, the outcome of the dynamic feature classification can be examined against the results of the static feature classification. This provides the ability of testing the performance gain of comprehensive dynamic features (RQ3) for each classifier.





## 5 Experimental Setup

The following chapter introduces the actual implementation of the experiment based on the methodology presented in Chapter 4. The collection of the data set, the software versions, the hardware specifications and the used algorithms during the experiment are presented as well as the experimental design and the technical implementation. It demonstrates the technical execution of the methodology for reproducibility of the use case.

### 5.1 Data Set Collection

For the experiment a data set of a total number of 9,823 Windows PE32 malware samples from the top 10 most frequent labelled malware categories, as listed in Section 4.1, are being used. The data set was provided by the NTNU Malwarelab but originally they were retrieved by Maltrieve [76] crawling seven different sources and collected from VirusShare [77] and VxHeaven [78].

### 5.2 Software Versions

Open source software has been used in order to conduct the experiment. Table 2 lists all the important software with the specific used version during the experiment. However, software which is already included in Cuckoo Sandbox or provided by the system as internal tools are not listed.

### 5.3 Hardware Specifications

Dynamic malware analysis can be very resource demanding, especially when a lot of samples are analysed in a short period of time. To be able to compete with the massive amount of malware each day, a suitable amount of resources has to be available. The host system used for the experiment consisted of the Intel® Xeon® CPU E5-2630 with 2.4GHz and four cores, as well as 16GB of RAM and disk space of 500GB. The invoked guest system for the dynamic analysis of the malware consisted of 8GB of RAM, 100GB disk space and one core from the host system since the experimental setup, which was implemented as nested virtualisation, didn't allow for more cores.

### 5.4 Experimental Design and Implementation

The experiment starts with the initial pre-processing of the malware with the PEframe tool. The remaining malware samples are then analysed with Cuckoo Sandbox but before that, the automated analysing system has to be configured accordingly. During the analysis the dynamic features of each malware sample is extracted and afterwards, prepared for processing of the machine learning library Weka. There, the malware samples are classified in their respective families with the help of machine learning applications. Fig. 6 visualises the different steps of the experimental setup

Software	Version	Release Date	Description
Ubuntu	18.04.1 LTS	July, 2018	Operating System
Windows	7 Ultimate x86	February, 2011	Operating System
Cuckoo Sandbox	2.0.6	June, 2018	Malware Analysis System
Oracle VM VirtualBox	6.0.4	January, 2019	Virtualisation Software
INetSim	1.2.8	June, 2018	Fake Internet Services
Python	2.7.15	May, 2018	Programming Language
Tcpdump	4.9.2	September, 2017	Packet Analyser
PEframe	5.0.1	Not Defined	Static Analyser for PEs
mitmproxy	v4.0	May, 2018	HTTPS proxy
Microsoft Office	2013	January, 2013	Productivity Software
Adobe Acrobat Reader DC	2019.010.20098	February, 2019	PDF Reader

Table 2: Software used in the Experiment

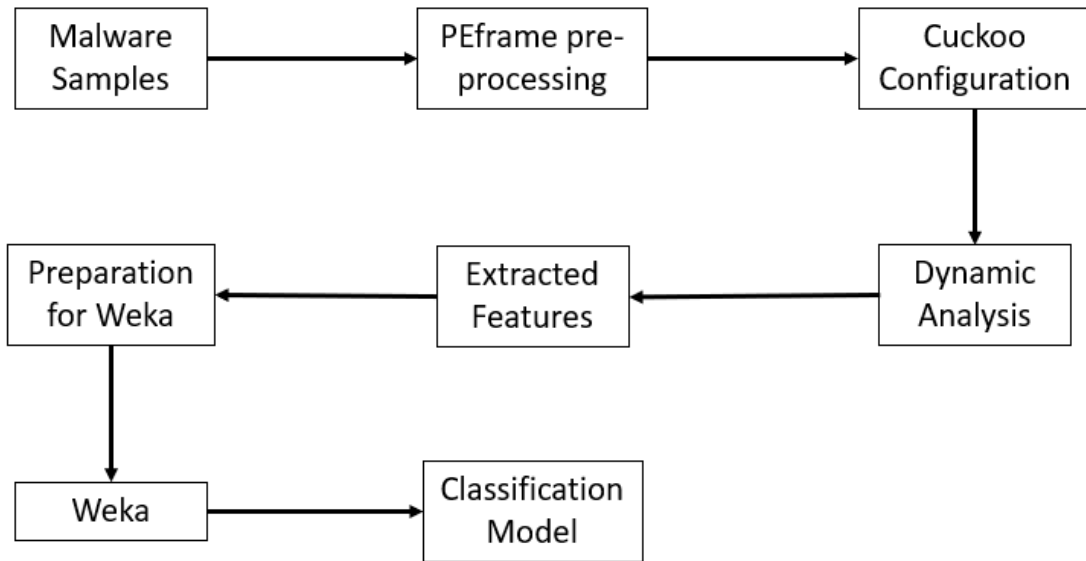


Figure 6: Different Steps of the Experimental Setup

#### 5.4.1 Malware Pre-processing

As first step every malware sample has been analysed with the static analysis tool PEframe. As already mentioned in Subsection 3.4.2, more sophisticated malware are using features to identify the act of being analysed. For example, indications of a debugger or execution inside a virtual machine can be recognised by such malware. However, since malware with such features would skew the analysis results, they will be excluded beforehand. A simple Python script in combination

with 'PEframe' statically analysed every malware sample and the ones with anti-debug and anti-VM features have been removed. An excerpt of the used Python script can be seen in Algorithm 1 shown as pseudocode. The full Python code is listed in Appendix A.1.

---

**Algorithm 1** Pseudocode for static analysis of malware samples with PEframe

---

```

for  $f \leftarrow files$  do
    PEframe( $f$ )
    if ( $key = "antivm"$  OR  $key = "antidbg"$ ) AND  $value \neq 0$  then
        | delete( $f$ )
    end
end

```

---

#### 5.4.2 Sandbox-related Configuration Details

As experimental environment a cloud-based virtual machine running the Ubuntu operating system has been set up. Its hardware specifications can be seen in Section 5.3. The aforementioned Cuckoo Sandbox was installed onto that system alongside other tools. A full list of all used tools can be found in Table 2. Cuckoo Sandbox has to be configured according to its documentation [7] alongside some small customisations. Thus, a Windows 7 virtual machine with the virtualisation software VirtualBox from Oracle is set up and configured. Windows 7 was chosen, because it is the best supported operating system by Cuckoo Sandbox as recommended by their manual [7]. A virtual network, attached to the host OS as 'Host-only Adapter', between host and guest system is used in order to make it as hard as possible for the malware to escape its controlled environment. Consequently, no tools, such as 'VirtualBox Guest Additions', are installed that could allow the malware to leak potentially harmful code to the host system. Apart from the network configuration, the guest has to be prepared in different aspects. The programming language Python has to be installed and some noise [79] produced by default Windows processes need to be removed. Moreover, built-in Windows security features, such as the Windows firewall, anti-virus scanners and automatic updates, have to be switched off to make the system as vulnerable as possible. Furthermore, in case of malware having different embedded files, Microsoft Office and Adobe Acrobat Reader have been installed to provide the ability of executing such file types. In addition, a powershell script, listed in Appendix A.2, is saved on the guest machine which is being used to extract memory features such as CPU and memory usage during the malware execution. Lastly, a snapshot of the machine state must be created. On the host system there are several tools to provide additional functionality for the malware analysis such as INetSim, Tcpdump and mitmproxy. Malware often requires an Internet connection to function properly and will therefore mostly not run when disconnected. To cope with this, a fake internet connection with INetSim is set up on the host system communicating with the malware during execution by providing simulation of common internet services. Tcpdump is then used to dump such network behaviour of the malware. In case of a malware performing SSL

encrypted requests over [HTTPS](#), the tool mitmproxy is used. This proxy pretends to the server and client to be its counterpart in the form of a Man-in-the-Middle. Furthermore, as already mentioned above, Cuckoo was customised to run the above mentioned powershell script, listed in [Appendix A.2](#), together with every malware execution. Therefore, the default package for analysing Windows executables [7] has been modified with the following lines of code:

---

**Algorithm 2** Modified default Python script for analysing Windows executables used by Cuckoo

---

```
def start(self, path):
    args = ["-ExecutionPolicy", "ByPass", "-File",
    ↪ "C:\Users\Win7\Downloads\CpuMemUsage.ps1"]

    self.execute("powershell.exe", args=args, trigger="file:%s" %
    ↪ "C:\Users\Win7\Downloads\CpuMemUsage.ps1")

def finish(self):
    self.send("C:\Users\Win7\Desktop\usage.txt")
```

---

Although every malware with anti-debug and anti-VM features has been preemptively removed in the pre-processing phase (see Subsection 5.4.1), Cuckoo Sandbox has some built-in features to cope with different, more unusual approaches of malware recognising the fact of being in an automated analysing system. For example is Cuckoo, amongst others, mimicking human activity with random mouse movements and clicks.

Fig. 7 shows a simplified model of the Cuckoo architecture used in the experiment with the small adaption that the Cuckoo host is running on a cloud service and only one analysis guest machine is used.

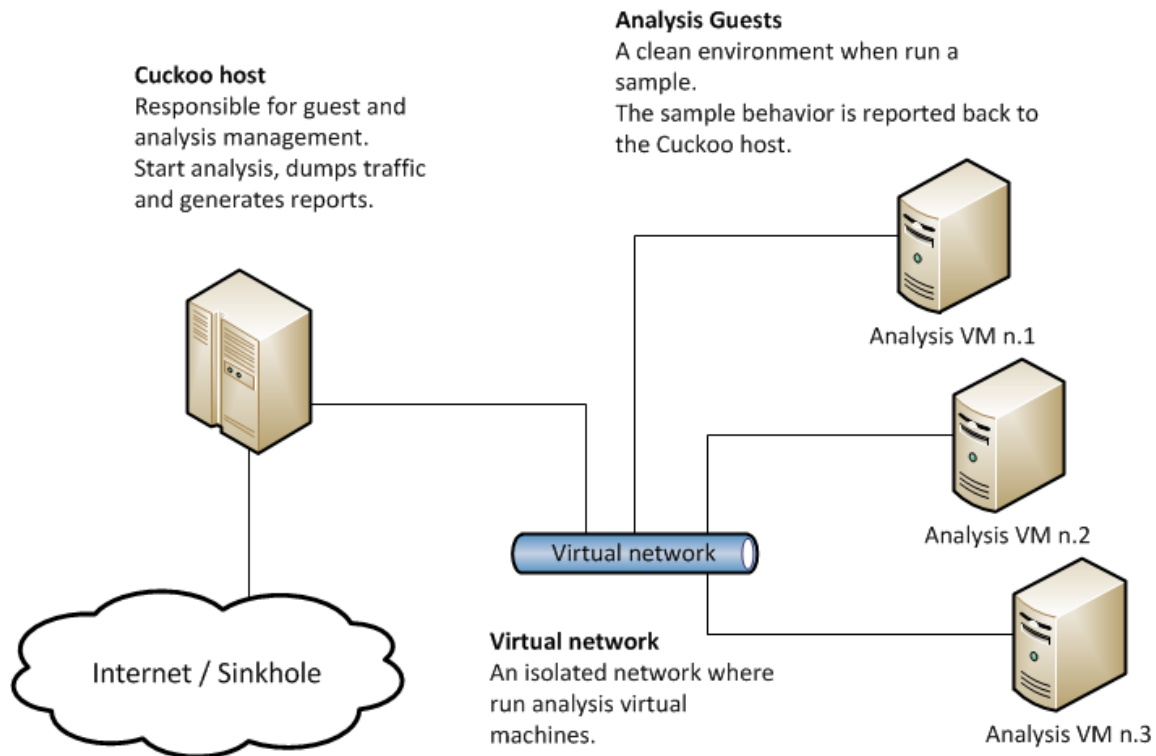


Figure 7: Cuckoo's main architecture [7]

### 5.4.3 Dynamic Malware Analysis and Feature Extraction

The remaining 8,305 pre-processed malware samples are then submitted to cuckoo for the dynamic analysis. Cuckoo assigns every sample with a unique task ID and then invokes the guest machine for the analysis over the configured virtual network and reverts it to the previously created snapshot. A Cuckoo agent, programmed in Python, that works over the network is used to transfer data between host and guest by using the XMLRPC protocol. A malware sample is transferred to the guest machine. In this controlled environment the sample is executed alongside the powershell script. During the execution, Cuckoo examines all changes made to the system or connection attempts to the outside world while the powershell analysis the CPU and RAM usage. Once the initial malware process and every process invoked by the malware is terminated, the analysis is considered to be finished. In case a malware tries to stay active and never terminates its processes, Cuckoo has a built-in 60 seconds timer. If this timer is reached, the analysis is being forced to terminate. Once Cuckoo is done analysing a sample, the results are being extracted to the host system with the Cuckoo agent and reported by different reporting modules. Various logs, reports and dumps are generated and combined in a JSON-format report. The usage.txt file, created by the powershell script, is also transferred to the host system. Cuckoo creates a directory structure with one folder

for each analysed malware sample on the host system.

Since 8,305 malware samples have been dynamically analysed, some minor errors are to be expected. Therefore, all report files are checked for errors produced by Cuckoo during the analysis and for a missing 'summary' field in the report. Cuckoo uses a 'debug' array with an 'errors' field, which is either empty or not, inside the [JSON](#)-structure. If this 'errors' field has any value inside or the 'summary' field is missing, the report is being deleted since it could skew the outcome of the experiment. Moreover, the results from the powershell script have to be appended to the Cuckoo report file in order to be easily extractable afterwards. A Python script takes over the work of iterating through the whole directory structure made by Cuckoo, finding and deleting the reports with errors or missing 'summary' field, copying the powershell results into the [JSON](#)-report structure and grouping all reports by their malware family based on the task [IDs](#) previously given by Cuckoo. The pseudocode of [Algorithm 3](#) shows the basic functionality of the Python script while the full code can be found in [Appendix A.3](#). [Table 3](#) lists the exact distribution of the individual families before and after PEframe processing and error removal.

**Algorithm 3** Pseudocode to remove errors, copy memory features into report and group all reports based on family

---

```
for  $i \leftarrow 1$  to 8306 do
  if  $\text{len}(\text{"errors"}) \neq 0$  or  $\text{len}(\text{"summary"}) = 0$  then
    delete(report(i))
  next(i)
end
for  $f \leftarrow \text{files}$  do
  if  $f.\text{endswidth}(\text{"usage.txt"})$  then
    for  $l \leftarrow \text{lines}$  do
      if  $l.\text{startswith}(\text{"Memory:"})$  then
        ram.append( $l[8:]$ )
      end
      if  $l.\text{startswith}(\text{"CPU:"})$  then
        cpu.append( $l[5:]$ )
      end
    end
     $\text{newDict} \leftarrow \{\text{peakCPU}, \text{peakRAM}, \text{averageCPU}, \text{averageRAM}\}$ 
     $\text{report}[\text{'summary'}].\text{update}(\text{newDict})$ 
    save(report(i))
  end
end
for  $x \leftarrow 1$  to 10 do
  if  $i \geq \text{folder}[x][\text{startTaskID}]$  and  $i < \text{folder}[x][\text{endTaskID}]$  then
    copy(report(i),  $\text{folder}[x][\text{familyName}]$ )
  end
end
end
```

---

<b>Familie</b>	<b>Original Amount</b>	<b>after PEframe (1)</b>	<b>after Error Removal (3)</b>
agent	1,000	866	600
hupigon	1,000	307	260
obfuscator	1,000	458	417
onlinegames	1,000	989	928
renos	1,000	986	775
small	1,000	983	869
vb	1,000	983	940
vbinject	1,000	949	921
vundo	823	784	323
zlob	1,000	1,000	976
<b>total</b>	<b>9,823</b>	<b>8,305</b>	<b>7,009</b>

Table 3: In the experiment used malware families with accurate amount distribution

#### 5.4.4 Feature preparation for Weka

The [JSON](#)-reports created by Cuckoo are not readable for Weka without further modifications. Weka expects a specific file structure, such as [ARFF](#), [JSON](#), or [CSV](#), consisting of an attribute section and a data section. The attribute section contains one entry for each feature, the data section contains one entry for each malware sample and one value linked to each attribute. To check which features were found, all disk and network feature names detected by Cuckoo, listed in [Table 4](#), as well as the appended memory feature from the powershell script, are extracted. To handle duplicates, only new found feature names are recorded.



Disk Features	Network Features	Memory Features
command_line	connects_host	averageCPU
directory_created	connects_ip	averageRAM
directory_enumerated	downloads_file	peakCPU
directory_removed	fetches_url	peakRAM
dll_loaded	resolves_host	
regkey_deleted		
regkey_opened		
regkey_read		
regkey_written		
file_copied		
file_created		
file_deleted		
file_exists		
file_failed		
file_moved		
file_opened		
file_read		
file_recreated		
file_written		
guid		
mutex		
wmi_query		

Table 4: All extracted dynamic features

A Python script creates the specific file format by writing the dynamic feature names into the attribute section and the values into the data section. The values of the data section concerning the memory features are the actual percentage of CPU and RAM usage during malware execution. Concerning disk and network features, two possible approaches were examined. The first approach was using a binary technique for the data section. This would specify a "1" or a "0" for the malware either possessing that feature or not. The other approach was an amount-based method in which the quantity of values of each feature was appended to the data section. Therefore, an example of a data section entry could either look like this:

```
"values" : ["1","1", "1", "1", "1", "0", "1", "1", "1", "1", "1", "0",
            "1", "1", "1", "0", "1", "1", "1", "1", "1", "0", "1", "1", "0",
            "0", "0", "0", "1", "1", "1", "1", "agent"]
```

or the equivalent line for the amount-based approach like this:

```
"values" : ["47","1", "527", "92", "48", "0", "72", "48", "76", "4",
            "0", "10", "33", "9", "0", "1606", "42", "1", "1", "6", "0",
            "7", "1", "0", "0", "0", "0", "17", "20", "64", "19", "agent"]
```

Each value between quotation marks represents one of the 31 features plus the malware family

name listed in the attribute section. The Pseudocode is presented in Algorithm 4 and the full Python code is listed in Appendix A.4.

---

**Algorithm 4** Pseudocode to extract all dynamic features and create Weka file

---

```

for  $f \leftarrow files$  do
  | for  $key = "summary"$  do
  | | if  $key \notin features$  then
  | | |  $features.append(key)$ 
  | | end
  | end
end

create(files) for  $f \leftarrow features$  do
  |  $file.append(f)$ 
end

for  $f \leftarrow files$  do
  | for  $key = "summary"$  do
  | | for  $f \leftarrow features$  do
  | | | if  $key = f$  then
  | | | | if  $key = memoryFeature$  then
  | | | | |  $file.append(key.value)$ 
  | | | | end
  | | | | if  $key = diskFeature$  or  $key = networkFeature$  then
  | | | | |  $file.append(len(key))$ 
  | | | | end
  | | | end
  | | end
  | end
end

```

---

For the second experiment the static features have already been extracted and stored as dataset. This dataset contained 328,350 entries and was provided by Andrii Shalaginov, the co-supervisor of this thesis. The dataset was already stored in a specific CSV-format which Weka can read. Therefore, the only thing left to do was extracting the same 7,009 samples used in the dynamic analysis, identified by their MD5 hash value. The created Python code to do this task can be found in Appendix

A.5. The dataset still holds the features from VirusTotal. However, they can easily be removed in Weka itself, since Weka lists all included attributes which can be excluded from the classification at the push of a button.

#### 5.4.5 Application of Machine Learning

The created file can then be fed to Weka, where the features are ranked, selected and classified in their respective families. To find the best classifier and combination of features, Weka has some built-in functions for attribute selection and functionalities for feature ranking and correlation. The various metrics for feature selection methods are presented in Section 4.4. After importing the file containing all features, filters can be applied. To include the appropriate feature selection method the supervised filter 'AttributeSelection' is added to the dataset. Then, the specific attribute evaluator and the search method are added. The attribute evaluator defines the metric for feature selection. It is the technique to evaluate each attribute in context of the output variable. The search method is the technique on how to rank or navigate different combinations of attributes [75]. Table 5 lists the used feature selection metrics and connects them to the corresponding module name and the necessary search method in Weka. The commonly used classifiers presented in Section 3.6 are used for the classification alongside a 10 fold cross-validation approach as test option. Cross-validation is deployed because the data, used for training and testing a machine learning algorithm, should never be derived from the same sample set. The detailed functionality of cross-validation is explained in Section 4.5.

Feature Selection Metric	Attribute Evaluator in Weka	Search Method in Weka
Correlation	CorrelationAttributeEval	Ranker
Information Gain	InfoGainAttributeEval	Ranker
Learner	WrapperSubsetEval	GreedyStepwise or BestFirst

Table 5: Used feature selection methods with connection to Weka module names for reproducibility



## 6 Results & Analysis

In this chapter the results of both experiments, introduced in Chapter 5, are presented and critically analysed.

Table 6 and Table 7 show the most commonly used classifiers for malware classification with machine learning, derived from Section 3.6, their true positive (TP) and false positive (FP) rate, the F-measure

$$2 \times \frac{\textit{precision} \times \textit{recall}}{\textit{precision} + \textit{recall}} \quad (6.1)$$

which is the harmonic mean between precision ( $\frac{TP}{TP+FP}$ ) and recall ( $\frac{TP}{TP+FN}$ ), and the area under the ROC curve. Table 6 displays the result of the binary classification while Table 7 lists the outcome of the amount-based classification. All values in both tables represent the calculated weighted average to combine the individual results per class with the amount of samples used by each class. It is calculated as followed:

$$\textit{WeightedAverage} = \sum_{i=1}^{10} \frac{v_i \times n_i}{N} \quad (6.2)$$

The weighted average is calculated as the sum of the value per class ( $v_i$ ) times the amount of samples per class ( $n_i$ ) for all ten classes divided by the total amount of samples ( $N$ ). This means that the weighted average takes the amount of samples per class into account. Therefore, a class with more malware samples influences the weighted average more than a class with less samples. Consequently, the overall accuracy is not being listed because in case of multi-class classification it is not reliable since it's not considering unbalanced sample distribution. Thus, the TP rate, calculated as weighted average, is considered as performance indication of the model. Moreover, 10-fold cross-validation was used as test option. For the k-nearest neighbours classifier only the k producing the best result is listed. To get a broad view of the results, initially no feature selection methods were applied.

It can be easily seen that the Random Forest classifier achieves the best results in both the binary and the amount-based classification approach. In case of multi-class classification it is important to highlight the TP rate per class, which is:

```
agent: 0.56; hupigon: 0.6; obfuscator: 0.61; onlinegames: 0.93;
renos: 0.95; small: 0.83; vb: 0.76; vbinject: 0.81; vundo: 0.87;
zlob: 0.95
```

for the binary classification approach and:

```
agent: 0.64; hupigon: 0.69; obfuscator: 0.7; onlinegames: 0.95;
renos: 0.97; small: 0.87; vb: 0.84; vbinject: 0.9; vundo: 0.94;
zlob: 0.98
```

Classifier	TP Rate	FP Rate	F-Measure	ROC Area
Naive Bayes	0.575	0.057	0.555	0.884
SVM	0.77	0.027	0.771	0.927
ANN	0.779	0.025	0.78	0.923
5-nearest Neighbours	0.8	0.023	0.8	0.955
Logistic Regression	0.74	0.033	0.732	0.943
J48	0.804	0.023	0.804	0.934
Random Forest	0.818	0.022	0.817	0.970

Table 6: Weighted average of different classifiers with binary approach classification of dynamic features

Classifier	TP Rate	FP Rate	F-Measure	ROC Area
Naive Bayes	0.338	0.079	0.303	0.767
SVM	0.525	0.073	0.494	0.787
ANN	0.641	0.047	0.635	0.871
1-nearest Neighbours	0.677	0.04	0.676	0.817
Logistic Regression	0.691	0.042	0.677	0.921
J48	0.842	0.018	0.841	0.932
Random Forest	0.873	0.015	0.872	0.984

Table 7: Weighted average of different classifiers with amount-based approach classification of dynamic features

for the amount-based approach. Since Random Forest seems to achieve the best results for the constructed experiment in this work, only the **RF** classifier is being considered from this point on. Moreover, it can be seen that the amount-based classification approach obtains better results in combination with Random Forest than the binary approach with the same classifier. Therefore, the commonly used feature selection methods, presented in Section 4.4, are applied to the amount-based classification approach. Table 8 presents the results for Correlation-based, Information Gain-based and Learner-based feature selection method and their overall classification performance in the last row in form of the true positive rate calculated as weighted average. In the column for Correlation-based feature selection the features are ranked by descending order and the values represent the correlation between each feature and the output variable. In the Information Gain column the values indicate the information, also called entropy, the features contribute for the output variable. The feature with the highest information contribution at top descending to the lowest. The third column presents the subset of attributes producing the best results. [75] The goal with finding the best subset is to simplify and to speed up the classification process. Fig. 8 visualises those results. Green indicates the first ten ranked features, orange is used to display the second ten features and red lists all features ranked as last eleven. The two features `dll_loaded` and `regkey_written` have a green label for all three metrics of feature selection. The features `file_created` and `file_written` are highly ranked in Correlation and Information Gain feature selection but are



<b>Correlation</b>	<b>Information Gain</b>	<b>Best Subset (Learner)</b>
resolves_host (0.17)	regkey_read (1.342)	regkey_written
downloads_file (0.121)	regkey_opened (1.205)	dll_loaded
dll_loaded (0.116)	file_exists (1.076)	regkey_opened
guid (0.108)	dll_loaded (0.984)	file_exists
peakCPU (0.107)	regkey_written (0.897)	file_failed
file_created (0.09)	directory_enumerated (0.846)	regkey_read
directory_created (0.088)	file_opened (0.702)	directory_enumerated
command_line (0.087)	file_created (0.674)	resolves_host
file_written (0.085)	file_read (0.626)	downloads_file
regkey_written (0.077)	file_written (0.521)	averageCPU
averageCPU (0.077)	file_failed (0.488)	
regkey_opened (0.077)	file_deleted (0.466)	
fetches_url (0.071)	mutex (0.455)	
mutex (0.071)	regkey_deleted (0.453)	
file_deleted (0.069)	resolves_host (0.446)	
connects_host (0.066)	averageCPU (0.418)	
file_recreated (0.065)	file_recreated (0.405)	
file_failed (0.061)	command_line (0.385)	
file_moved (0.06)	peakRAM (0.341)	
file_opened (0.055)	averageRAM (0.316)	
file_exists (0.053)	guid (0.288)	
regkey_read (0.047)	downloads_file (0.227)	
averageRAM (0.042)	directory_created (0.205)	
peakRAM (0.041)	peakCPU (0.185)	
directory_removed (0.039)	connects_ip (0.167)	
file_read (0.039)	file_copied (0.128)	
directory_enumerated (0.037)	fetches_url (0.104)	
regkey_deleted (0.033)	connects_host (0.067)	
wmi_query (0.025)	directory_removed (0.048)	
file_copied (0.017)	wmi_query (0.04)	
connects_ip (0.009)	file_moved (0.032)	
<b>0.875</b>	<b>0.874</b>	<b>0.858</b>

Table 8: Feature selection methods of dynamic features



=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.650	0.031	0.663	0.650	0.657	0.625	0.943	0.753	agent
	0.708	0.010	0.739	0.708	0.723	0.713	0.971	0.790	hupigon
	0.698	0.015	0.748	0.698	0.722	0.706	0.967	0.814	obfuscator
	0.955	0.004	0.976	0.955	0.965	0.960	0.992	0.982	onlinegames
	0.968	0.003	0.978	0.968	0.973	0.969	0.999	0.995	renos
	0.867	0.015	0.891	0.867	0.879	0.862	0.988	0.935	small
	0.837	0.029	0.816	0.837	0.826	0.799	0.980	0.918	vb
	0.910	0.028	0.832	0.910	0.869	0.850	0.989	0.938	vbinject
	0.935	0.003	0.935	0.935	0.935	0.932	0.995	0.969	vundo
	0.977	0.002	0.986	0.977	0.981	0.979	0.998	0.994	zlob
Weighted Avg.	0.875	0.015	0.876	0.875	0.875	0.861	0.985	0.928	

Figure 9: Detailed performance evaluation of each malware family for Random Forest classifier and Correlation-based feature selection of dynamic features

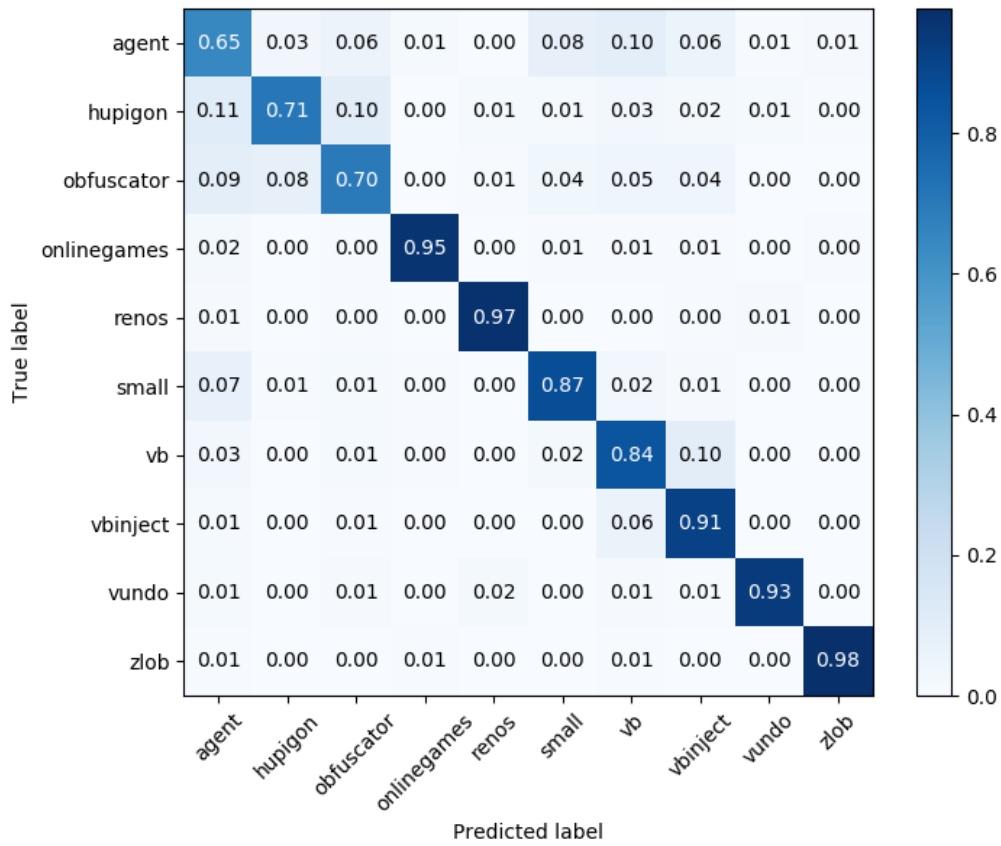


Figure 10: Confusion matrix for Random Forest classifier and Correlation-based feature selection of dynamic features

seen that the only noteworthy falsely identified families are vb and vbinject which get mistakenly predicted as the other class with a rate of around 0.1.

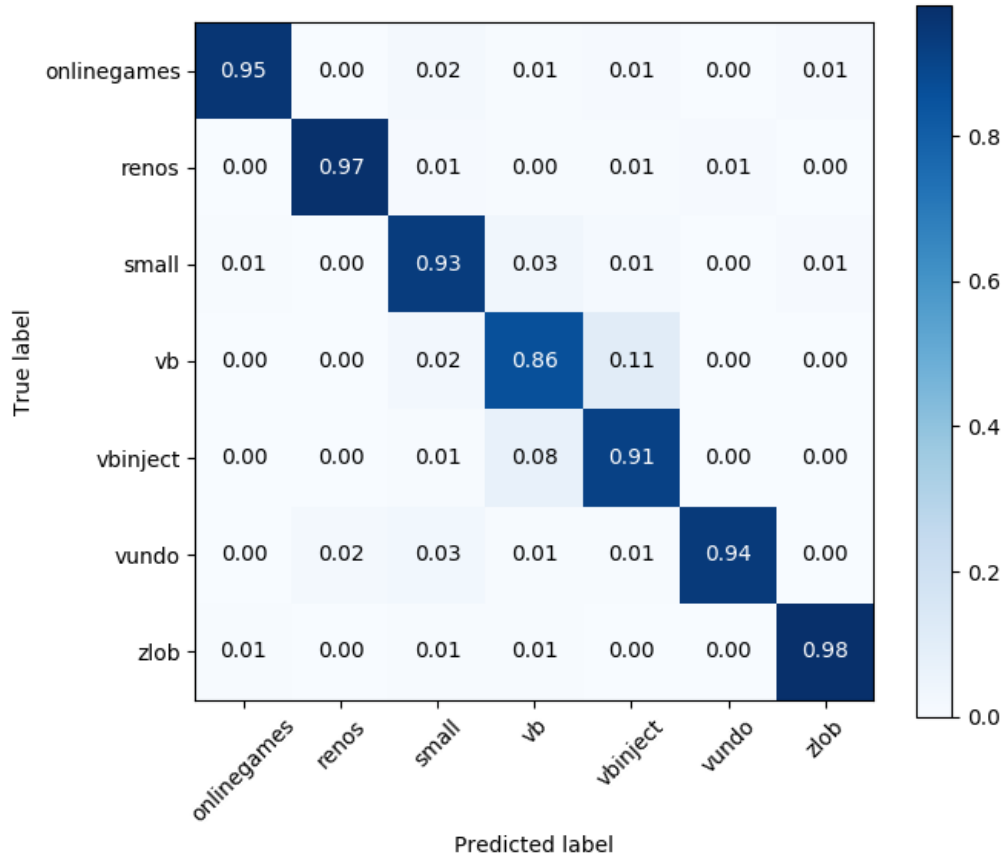


Figure 11: Confusion matrix for Random Forest classifier of the seven best performing families of dynamic features

In addition, the second experiment with the static features mentioned in Subsection 5.4.4 achieves an overall performance of 86.7% without any feature selection methods applied. The TP rates per class are:

agent: 0.63; hupigon: 0.63; obfuscator: 0.84; onlinegames: 0.94;  
 renos: 0.97; small: 0.84; vb: 0.79; vbinject: 0.9; vundo: 0.93;  
 zlob: 0.98

To analyse the static features even further, the best eight features, provided by Correlation and Information Gain feature selection, and the best subset of features are presented in Table 9. The values denote the same as in Table 8 listed above.

To get a better and simpler comparison of the performance of static and dynamic features, the

Correlation	Information Gain	Learner
pe_detected (0.21)	filesize (1.75)	pe_api
pe_packer (0.20)	size_TOT (1.47)	pe_library
filesize (0.14)	size_TEXT (1.41)	pe_detected
pe_api (0.13)	size_DATA (1.3)	size_DATA
pe_library (0.12)	pe_api (0.92)	size_TOT
size_TOT (0.10)	pe_library (0.7)	filesize
size_DATA (0.09)	pe_packer (0.61)	
size_TEXT (0.07)	pe_detected (0.34)	
<b>0.870</b>	<b>0.870</b>	<b>0.866</b>

Table 9: Feature selection methods of static features

true positive rates are visualised in Fig. 12. It compares the dynamic and the static features by class against each other based on their true positive rates from the Correlation-based feature selection method. It can be seen that the dynamic features are either on the same level or better than the static features for all families except for the obfuscator family which performs far better with static features.

Table 9 shows that from the best performing static features only the feature `pe_dll` is used in the dynamic approach. This means that the static features could add another level of abstraction to the classification results of the dynamic features. Therefore, merging the dynamic with the static features results in an even higher true positive rate of 0.923. Looking at the best eleven performing features based on the three metrics for feature selection, it can be seen that the features are ranked by a combination of their individual ranking from Table 8 and Table 9 which are:

```
resolves_host, pe_detected, pe_packer, downloads_file, pe_api,
dll_loaded, guid, peakCPU, pe_library, filesize, file_created
```

for Correlation,

```
regkey_read, filesize, regkey_opened, file_exists, size_TOT,
dll_loaded, size_TEXT, regkey_written, directory_enumerated,
size_DATA, file_opened
```

for Information Gain and

```
pe_api, pe_detected, size_DATA, size_TOT, filesize, regkey_written,
regkey_opened, file_exists, regkey_read, directory_enumerated,
resolves_host
```

as best subset of features.

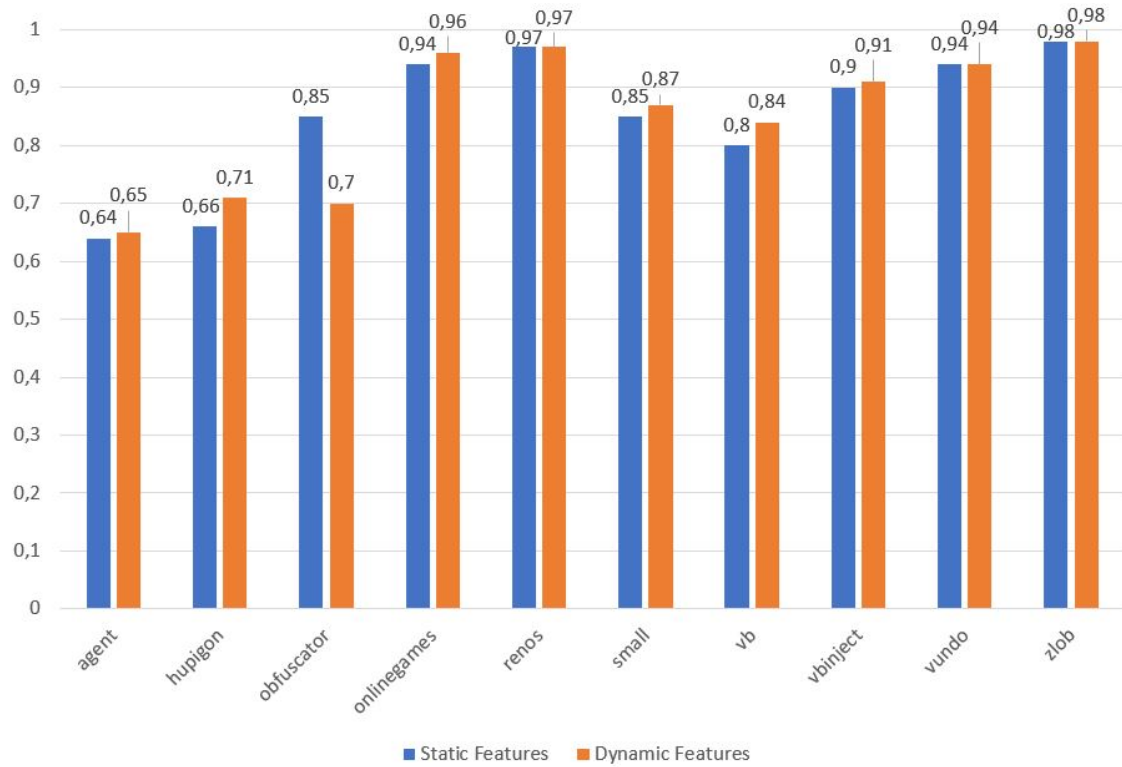


Figure 12: Comparison of TP rates from Correlation-based feature selection of static against dynamic features by class

## 7 Discussion

In the following chapter the results, presented in Chapter 6, are interpreted, explained and discussed in detail.

First, two major findings have to be explained; Random Forest as best performing classifier with amount-based classification achieving the best results. As stated by Tian et al., *RF has excellent accuracy among current classifier algorithms* [46], who also achieve their best results with Random Forest. The RF classifier is a combination of decision trees. For a classification every tree is allowed to make a decision and the class with the most votes determines the final classification. Therefore, the evaluation process can be parallelised, which makes Random Forest an efficient classifier also for big datasets. Random Forest is also very robust in respect to noise. RF can also better handle a high-dimensional feature space compared to other classifiers, which applies to the conducted experiment of this thesis with 31 different features. Even removing the worst performing features from the experiment, based on Information Gain and Correlation, will decrease the overall classification result of Random Forest. Some classifiers, such as Logistic Regression or Naive Bayes for example, are rather used for binary classification while others, such as Artificial Neural Networks, increase in time complexity up to an unmanageable processing time by adding hidden layers to increase the mapped complexity. Furthermore, the two methods of mapping the data to the attributes, binary and amount-based, achieve different results. For all classifiers, except J48 and Random Forest, the binary approach performs better than the amount-based. However, the two decision tree classifiers, which perform the best for both classification methods, achieve better results for the amount-based approach. Using the exact amount of occurrences of each feature instead of a binary indication adds another level of abstraction to the dataset. Therefore, RF benefits from it since it can handle a high-dimensional feature space.

Concerning the features selection methods, it can be seen which feature performs the best and contributes the most towards the overall classification accuracy. Table 8 lists the individual features with the exact values as contribution to the total performance. Fig. 8 visualises those results in a comprehensible manner. On the one hand, some features, such as `directory_removed`, `file_copied`, `wmi_query`, `connects_ip`, `file_moved`, `connects_host`, `averageRAM` and `peakRAM`, perform worse than others. On the other hand, some features contribute more to the overall performance. Those features are `dll_loaded`, `regkey_written`, `regkey_opened`, `file_created`, `file_written` and `resolves_host`. To interpret these results more information about the individual distribution of the data points of each feature is needed. Weka supports such a possibility by visualising the distribution of data points for each feature. Fig. 13 shows the data distribution of the worst performing features while Fig. 14 visualises the data distribution of the best performing features. Table 10 summarises the values of the highest point of the curve for each feature. If those values are

high, this means that the particular feature has many malware samples corresponding to the same data points. Therefore, the more similar data points a feature has, the less distinctive the feature is. Based on the table it can be seen that the features contributing less to the overall classification performance are also less distinctive than the features with a high contribution. For example, all disk and network features from the worst performing group of features have a range of zero or close to it. This means that most of the malware samples do not remove directories, move files, connect to hosts or query WMI and they copy files and connect to IPs only a little. Concerning memory features, most of the malware samples use around 20 to 21 % of the RAM in average and as peak. Regarding the best performing features, around half the malware samples write to registry keys, create files, write to files and resolve hosts little to none. In conclusion, features with a low Correlation and Information Gain, as well as not being part of the best subset of features, have high values in the same range making them less distinctive. Furthermore, the two best performing features `dll_loaded` and `regkey_written`, which can be seen in Fig. 8, are two very distinctive features heavily affecting the functionality of the malware. DLLs are loaded by the malware to use different functions and registry keys are written in order to manipulate system values. Therefore, those two features contribute the most to the overall malware classification because each malware family has a different functionality which influences the use of DLLs and the writing of registry keys. Moreover, from the group of the worst features there are some, such as `directory_removed` and `file_copied`, which are not often supported by malware, especially the ten malware families used in this experiment. If the intention of the malware is not explicitly removing directories or copying files, then those two functions are usually not implemented since they are not common functionalities for malware. Besides, RAM is also not adding much to the overall performance. This could be a result of the 8GB of RAM available for the guest machine. Since the average RAM and peak RAM usage was calculated by the percentage of occupied RAM during the execution of the malware, it wasn't influenced a lot by the rather small malware invoking only a couple of processes. If the total RAM would have been 1-2GB of RAM, the malware execution would have influenced the total percentage much more. Moreover, an attempt to rank the three groups of features is done. Analysing Fig. 8 and adding values to the different ranks; let green have the value of 1, orange of 2 and red counts as 3 while no colour (white) is calculated as 2.5. This leads to an average value of 5.9 for disk, 6.5 for network and 6.6 for memory features. Therefore, disk features contribute the most to the overall classification in this experiment. This is because disk features influence the malware functionality the most and they are easy to extract, evaluate and classify. Network traffic is not used by every malware and therefore not represented by the common malware samples. Memory features are the hardest to extract due to their volatile nature but they carry a lot of potential for the future of multinomial malware classification.

Furthermore, looking at Fig. 10 it can be seen that not all families are equally predictive. Especially the three classes `agent`, `hupigon` and `obfuscator` are wrongly predicted for one another quite often. While `agent` is predicted as `hupigon` and `obfuscator` in an average of 4.5% of the times, `hupigon` is predicted as `agent` and `obfuscator` in 10 to 11% of the cases and `obfuscator` is predicted as `agent` and `hupigon` with a rate of 8 to 9%. Fig. 11 shows the result of removing those three

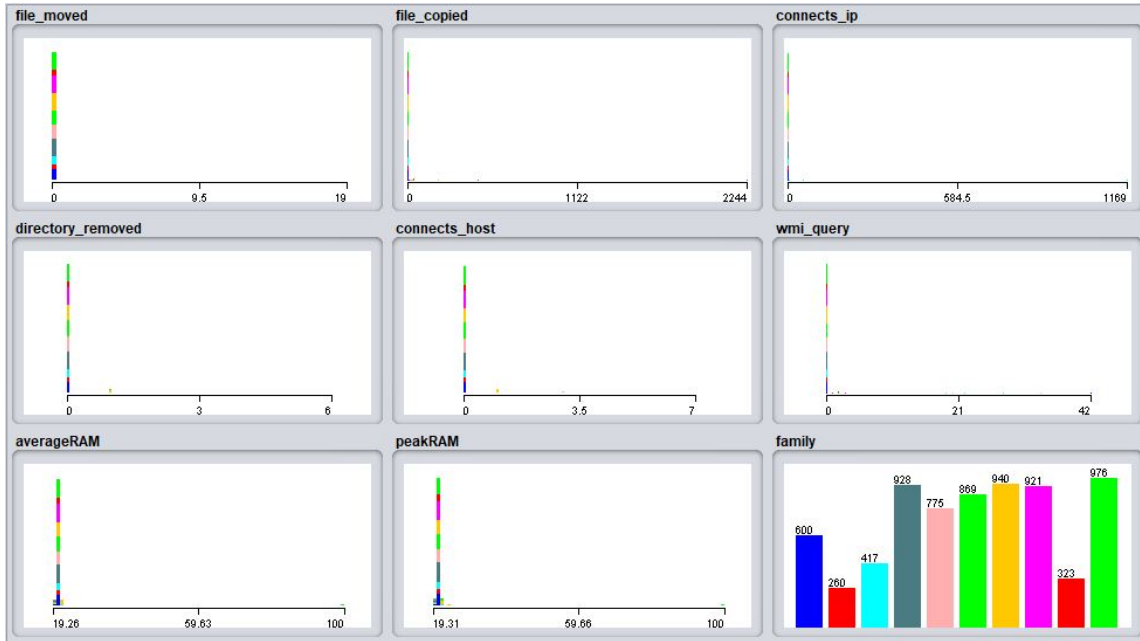


Figure 13: Data distribution of the worst performing dynamic features

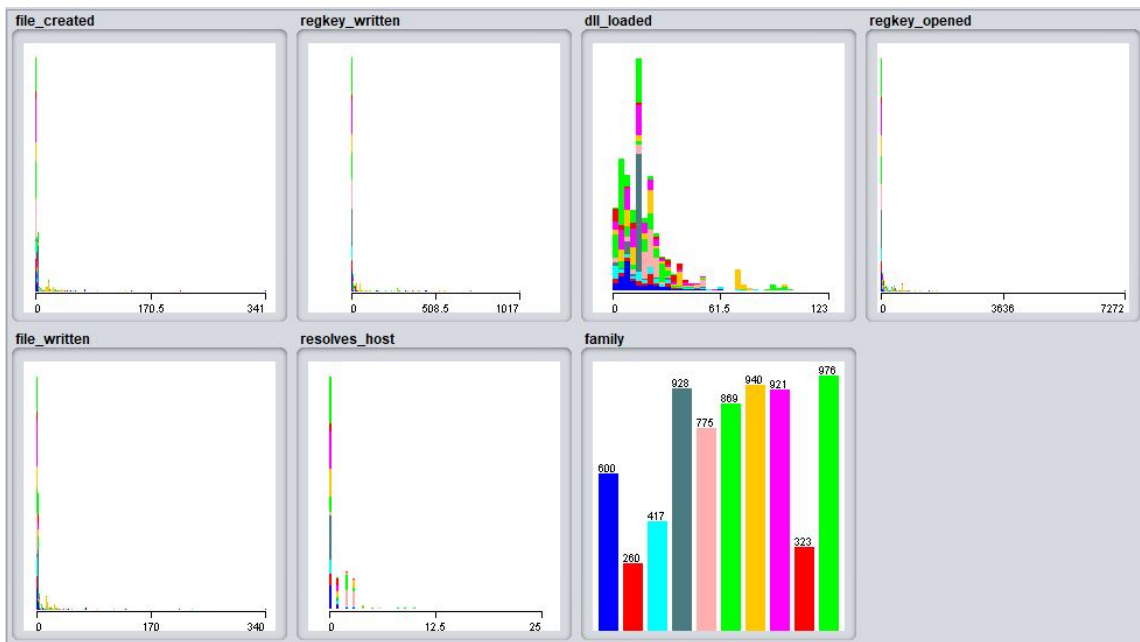


Figure 14: Data distribution of the best performing dynamic features

	Feature	Value	Range
<b>Worst Features</b>	directory_removed	6757	0
	file_copied	6881	0-7
	wmi_query	6848	0
	connects_ip	7006	0-4
	file_moved	6873	0
	connects_host	6546	0
	averageRAM	6009	20-21
	peakRAM	5902	20-21
<b>Best Features</b>	dll_loaded	1490	13-16
	regkey_written	5764	0-7
	regkey_opened	5465	0-34
	file_created	4115	0-1
	file_written	3953	0-1
	resolves_host	4778	0

Table 10: Data points of the peak of the curve based on Fig. 13 and Fig. 14

families from the classification process, which leads to an overall classification accuracy of 94%. This proves that the performance of malware classification is highly dependent on the different malware families used in the process. Not all families are easy to detect. However, comparing the individual classification rates of each malware family from Fig. 10 with the detailed description listed in Section 4.1, it can be seen that agent and hupigon are rather powerful malware families. In terms of functionality they offer a lot of variety like installing Adware, downloading additional malware, changing Windows configurations, opening backdoors, starting services and communicating with the attacker. The same with the obfuscator family, it is a broader malware family in terms of functionality. It rather can be seen as collective name for malware with obfuscation techniques but therefore, can have any kind of functionality. In contrary, malware families, such as onlinergames, renos, small, vundo and zlob, are more specific in their functions. Malware from the onlinergames family focus on online game key strokes, renos mostly shows fake security warnings, small downloads PUPs, vundo deals with pop-up advertisement and zlob focuses on internet explorer. Those malware families are rather unique and distinctive in their actions. Moreover, the two families vb and vbinject are also mistakenly predicted as one another. This is due to the fact that they show strong similarities in their programming since both are based on Visual Basic. Besides, it can be seen that the malware samples from the agent family are 7 to 8% of the times wrongly predicted as member of the small family and vice versa. This could be related to the fact that both have the functionality of downloading additional malware. In addition, malware from the agent family are also mistakenly predicted to be a sample from the vb family in 10% of the cases. This occurs because malware of the agent family is not quite distinctive with their functionalities and vb is more of a generic, Visual Basic-based term for malware unclear about the exact functions.

In addition, the second experiment dealt with static feature classification in order to provide a



comparison functionality of static against dynamic features. The achieved performance of the classification with the static features is, even though it is lower than the performance of the dynamic features, quite impressive. It was unexpected for the static features to achieve such a high accuracy. Especially the performance of the obfuscator malware family was surprising, since obfuscated malware samples were suspected to perform bad with static features, but the contrary happened. This is because the obfuscator family has a set of distinct static characteristics describing specific properties of this class. Fig. 15 shows the data distribution of the best performing static features, based in Table 9, with distinct characteristics of malware samples from obfuscator. Red depicting the obfuscator family and blue the combination of all other nine families. It can be seen that the obfuscator family, for example, has a bigger file size and file size regions than the other families in average. Moreover, the amount of packers in obfuscated malware tends to be larger. However, the best performing static features were `pe_detected`, `pe_packer`, `pe_api`, `pe_library`, `filesize`, `size_TOT`, `size_DATA` and `size_TEXT`. From those features only `pe_library`, which can be resembled by `dll_loaded`, is also used in the dynamic feature classification. This means that the classification accuracy of the dynamic features can be further improved by adding more features, as the results of merging the dynamic with the static features indicates. The combination of both groups improves the classification accuracy from 87.5% (dynamic features) to 92.3% (combination of static and dynamic). Therefore, dynamic features can be further improved by increasing the feature space to a higher dimension. Especially the features concerning the file size were contributing much to the overall performance. A process of dynamically extracting the actual file size and the individual file size regions of the malware samples could increase the classification accuracy by decreasing the influence of obfuscation simultaneously.

Now the research questions stated in Chapter 1 can be answered. Even though static features perform an overall decent classification result, they could be easily manipulated by obfuscation techniques. As seen in Table 9, which shows the best performing static features, file sizes were quite important for the classification of static features. Unfortunately, file sizes can easily be changed without changing the overall functionality by adding unnecessary code fragments, for example, or slightly transforming parts of the code. Moreover, statically extracted strings could be obfuscated by the use of encryption techniques which would make nearly all PEframe features invalid. Therefore, it is important to focus more on dynamic features, useful for multinomial malware classification, such as `dll_loaded`, `regkey_written`, `regkey_opened`, `file_created`, `file_written` and `resolves_host`. Even though the performance gain is, with an increase of 0.5%, not exceptionally high, the result is less prone to obfuscation. Moreover, the performance of dynamic features can be increased by adding API calls, for example. Since technical difficulties in the beginning of the experiment prevented the extraction of API calls, it was decided to not include them in the experiment. However, adding the statically extracted `pe_api` feature to get a glimpse of the possible performance, results in a classification accuracy of 89.7%, which would be a performance increase of 2.7% compared to the static features.

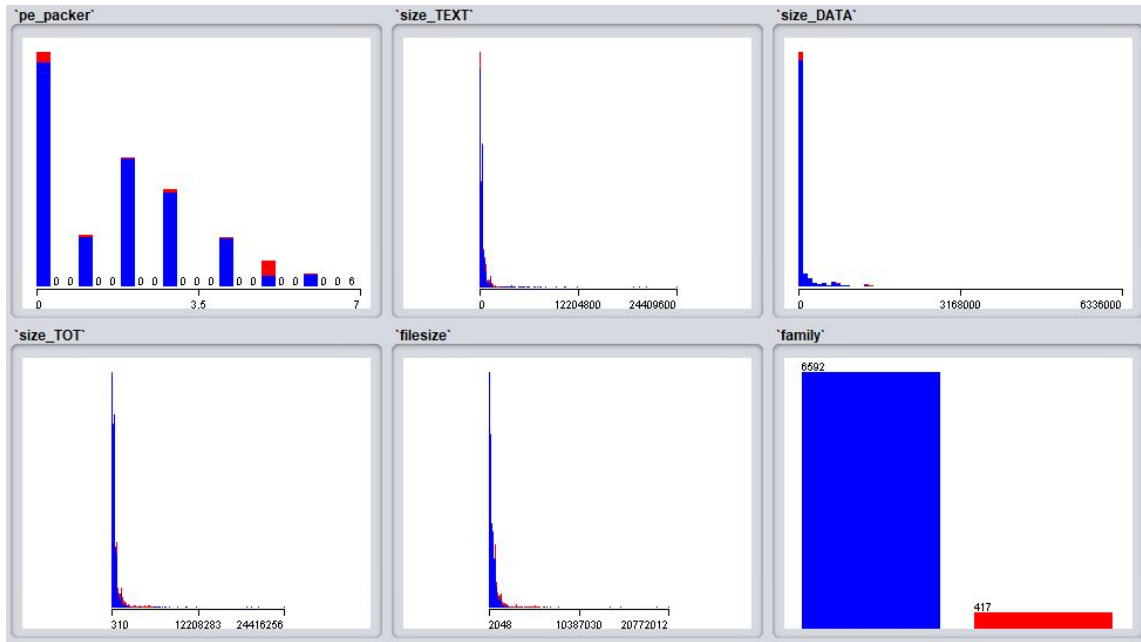


Figure 15: Data distribution of static features with obfuscator as red

## 7.1 Implications

The following two subsections describe the long-term impact of this study and how the achieved results can be applied in future works.

### 7.1.1 Theoretical Implications

Based on RQ1 and RQ3, the limitations of the static features and the performance gain of the dynamic features have been analysed. The achieved results show that the future of malware detection/classification will be based on dynamic malware analysis. Therefore, further research in this area needs to be done.

### 7.1.2 Practical Implications

Regarding RQ2, the best dynamic features for multinomial malware classification needed to be found. Table 10 lists the best and the worst performing features of this study. Based on this knowledge, it can be avoided to use the worst performing features in future works. Instead, the focus can be put on the best performing features with the attempt to improve the use of those features in a different structure as elaborated further in Chapter 8.

## 7.2 Limitations of the Study

Despite the valuable results in general, there are some limitations to the experiment. First, it has to be mentioned that the conducted experiment works only for malware samples which are unable

to detect their own execution in a sandbox environment. In the beginning of the experiment all malware samples with anti-VM and anti-debug features have been removed beforehand. Malware with those kind of abilities are unable to be classified by this work. Second, only Windows PE32 malware samples have been analysed in the experiment. Malware written for different operating systems, such as macOS or Android, can not be examined with this approach. Moreover, in the experiment, it was taken care of embedded Word documents and PDF files, since those are the most common embedded file formats in malware. However, other file formats embedded in a malware sample are not recognised and therefore, were not executed during dynamic analysis. Furthermore, since the performance of the analysis was not considered to be a requirement of the work, it is rather not suitable for real-time analysis. Last, the concept of reliability has to be considered. The number of malware samples is not equally distributed among the ten families, which makes the experiment not completely reliable.



## 8 Conclusion

The following chapter summarises and concludes the content of the experiment conducted throughout this thesis.

9,823 Windows PE32 malware samples were originally received for malware classification. After a pre-processing phase 8,305 samples remained. With the automated malware analysis system Cuckoo Sandbox the remaining samples were executed inside a controlled environment, a virtual machine. For each malware sample the dynamic features were extracted and stored in a report. After the error removal phase 7,009 reports of malware samples remained for classification. The extracted features were put together in a single file, readable for the machine learning library Weka. Therefore, the name of the features were stored in the attribute section while the number of individual occurrences was saved as data points. Selected metrics of feature selection methods were used to rank the features and to analyse the individual contribution of each feature. Moreover, static features of the same 7,009 malware samples were used to compare their results to the performance of the dynamic features. While the static features achieved an overall classification accuracy of 87% TP, an F-measure value of 0.869 and an AUC of 0.985, the dynamic features accomplished a performance of 87.5% (RQ3) in terms of true positive rate, a F-measure value of 0.875 and an AUC of 0.985. Even though this performance gain seems small, it has to be considered that dynamic features, unlike static features, are not susceptible to obfuscation techniques (RQ1). Hence, the best performing dynamic features for multinomial malware classification are the modified and opened registry keys, the created and modified files, the loaded DLLs and the resolved hosts (RQ2).

As mentioned in Section 7.2, malware samples with anti-VM and anti-debug features are not included in the experiment. This is a common approach for dynamic malware classification also in other literature. However, this is not an appropriated method in the long run for malware analysts to exclude malware samples with such features. Therefore, more research towards automated malware analysis systems with the ability to run such malware samples has to be done in the future. For example, the software VMCloak, programmed and maintained by one of the developer of Cuckoo Sandbox, aims to target this issue. Amongst other things it tries to make the set up virtual machine more difficult to detect with conventional ways. Moreover, the used dynamic features for multinomial malware classification achieved a good performance, however, they could have been used in a different structure, as suggested by multiple recent literature. For example, the exact value of the called APIs, the modified registry keys or the queried WMIs could have been taken instead of the total amount as single value. Another approach could involve certain sequences of API calls or loaded DLLs instead of the overall number of occurrences or a different weighting of features, since some API calls, for example, are more relevant than others, as stated by Kolosnjaji et al. [80]. Furthermore, the used memory features can be improved in precision and extraction. Instead of just

considering the first ten seconds of the running malware process, a different or longer time frame could have been analysed as well as another extraction method instead of a powershell module. In addition, other memory features have to be analysed on their performance towards multinomial malware classification. Such features could be, for example, the number of spawned processes by the malware. Cuckoo's monitor could be used for this since it follows the malware through all processes either generated or taken over by it. Another approach could be to extract and analyse full machine or process memory dumps. However, this would heavily increase the time performance of the analysis.

## Bibliography

- [1] AV-TEST. 2018. Malware. <https://www.av-test.org/en/statistics/malware/>. Accessed: 2018-11-12.
- [2] Hansen, S. S. & Larsen, T. M. T. Dynamic malware analysis: Detection and family classification using machine learning. Master's thesis, Aalborg University, 2015.
- [3] VirusTotal. Virustotal. <https://www.virustotal.com/>. Accessed: 2018-11-12.
- [4] Pektaş, A. *Classification des logiciels malveillants basée sur le comportement à l'aide de l'apprentissage automatique en ligne*. PhD thesis, Université Grenoble Alpes, 12 2015.
- [5] Levin, B. & Simpson, D. 2018. Malware names. <https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming#variant-letter>. Accessed: 2019-01-03.
- [6] Banin, S., Shalaginov, A., & Franke, K. 2016. Memory access patterns for malware detection, norsk informasjonssikkerhetskonferanse (nisk). *Norsk Informasjonssikkerhetskonferanse*, 96–107.
- [7] Guarnieri, C., Tanasi, A., Bremer, J., & Schloesser, M. 2010. *Cuckoo Sandbox Book*. Read the Docs, 2.0.6 edition.
- [8] Ravula, R. R. Classification of malware using reverse engineering and data mining techniques. Master's thesis, University of Akron, 2011.
- [9] Kaspersky Lab. 2017. Kaspersky lab detects 360,000 new malicious files daily – up 11.5% from 2016. [https://www.kaspersky.com/about/press-releases/2017\\_kaspersky-lab-detects-360000-new-malicious-files-daily](https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily). Accessed: 2018-11-12.
- [10] Liu, J., Wang, Y., Xie, P., Wang, Y., & Huang, Z. 2015. Malware similarity analysis based on graph similarity flooding algorithm. In *Advances in Computer Science and Ubiquitous Computing*, Park, D.-S., Chao, H.-C., Jeong, Y.-S., & Park, J. J. H., eds, 31–37, Singapore. Springer Singapore. doi:10.1007/978-981-10-0281-6\_5.
- [11] Alkhateeb, E. M. S. August 2017. Dynamic malware detection using api similarity. In *2017 IEEE International Conference on Computer and Information Technology (CIT)*, 297–301. doi:10.1109/CIT.2017.14.

- [12] Payer, M., Crane, S., Larsen, P., Brunthaler, S., Wartell, R., & Franz, M. 2014. Similarity-based matching meets malware diversity. *CoRR*, abs/1409.7760.
- [13] Park, J., Kim, M., Noh, B., & Joshi, J. B. D. September 2006. A similarity based technique for detecting malicious executable files for computer forensics. In *2006 IEEE International Conference on Information Reuse Integration*, 188–193. doi:10.1109/IRI.2006.252411.
- [14] Yi, Y., Lingyun, Y., Rui, W., Purui, S., & Dengguo, F. 2011. Depsim: A dependency-based malware similarity comparison system. In *Information Security and Cryptology*, Lai, X., Yung, M., & Lin, D., eds, 503–522, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-642-21518-6\_35.
- [15] Han, K.-S., Kim, I.-K., & Im, E. G. 2012. Malware classification methods using api sequence characteristics. In *Proceedings of the International Conference on IT Convergence and Security 2011*, Kim, K. J. & Ahn, S. J., eds, 613–626, Dordrecht. Springer Netherlands. doi:10.1007/978-94-007-2911-7\_60.
- [16] Kim, J., Lee, S., Youn, J. M., & Choi, H. 2017. A study of simple classification of malware based on the dynamic api call counts. In *Advances in Computer Science and Ubiquitous Computing*, Park, J. J. J. H., Pan, Y., Yi, G., & Loia, V., eds, 944–949, Singapore. Springer Singapore.
- [17] Kim, H., Kim, J., Kim, Y., Kim, I., Kim, K. J., & Kim, H. September 2017. Improvement of malware detection and classification using api call sequence alignment and visualization. *Cluster Computing*. doi:10.1007/s10586-017-1110-2.
- [18] Narayanan, B. N., Djaneye-Boundjou, O., & Kebede, T. M. July 2016. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, 338–342. doi:10.1109/NAECON.2016.7856826.
- [19] Chia-mei, C. & Gu-hsin, L. August 2014. Research on classification of malware source code. *Journal of Shanghai Jiaotong University (Science)*, 19(4), 425–430. doi:10.1007/s12204-014-1519-1.
- [20] Grégio, A. R. A., de Geus, P. L., Kruegel, C., & Vigna, G. 2013. Tracking memory writes for malware classification and code reuse identification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Flegel, U., Markatos, E., & Robertson, W., eds, 134–143, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-642-37300-8\_8.
- [21] Liu, L., Wang, B.-s., Yu, B., & Zhong, Q.-x. September 2017. Automatic malware classification and new malware detection using machine learning. *Frontiers of Information Technology & Electronic Engineering*, 18(9), 1336–1347. doi:10.1631/FITEE.1601325.



- [22] Lee, T., Choi, B., Shin, Y., & Kwak, J. August 2018. Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient. *The Journal of Supercomputing*, 74(8), 3489–3503. doi:10.1007/s11227-015-1594-6.
- [23] Islam, R., Tian, R., Batten, L. M., & Versteeg, S. 2013. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2), 646 – 656. doi:https://doi.org/10.1016/j.jnca.2012.10.004.
- [24] Sathyanarayan, V. S., Kohli, P., & Bruhadeshwar, B. 2008. Signature generation and detection of malware families. In *Information Security and Privacy*, Mu, Y., Susilo, W., & Seberry, J., eds, 336–349, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70500-0\_25.
- [25] Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., & Quilbeuf, J. 2018. Tutorial: An overview of malware detection and evasion techniques. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, Margaria, T. & Steffen, B., eds, 565–586, Cham. Springer International Publishing. doi:10.1007/978-3-030-03418-4\_34.
- [26] Islam, R. & Altas, I. 2012. A comparative study of malware family classification. In *Information and Communications Security*, Chim, T. W. & Yuen, T. H., eds, 488–496, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-642-34129-8\_48.
- [27] Preda, M. D., Christodorescu, M., Jha, S., & Debray, S. January 2007. A semantics-based approach to malware detection. *SIGPLAN Not.*, 42(1), 377–388. doi:10.1145/1190215.1190270.
- [28] lastline. 2017. An introduction to advanced malware and how it avoids detection. [https://www.infosecurityeurope.com/\\_novadocuments/357216?v=636295318428200000](https://www.infosecurityeurope.com/_novadocuments/357216?v=636295318428200000). Accessed: 2018-11-23.
- [29] Sharma, A. October 2015. Evolution and detection of polymorphic and metamorphic malwares: A survey. *International Journal of Computer Applications*. doi:10.13140/RG.2.1.5104.1763.
- [30] Griffin, K., Schneider, S., Hu, X., & Chiueh, T.-c. 2009. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, Kirda, E., Jha, S., & Balzarotti, D., eds, 101–120, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04342-0\_6.
- [31] Mohamed, G. A. N. & Ithnin, N. B. 2018. Sbrt: Api signature behaviour based representation technique for improving metamorphic malware detection. In *Recent Trends in Information and Communication Technology*, Saeed, F., Gazem, N., Patnaik, S., Saed Balaid, A. S., & Mohammed, F., eds, 767–777, Cham. Springer International Publishing. doi:10.1007/978-3-319-59427-9\_79.

- [32] Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. 2011. Malware images: visualization and automatic classification. In *VizSEC*. doi:10.1145/2016904.2016908.
- [33] Islam, N., Das, S., & Chen, Y. April 2017. On-device mobile phone security exploits machine learning. *IEEE Pervasive Computing*, 16(2), 92–96. doi:10.1109/MPRV.2017.26.
- [34] Banin, S. & Dyrkolbotn, G. O. 2018. Multinomial malware classification via low-level features. *Digital Investigation*, 26, 107–117. doi:10.1016/j.diin.2018.04.019.
- [35] Bounouh, T., Brahim, Z., Al-Nemrat, A., & Benzaid, C. 2016. A scalable malware classification based on integrated static and dynamic features. In *Global Security, Safety and Sustainability - The Security Challenges of the Connected World*, Jahankhani, H., Carlile, A., Emm, D., Hosseinian-Far, A., Brown, G., Sexton, G., & Jamal, A., eds, 113–124, Cham. Springer International Publishing. doi:10.1007/978-3-319-51064-4\_10.
- [36] Fang, Y., Yu, B., Tang, Y., Liu, L., Lu, Z., Wang, Y., & Yang, Q. 2017. A new malware classification approach based on malware dynamic analysis. In *Information Security and Privacy*, Pieprzyk, J. & Suriadi, S., eds, 173–189, Cham. Springer International Publishing. doi:10.1007/978-3-319-59870-3\_10.
- [37] Tian, R., Islam, R., Batten, L., & Versteeg, S. Oct 2010. Differentiating malware from cleanware using behavioural analysis. In *2010 5th International Conference on Malicious and Unwanted Software*, 23–30. doi:10.1109/MALWARE.2010.5665796.
- [38] Shalaginov, A. & Franke, K. Dec 2016. Automated intelligent multinomial classification of malware species using dynamic behavioural analysis. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, 70–77.
- [39] Rieck, K., Trinius, P., Willems, C., & Holz, T. December 2011. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4), 639–668.
- [40] Nari, S. & Ghorbani, A. A. Jan 2013. Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, 642–647. doi:10.1109/ICCNC.2013.6504162.
- [41] Gandotra, E., Bansal, D., & Sofat, S. 01 2014. Malware analysis and classification: A survey. *Journal of Information Security*, 05, 56–64. doi:10.4236/jis.2014.52006.
- [42] Pircoveanu, R. S., Hansen, S. S., Larsen, T. M. T., Stevanovic, M., Pedersen, J. M., & Czech, A. June 2015. Analysis of malware behavior: Type classification using machine learning. In *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 1–7. doi:10.1109/CyberSA.2015.7166115.
- [43] Shalaginov, A., Grini, L. S., & Franke, K. July 2016. Understanding neuro-fuzzy on a class of multinomial malware detection problems. In *2016 International Joint Conference on Neural Networks (IJCNN)*, 684–691. doi:10.1109/IJCNN.2016.7727266.

- [44] Damodaran, A., Troia, F. D., Visaggio, C. A., Austin, T. H., & Stamp, M. 2015. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13, 1–12.
- [45] Shijo, P. & Salim, A. 2015. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46, 804 – 811. doi:10.1016/j.procs.2015.02.149.
- [46] Tian, R., Batten, L., Islam, R., & Versteeg, S. Oct 2009. An automated classification system based on the strings of trojan and virus families. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 23–30. doi:10.1109/MALWARE.2009.5403021.
- [47] Tian, R., Batten, L. M., & Versteeg, S. C. Oct 2008. Function length as a tool for malware classification. In *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, 69–76. doi:10.1109/MALWARE.2008.4690860.
- [48] Zhao, H., Xu, M., Zheng, N., Yao, J., & Ho, Q. Jan 2010. Malicious executables classification based on behavioral factor analysis. In *2010 International Conference on e-Education, e-Business, e-Management and e-Learning*, 502–506. doi:10.1109/IC4E.2010.78.
- [49] Ahmed, F., Hameed, H., Shafiq, M. Z., & Farooq, M. 2009. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2Nd ACM Workshop on Security and Artificial Intelligence*, 55–62, New York, NY, USA. ACM. doi:10.1145/1654988.1655003.
- [50] Uppal, D., Sinha, R., Mehra, V., & Jain, V. Sep. 2014. Malware detection and classification based on extraction of api sequences. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2337–2342. doi:10.1109/ICACCI.2014.6968547.
- [51] Distler, D. & Hornat, C. 2007. Malware analysis: An introduction. *SANS Institute, InfoSec Reading Room*.
- [52] You, I. & Yim, K. 11 2010. Malware obfuscation techniques: A brief survey. *Proceedings - 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010*, 297–300. doi:10.1109/BWCCA.2010.85.
- [53] Sinay, Y. 2017. Common malware evasion techniques. <http://blogs.microsoft.co.il/yuval14/2017/06/20/common-malware-evasion-techniques/>. Accessed: 2019-01-11.
- [54] Wojtczuk, R. & Rutkowska, J. 2011. Following the white rabbit: Software attacks against intel(r) vt-d technology. <https://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf>. Accessed: 2019-01-11.
- [55] Xie, P., Lu, X., Wang, Y., Su, J., & Li, M. 2013. An automatic approach to detect anti-debugging in malware analysis. In *Trustworthy Computing and Services*, Yuan, Y., Wu, X.,

- & Lu, Y., eds, 436–442, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-642-35795-4\_55.
- [56] Chen, P., Huygens, C., Desmet, L., & Joosen, W. 2016. Advanced or not? a comparative study of the use of anti-debugging and anti-vm techniques in generic and targeted malware. In *ICT Systems Security and Privacy Protection*, Hoepman, J.-H. & Katzenbeisser, S., eds, 323–336, Cham. Springer International Publishing. doi:10.1007/978-3-319-33630-5\_22.
- [57] Schiffman, M. 2010. A brief history of malware obfuscation: Part 1 of 2. [https://blogs.cisco.com/security/a\\_brief\\_history\\_of\\_malware\\_obfuscation\\_part\\_1\\_of\\_2](https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2). Accessed: 2019-01-17.
- [58] Gragido, W. & Elisan, C. 2012. Polymorphic and metaphoric threats and your cyber future. <https://www.blackhat.com/docs/webcast/polymorphis-and-metaphoric-threats-and-your-cyber-future.pdf>. Accessed: 2019-01-17.
- [59] Natani, P. & Vidyarthi, D. 2014. An overview of detection techniques for metamorphic malware. In *Intelligent Computing, Networking, and Informatics*, Mohapatra, D. P. & Patnaik, S., eds, 637–643, New Delhi. Springer India. doi:10.1007/978-81-322-1665-0\_63.
- [60] Maggi, F., Bellini, A., Salvaneschi, G., & Zanero, S. 2011. Finding non-trivial malware naming inconsistencies. In *Information Systems Security*, Jajodia, S. & Mazumdar, C., eds, 144–159, Berlin, Heidelberg. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25560-1\_10.
- [61] Sebastián, M., Rivera, R., Kotzias, P., & Caballero, J. 2016. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, Monrose, F., Dacier, M., Blanc, G., & Garcia-Alfaro, J., eds, 230–253, Cham. Springer International Publishing. doi:10.1007/978-3-319-45719-2\_11.
- [62] Kelchner, T. 2010. The (in)consistent naming of malware. *Computer Fraud & Security*, 2010, 5–7. doi:10.1016/S1361-3723(10)70007-5.
- [63] CARO. Welcome to the caro website. <http://www.caro.org/index.html>. Accessed: 2019-01-04;.
- [64] CARO. 1991. A new virus naming convention. <http://www.caro.org/articles/naming.html>. Accessed: 2019-01-04.
- [65] CARO. 1991. A virus by any other name - virus naming updated. <http://www.caro.org/articles/namingupdated.html>. Accessed: 2019-01-04; First publication 1991 with subsequent revisions later on.
- [66] Trend Micro. 2018. New threat detection naming scheme in trend micro. <https://success.trendmicro.com/solution/1119738>. Accessed: 2019-01-04.

- [67] Wikipedia. 2019. Machine learning. [https://en.wikipedia.org/wiki/Machine\\_learning](https://en.wikipedia.org/wiki/Machine_learning). Accessed: 2019-03-20.
- [68] Asiri, S. 2018. Machine learning classifiers. <https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623>. Accessed: 2019-04-09.
- [69] Wikipedia. Bayes' theorem. [https://en.wikipedia.org/wiki/Bayes%27\\_theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem). Accessed: 2019-04-09.
- [70] Wikipedia. Vapnik–chervonenkis theory. [https://en.wikipedia.org/wiki/Vapnik%E2%80%9393Chervonenkis\\_theory](https://en.wikipedia.org/wiki/Vapnik%E2%80%9393Chervonenkis_theory). Accessed: 2019-04-08.
- [71] Brownlee, J. 2016. Logistic regression for machine learning. <https://machinelearningmastery.com/logistic-regression-for-machine-learning/>. Accessed: 2019-04-09.
- [72] Wikipedia. C4.5 algorithm. [https://en.wikipedia.org/wiki/C4.5\\_algorithm](https://en.wikipedia.org/wiki/C4.5_algorithm). Accessed: 2019-04-09.
- [73] F-Secure. Threat descriptions. [https://www.f-secure.com/en/web/labs\\_global/threat-descriptions](https://www.f-secure.com/en/web/labs_global/threat-descriptions). Accessed: 2019-03-14.
- [74] Grini, L. S., Shalaginov, A., & Franke, K. 2018. Study of soft computing methods for large-scale multinomial malware types and families detection. In *Recent Developments and the New Direction in Soft-Computing Foundations and Applications: Selected Papers from the 6th World Conference on Soft Computing, May 22-25, 2016, Berkeley, USA*, Zadeh, L. A., Yager, R. R., Shahbazova, S. N., Reformat, M. Z., & Kreinovich, V., eds, 337–350, Cham. Springer International Publishing. doi:10.1007/978-3-319-75408-6\_26.
- [75] Brownlee, J. 2016. How to perform feature selection with machine learning data in weka. <https://machinelearningmastery.com/perform-feature-selection-machine-learning-data-weka/>. Accessed: 2019-04-02.
- [76] krmaxwell. 2015. Maltrieve. <https://github.com/krmaxwell/maltrieve>. Accessed: 2019-02-13.
- [77] Roberts, J.-M. 2012. Virusshare. <https://virusshare.com/>. Accessed: 2019-02-13.
- [78] No author given. 1999. Vx heaven. <http://83.133.184.251/virensimulation.org/>. Accessed: 2019-02-13.
- [79] doomedraven. 2019. disable win7noise.bat. [https://github.com/doomedraven/Tools/blob/master/Windows/disable\\_win7noise.bat](https://github.com/doomedraven/Tools/blob/master/Windows/disable_win7noise.bat). Accessed: 2019-02-14.

- [80] Kolosnjaji, B., Zarras, A., Lengyel, T., Webster, G., & Eckert, C. 2016. Adaptive semantics-aware malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Caballero, J., Zurutuza, U., & Rodríguez, R. J., eds, 419–439, Cham. Springer International Publishing.

## A Appendix

The following chapter contains the complete raw data used for the experiment.

### A.1 PEframe Python code

```
import os,sys
import json

rootdir = "~/Desktop/experiment/malware/IJCNN_10000files"

for subdir, dirs, files in os.walk(rootdir): #move through directories
    for file_name in files: #move through files
        os.system('peframe --json ' + os.path.join(subdir, file_name) + ' >
        ↪ temp_file.json') #call 'peframe' on current file

        with open('temp_file.json', 'r') as f:
            data = json.load(f) #load json data

            for key, value in dict.items(data["pe_info"]): #search array pe_info
                if key == "antivm_info" or key == "antidbg_info":
                    if len(value) != 0: #antidbg/antivm features found
                        os.system('rm ' + os.path.join(subdir, file_name)) #remove
                        ↪ file
```

### A.2 Powershell script to extract CPU usage and memory usage during malware execution

```
For($a=1;$a -lt 21;$a++){
$os = Get-WmiObject Win32_OperatingSystem
$pctFree =
↪ [math]::Round(($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100,2)
$memUsage = 100 - $pctFree #used memory space in percent

$proc = get-counter -Counter "\Processor(_Total)\% Processor Time"
$cpu=($proc.readings -split ":")[-1] #CPU usage in percent

"CPU: " + [string]$cpu | Add-Content C:\Users\Win7\Desktop\usage.txt #write cpu
↪ usage to file
"Memory: " + [string]$memUsage | Add-Content C:\Users\Win7\Desktop\usage.txt
↪ #write memory usage to file
```

```
Start-Sleep -Milliseconds 500 #sleep for 0.5s
}
```

### A.3 Python code for pre-processing task. Find errors, copy memory features into report and group all reports based on family.

```
import json
import os,sys

rootdir = '~/.cuckoo/storage/analyses/'

#create folder
folder = [('agent',2,868), ('hupigon',868,1175), ('obfuscator',1175,1633),
→ ('onlinegames',1633,2622), ('renos',2622,3608), ('small',3608,4501),
→ ('vb',4501,5574), ('vbinject',5574,6523), ('vundo',6523,7307),
→ ('zlob',7307,8307)] #list of tuples (family_name,start_taskID,end_taskID)

os.system('mkdir ~/Desktop/malwareFamiliesReports/')
for i in range(0,10):
    os.system('mkdir ~/Desktop/malwareFamiliesReports/' + folder[i][0]) #create
    → folder for malware family

for i in range(2,8307):
    #delete errors
    report_dir = rootdir + str(i) + "/reports/report.json"
    summary_exists = False
    error_found = False
    with open(report_dir, 'r') as f:
        data = json.load(f)

    for key, value in dict.items(data["debug"]):
        if key == "errors":
            if len(value) != 0:
                error_found = True
    if error_found == True:
        os.system('rm ' + report_dir)
        continue
    for key, value in dict.items(data["behavior"]):
        if key == "summary":
            if len(value) != 0:
                summary_exists = True
    if summary_exists == False:
        os.system('rm ' + report_dir)
        continue
```



```

#memory features into report.json
for subdir, dirs, files in os.walk(rootdir + str(i) + "/files/"):
    for file_name in files:
        if file_name[-9:] == "usage.txt":
            with open(os.path.join(subdir, file_name), 'r') as uf:
                ram = []
                cpu = []
                for ln in uf:
                    if ln.startswith("Memory:"):
                        ram.append(float(ln[8:-2]))
                    if ln[0].isdigit() == True:
                        cpu.append(float(ln[:-1]))
            uf.close()
            if len(ram) == 0 or len(cpu) == 0:
                break
            peakCPU = max(cpu)
            peakRAM = max(ram)
            averageCPU = sum(cpu)/len(cpu)
            averageRAM = sum(ram)/len(ram)
            new_dic = {'peakCPU':peakCPU, 'peakRAM':peakRAM,
                ↪ 'averageCPU':averageCPU, 'averageRAM':averageRAM}
            with open(report_dir, 'r+') as rf:
                data = json.load(rf)
                data["behavior"]["summary"].update(new_dic)
            with open(report_dir, 'w') as rf_out:
                json.dump(data, rf_out, indent=4)
            break

#move to directory
for x in range (0,10): #start_taskID to end_taskID
    if i >= folder[x][1] and i < folder[x][2]:
        os.system('cp ' + report_dir + ' ~/Desktop/malwareFamiliesReports/' +
            ↪ folder[x][0] + '/report_' + str(i) + '.json') #copy report to
            ↪ folder
        break

```

#### A.4 Python code to extract all dynamic features and create Weka file

```

import json
import os,sys

rootdir = '~/Desktop/reports/'

features = []

for subdir, dirs, files in os.walk(rootdir):

```

```

for file_name in files:
    with open(os.path.join(subdir, file_name), 'r') as f:
        data = json.load(f)

    for key, value in dict.items(data["behavior"]):
        if key == "summary":
            for key, val in dict.items(value):
                if str(key) not in features:
                    features.append(str(key))

s.close()

k = open('~/Desktop/malware_classification.json', 'a')
k.write('{"header" : {"relation" : "malware","attributes" : [\n')

for y in features:
    k.write('{"name" : "'+str(y)+'", "type" : "numeric", "class" : false, "weight"
    ↪ : 1.0}\n') #attributes
k.write('{"name" : "family", "type" : "nominal", "class" : false, "weight" :
    ↪ 1.0, "labels"
    ↪ : ["agent", "hupigon", "obfuscator", "onlinegames", "renos", "small", "vb", "vbinject", "vundo", "zlob"]}]')
    ↪ : [\n\n')

for subdir, dirs, files in os.walk(rootdir):
    for file_name in files:
        k.write('{"sparse" : false, "weight" : 1.0, "values" : [')
        with open(os.path.join(subdir, file_name), 'r') as f:
            data = json.load(f)

        for key, value in dict.items(data["behavior"]):
            if key == "summary":
                for x in features:
                    if x == "averageCPU" or x == "averageRAM" or x == "peakCPU"
                    ↪ or x == "peakRAM":
                        feature_found = False
                        for key, val in dict.items(value):
                            if key == x:
                                k.write('"' + str(val) + '",')
                                feature_found = True
                        if feature_found != True:
                            k.write('"?",')
            else:
                feature_found = False
                for key, val in dict.items(value):
                    if key == x:
                        k.write('"' + str(len(val)) + '",')

```

```

        feature_found = True
        if feature_found != True:
            k.write('"0",')
            k.write('"' + str(os.path.basename(subdir)) + '"')
            k.write(']]\n')
k.write(']]')
k.close()

```

## A.5 Python code to extract all entries from static feature dataset

```

import csv, pickle
import json,os,sys

rootdir = '~/Desktop/reports_1/'

#extract all MD5 hash values of the used 7,009 malware samples
for subdir, dirs, files in os.walk(rootdir):
    for file_name in files:
        s = open('/home/thilod/Desktop/md5.txt', 'a')
        with open(os.path.join(subdir, file_name), 'r') as f:
            data = json.load(f)

            for key, value in dict.items(data["target"]):
                if key == "file":
                    for key, val in dict.items(value):
                        if str(key) == "md5":
                            s.write(str(val) + "\n")
                            break

s.close()

#extract static feature entries based on md5 hash value
path = "/home/thilod/Desktop/md5.txt" #file with all 7,009 md5 hash values

with open("~/Desktop/data_raw-no_header.csv") as csvfile: #plain dataset file
    data = list(csv.reader(csvfile))

lines = []
final = []

with open(path, "r") as f:
    for line in f:
        lines.append(line.strip())

for i in lines:
    for x in data:
        if i == x[0][2:]:
            final.append(x)

```

```

        break

with open("/home/thilod/Desktop/static_features.txt", "w") as sf: #txt file has
    → to be saved as csv file afterwards
    for item in final:
        out = ','.join(item)
        sf.write(out)
        sf.write("\n")

```

## A.6 Confusion Matrix for Random Forest Classifier and oneR-based Feature Selection

```

from mlxtend.plotting import plot_confusion_matrix
import matplotlib.pyplot as plt
import numpy as np

multiclass = np.array([
    [390,19,34,9,2,46,57,35,3,5],
    [29,184,26,0,3,3,7,4,3,1],
    [39,33,291,0,3,15,19,16,1,0],
    [15,2,4,886,0,5,6,5,1,4],
    [6,1,3,0,750,1,2,3,8,1],
    [62,6,11,4,1,753,20,9,0,3],
    [25,0,11,4,1,16,787,95,1,0],
    [13,0,6,0,0,3,59,835,2,0],
    [4,1,3,0,7,1,3,2,302,0],
    [5,3,0,5,0,2,5,0,2,954]])

labels = ['agent', 'hupigon', 'obfuscator', 'onlinegames', 'renos', 'small',
    → 'vb', 'vbinject', 'vundo', 'zlob']
fig, ax = plot_confusion_matrix(conf_mat=multiclass,
    → colorbar=True, show_absolute=False, show_normed=True)
plt.ylabel("True label")
plt.xlabel("Predicted label")
tick_marks = np.arange(len(labels))
plt.xticks(tick_marks, labels, rotation=45)
plt.yticks(tick_marks, labels)

plt.show()

```