Doctoral theses at NTNU, 2023:11

Sergii Banin

Malware detection and classification using low-level features

Doctoral thesis

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of Philosophiae Doctor
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of Science and Technology

Sergii Banin

Malware detection and classification using low-level features

Thesis for the Degree of Philosophiae Doctor

Gjøvik, January 2023

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

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

ISBN 978-82-326-6061-2 (printed ver.)
ISBN 978-82-326-6679-9 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2023:11

Printed by NTNU Grafisk senter

It is an unfortunate fact that the bulk of humanity is too limited in its mental vision to weigh with patience and intelligence those isolated phenomena, seen and felt only by a psychologically sensitive few, which lie outside its common experience.

— H. P. Lovecraft, The Tomb

Abstract

Nowadays, computers and computer systems are involved in most areas of our lives. Employees and users of manufacturing and transportation, banking and healthcare, education, and entertainment rely on computers and networks which allow for better, faster, and often remote control and access to various services. As it often happens - commodity comes with unwanted side effects. The computers can be misused by malicious actors which tend to disrupt operations, spoof, steal or destroy sensitive data or gain remote control over the victim systems. These and other malicious actions are often made using malicious software or *malware*. Thereby, malware detection and analysis play a significant role in the Information Security domain.

Various methods are used for malware analysis and detection. They can be roughly divided into two major groups: static and dynamic. Static methods rely on features derived from malware without it being launched: strings, section names, entropy, etc. Dynamic methods rely on dynamic or behavioral features which are extracted when malware is launched. Often, static features are easier to extract than behavioral properties. However, it is easier for malware authors to alter static features in order to thwart static malware detection. Information Security researchers have studied the applicability of different sources of behavioral features: process activity, file activity, network activity, etc. Such behavioral features can be called *high-level* features. Malware authors also tend to alter them: change names of processes and dropped files, change IP addresses, and so on. However, malware is always executed on the system's hardware. Therefore, features that emerge directly from hardware can also be used as a source of behavioral features. Such features are called hardware-based or *low-level features*: memory activity, executed opcodes, hardware-performance counters, etc. Since it is impossible for malware to avoid execution on the system's hardware, in this Thesis we focus on the applicability of low-level features for malware detection.

Researchers have already shown, that such low-level features as opcodes and hardware performance counters can be used for malware detection. However, to the author's knowledge, no one has used memory access patterns for malware detection prior to the beginning of our work. Thus, in this Thesis, we focus on the applicability of memory access patterns for malware detection and analysis. In our work, we present a methodology and experimental evaluation of malware detection and classification using memory access patterns. We show that memory access patterns can be used for malware detection and classification. Moreover, during our research we found, that it is possible to detect and classify malware based on the memory access patterns before launched malware reaches its Entry Point. This means, that we found a way to stop malware that has been already launched before it has a chance to conduct any malicious actions. We also show, how low-level features can be correlated with their high-level counterparts. While conducting our research, we extensively used Machine Learning (ML) methods. In this Thesis, we use various methods to analyze the performance of ML models, which can be helpful for other researchers.

Preamble

This thesis is submitted in partial fulfilment of the requirements for the degree of Philosophiae Doctor (PhD) at the Norwegian University of Science and Technology (NTNU). This work has been performed at the faculty of information technology and electrical engineering, department of information security and communication technology at NTNU from 2016 until 2022.

This research was carried out under supervision from associate professor Geir Olav Dyrkolbotn and professor Katrin Franke. The research leading to these results has received funding from the Center for Cyber and Information Security, under budget allocation from the Ministry of Justice and Public Security.

Sergii Banin

Acknowledgements

I took an overly long and weird path during the completion of my PhD. The Thesis you are now holding in your hands was written during the most unforeseeable global events that influenced everyone around including me. These events made me become severely unproductive and incapable of focusing on writing. Nevertheless, along my path to completion, I was surrounded and influenced by wonderful people who willingly or unwillingly helped me to get to the finish line.

First of all I'd like to thank my first supervisor Dr. Geir Olav Dyrkolbotn. It is only now I understand how much courage it probably took to accept me as his PhD candidate. The topic of my Thesis was completely new at that time, and thereby no one could predict the probability of success in the end. I know, our collaboration was not always productive. Especially toward the end. And it was completely my fault. Nevertheless, it was an honor to work with him. By carefully placing guardrails around he managed to keep me in the proximity of the right path and directed me in the right direction. Secondly, I'd like to say a big thank you to my second supervisor Dr. Katrin Franke. Even though we didn't cooperate as often, her common question "Why?" was always somewhere in my head contributing to increased attention to the details. Moreover, it was she who caught me in the nets of academia and I didn't regret it a single time.

I also wish to thank my parents, my mother Yevheniia Andriienko, and father Oleksandr Banin who supported my endeavours even being thousands of kilometers away. Even though I tried not to bother them too much, I always knew that I'll be supported and given advice no matter how I was doing in any given moment. I am also grateful to my grandparents: grandmother Tetyana Andriienko who was always happy just to see me, and grandfather Igor Andriienko, an engineer himself, who always encouraged me to push it through. I would also like to mention my granduncle Dr. Valentyn Lobodyuk who served as an example of a Doctor in the family.

Moreover, it was a great pleasure to work with my colleagues Dr. Kyle Porter and Dr. Jan William Johnsen. I appreciate the possibility to bounce some thoughts and ideas off you. Special thank you goes to Dr. Andrii Shalaginov for useful insights and advices as he was way ahead on his path to PhD.

Very special thank you goes to Dr. Olga Ogorodnyk. For inspiring to accept the offer for the PhD candidate position. For pursuing your path to PhD together with me. And for sharing the adventure times of your life with me.

I'd also wish to thank my friends Marina Shalaginova, Christoffer Vargtass Hallstensen, Dr. Anastasiia Moldavska, Juan Victor Abreu-Peralta, and Dr. Radina Stoykova. An additional thank you goes to Andrii Sukhanov, Oleksandr Bukalo, and Dr. Kyrylo Kofonov for making my visits back home in Kyiv the way they should have been. In these frightful times of war I cannot be more grateful to Sergiy Chekmarev who voluntarily took arms and is defending our motherland Ukraine: stay safe!

Finally I'd like to thank the Norwegian University of Science and Technology, the NTNU in general, and its campus Gjovik in particular. I appreciate the opportunity to use the laboratory, cafeteria, coffee machine, and servers. I am grateful to the administrative staff for helping me out with bureaucracy and formalities. Also, I wish to say thank you to the IT and Digital Security divisions for keeping the necessary infrastructure safe, up, and running. Special thank you goes to Lars Erik Pedersen who is a true wizard of virtualization and Linux. I appreciate fixing issues with the virtual server and provisioning extra terabytes on the go while I was running out of storage space: three papers were made based on data stored in that additional partition.

Contents

Tables

Figures

Part I

Introduction

Chapter 1

Introduction

Different electronic devices and computer systems are constantly involved in various aspects of the life of modern society. Computers are used for control in transportation, electricity production, manufacturing, and so on. People rely on computers for storing, processing, and transmission of personal and sensitive information. It is no surprise that criminals and malicious actors try to misuse computers and systems to steal personal data or disrupt industry operations. Thereby, Information Security gained an important role in the everyday life of digitized society. As older vulnerabilities and attacks become obsolete thanks to Information Security specialists, new attack vectors are being discovered by malicious actors. This race of arms shows a need for constant improvement of existing and creation of new security mechanisms. As malware is often involved in cyber-attacks, malware detection and analysis is an important part of the Information Security domain. In this Thesis, we present a novel approach for malware detection and analysis.

1.1 Background and Motivation

Malware is malicious software that was created in order to perform unwanted, often illegal actions in computers and computer systems. Malware can be made in the form of a standalone executable with only malicious functionality. It can also be injected into the legitimate executables to perform a malicious activity in the background while the user continues to use the "normal" application. Execution of malware in the victim system can be a part of a cyber-attack. The effects of attacks on computers and systems vary from simple unwanted ads in the browser of a private person's PC to the disruption in the electricity supply grids for hundreds of thousands of people[38]. The field of Information Security defines and implements various practices aimed at defending both private and corporate customers from malicious actors. In order to develop appropriate defense mechanisms and tools, it is important to understand which steps an adversary might take to achieve one's

goals. Attacks on systems and services can be divided into seven stages of a cyber-attack kill chain[26]

1. Reconnaissance - adversary identifies a target and tries to obtain as much knowledge about it as possible.
2. Weaponization - creation of malware that is tailored to the properties of the victim system.
3. Delivery - transmission of malware into the victim system.
4. Exploitation - malware exploits a vulnerability in the system or user behavior.
5. Installation - malware achieves persistence in the system if it is necessary.
6. Command and Control - malware gives the adversary an opportunity to access the system remotely.
7. Actions on Objective - adversary performs malicious actions in the system.

Malware is often used in the victim systems on Stages 4 to 7 from the list above. Moreover, Stage 3 (Delivery) might also involve malware that was already placed in the system before the major attack will happen. Normally, it is desirable to not let malware be placed in the system in the first place. Thereby, it is important to detect malware in the system as soon as possible. This is often done through various policies and tailored user privileges. For example, it may be forbidden to use personal flash drives on the computers of the company, or a network firewall may be set up to restrict connections to certain Internet addresses. However, both users and system administrators are human beings, thus tend to make mistakes. If malware made its way into the system, it is preferable to detect it before it becomes active: before it is launched.

To identify malware that is stored in the system and not being launched, a set of *static* malware detection techniques is used. Static malware detection relies on static properties of the executable file[41]: hash sums, PE32 header, strings, byte and opcode sequences, etc. Static properties or *features* emerge from the file itself and are relatively easy to derive. Static malware detection techniques often rely on databases of previously seen malware. However, various anti-detection techniques such as obfuscation and encryption are used by malware authors to make the same malicious executable (with the same functionality) have a different static appearance. For example, a change in one bit in the malicious file will make its hash sum completely different. Nevertheless, malware becomes malicious only when it is executed. When a malicious executable is launched it reveals its functionality. This makes it possible to collect a *behavioral trace*: a record of actions

performed by malware in the victim system. For example, API calls, opcode sequences, memory activity, dropped, read, and written files as well as launched processes and accessed network resources can be used to describe the behavioral properties of malware. Such properties are used by the *dynamic* malware detection methods. Dynamic malware detection relies on these characteristics and may help when static methods fail to identify a new variant of previously seen malware. Obviously, malware authors try to make such detection harder as well. For example, they can use different network addresses or drop files with randomly generated names. Malware can also be made to detect anti-virus solutions and disrupt their work first.

Both static and dynamic detection methods rely on features: properties of executables that allow to distinguish between malicious and benign software. It is often the task of a malware analyst to decide which properties are better to use for such a task. Often malware incorporates various anti-analysis techniques that make analysis harder. For example, encryption and obfuscation make static analysis harder. In its turn, anti-debug and anti-VM techniques are aimed at making dynamic analysis harder. Dynamic malware analysis is normally made inside an isolated and controlled environment: often a Virtual Machine (VM) that has a set of monitoring tools in place. Regardless of the anti-detection and anti-analysis techniques malware is made to be *executed*. While everything that is executed in the system is executed on the system's *hardware*. Thus, analysis of the hardware activity produced by malware may be a reliable source of information and distinctive properties. Hardware activity can be used to extract hardware-based or *low-level* features. Low-level features consist of but are not limited to: CPU instructions (opcodes), memory activity, disk activity, GPU activity, and so on. In this Thesis, we split behavioral features into high- and low-level features. For example, API-calls and file activity are the high-level features while CPU and memory activity are the low-level features. In this Thesis, we study the applicability of low-level features for malware detection and analysis and use hardware as a source of relevant information as shown in Figure 1.1.

Malware analysis is a set of techniques aimed at revealing the goals of malware and the ways these goals are achieved. Based on the malware goals, or *what* the malware does, malware samples can be divided into malware *types*. In its turn, based on *how* the malware achieves its goals malware can be divided into malware *families*[11]. The process of assigning malware into categories is called *malware classification*. Malware classification is important for improvements of the security mechanisms. For example, knowledge about malware types allows strengthening security policies and pre-attack defense mechanisms. In its turn, knowledge about malware families helps in post-attack actions as security specialists will have a better understanding of what has been changed in the victim system. In this Thesis,

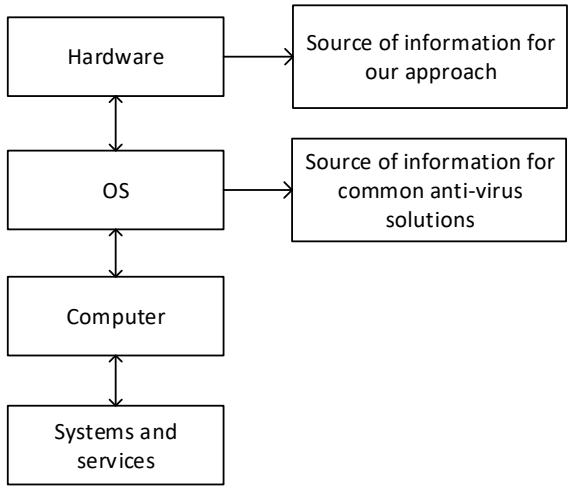


Figure 1.1: Source of information

we also explore, how low-level features can be used for malware classification.

One of the problems of malware analysis and detection is large amounts of malware in the wild. For example, the VirusShare [48] resource contains more than 36.6M samples by the end of the year 2020. Every day hundreds of thousands of new samples are discovered[8]. This makes manual analysis infeasible, thus forcing analysts and information security companies to use *machine learning*. Machine Learning (ML) is a set of statistical methods aimed at deriving knowledge from large amounts of data by finding common characteristics of analyzed objects[29]. ML methods can be roughly divided into supervised and unsupervised methods. The supervised methods can be divided into classification and regression methods. In malware detection and analysis it is often important to perform *classification*: classify malware into categories (multinomial classification) or distinguish between benign and malicious executables (binary classification). ML methods rely on *features*, certain properties of the analyzed objects that the user believes can carry information necessary for classification. When a type of features is decided on, the values of such features are collected from samples of the dataset: data collection. After the data has been collected an ML algorithm is used to build a *model*: statistical representation of the dataset. We elaborate on the topic of Machine Learning in the Section 2.1.6.

One of the problems with the utilization of low-level features is that it may be challenging to extract and record low-level activity. This may be one of the reasons

why hardware-based features did not receive enough attention from the research community by the beginning of our research. Moreover, a certain type of low-level features such as memory access patterns has never been used for malware detection and analysis before the time research leading to this Thesis was started. These factors motivated and shaped our research.

Originally, we were planning to explore the applicability of various low-level features for malware detection and analysis. However, as we began our studies we faced significant difficulties which made us narrow our focus to the study of memory access patterns: a previously unexplored type of low-level features. First of all, in this Thesis, we describe, how memory access patterns can be used for malware detection. We outline the necessary amount of memory accesses needed for the analysis. Describe our ways of data preprocessing. We also show, which ML algorithms trained on memory access patterns show the best performance in malware detection. Moreover, we elaborate on how one can deal with huge amounts of unique memory access patterns with help of several different feature selection techniques. The amounts of data needed to be processed forced us to develop and study our own feature selection method which is described in paper P5[10]. Later, having a basic understanding of memory access patterns in malware detection, we study the applicability of those for malware classification. In our paper [11] we evaluate the performance of ML models trained to distinguish between malware families and types. In that work, we present a valuable way of analysis of classification performance with help of subcategories. Moreover, there we present the definitions of malware family and malware type, as such definitions were rarely present in relevant literature at the time of writing. As we went deeper into the topic we realized, that memory access patterns are not human-understandable. Thus, we performed an attempt to correlate our low-level features with more human-understandable API-calls - high-level features in P3 [12]. During that research stage, we found, that API calls and memory access patterns allow for better classification performance if used together. Moreover, we discovered that most of the data we analyzed emerged from before the Entry Point (BEP). Basically, this meant, that we were able to detect and classify malware before it has any chance to perform any malicious actions, as the logic of executable is executed after the Entry Point (AEP) as shown in the Figure 1.2. We took this finding and developed a BEP-AEP approach in malware analysis that is described in [14]. With this approach, we deliberately differentiate between behavioral activity produced by launched malware BEP and AEP. In that work we have shown, that malware detection is possible based solely on memory access patterns produced BEP. This is one of our major contributions, as we push the last line of defense and create an ability to stop launched malware from doing any potential harm to the victim system. The last stage of our research was aimed at the study of the applicability of BEP memory access patterns for

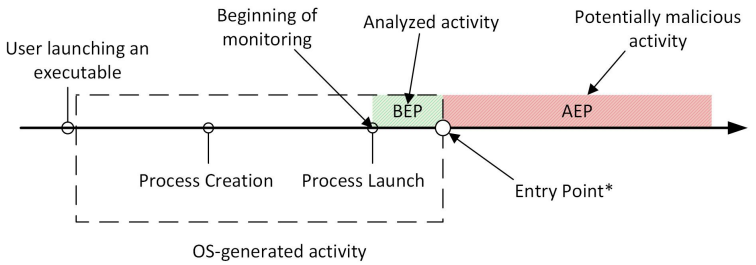


Figure 1.2: Source of information

the detection of previously unseen malware. In P6 [13] we explored how models trained on earlier malware samples are capable of detecting malware that was first discovered months later after the model update.

1.2 Aim and Scope

The aim of this research is to explore the applicability of low-level features for malware detection, classification, and analysis. Within our study, firstly, we focus on the use of memory access patterns for malware detection. Specifically, in several papers, we investigated which malware detection and classification performance can be achieved using memory access patterns. One paper is dedicated to investigating the possibility to detect a launched malware before the newly created process reaches its main module. As low-level features are often meaningless to human analysts we have also performed an attempt to build a semantic bridge between low- and high-level features. Moreover, during our research, we show, that combining memory access patterns with high-level features (API calls) allows to improve the performance of ML models trained to classify malware into categories and types.

1.3 Research questions

Execution of any executable results in the activity on the system's hardware. This activity may be a source of information relevant for the malware detection. In this thesis we focus on the study of the applicability of low-level features for malware detection and analysis and the main research question can be stated as follows:

How can low-level features be used in malware detection and analysis?

While working to answer the main research question we outlined the following subquestions:

1. What are the best practices of low-level features applications for malware analysis?

2. How can low-level features be used for malware detection?

   (a) What is the performance of our approach?

3. How can low-level features be used for malware classification?

   (a) What is the performance of our approach?

4. How can malware detection and analysis be improved by the use of the low-level features?:

   (a) How low-level features can improve understanding of malware detection and analysis and contribute to better domain knowledge?

   (b) How can low-level features improve malware detection capabilities?

   (c) To what extent low-level features can improve malware classification accuracy?

1.4 Outline of the Thesis

The remainder of the Thesis is arranged as follows. Part I consists of theoretical background and a condensed description of our research and findings:

- In Section 2 we provide a theoretical background necessary for understanding the contributions of the Thesis. First, we briefly describe computer operations. Later, we elaborate on the concepts of malware and malware analysis. Then, we justify the importance of low-level features analysis. Later, we elaborate on the topics of Machine Learning and its application in malware analysis. We conclude this section we an overview of the papers which present research on the topic of the application of low-level features for malware analysis.

- In Section 3 we provide a brief description of the published articles. We make a short description of each paper and its findings. We also show, how papers help us to answer research questions from Section 1.3

- In Section 4 we outline our contributions. In this section, we describe our most valuable findings and explain how they contribute to the knowledge area.

- In the Section 5 we describe theoretical implications and practical considerations. There we justify important theory necessary for future research. We also explain practical aspects of our research that might be useful in future research as well. Later, we discuss ethical and legal aspects that may arise in the research similar to ours. We conclude this section by outlining the limitations of our research and possible directions for future research.

Part II consists of the papers that present details and results of our research.

Chapter 2

Theoretical Foundations

This chapter is dedicated for providing the reader with a theoretical background necessary for an understanding of the rest of the Thesis. We begin with introducing the Basic Concepts where we elaborate on: general overview of the topic; computer operations basics; malware; dynamic malware analysis; low-level behavioral features; machine learning; assessing the quality of machine learning aided malware detection; feature selection. We later provide an overview of the articles related to the topic of low-level features utilization in malware detection and analysis.

2.1 Basic Concepts

In this section, we provide concepts that are necessary for the reader to understand the remainder of the Thesis.

2.1.1 General overview

Before the thorough elaboration on the basic concepts, we present a general overview of the necessary terms and their semantic connections. In the following subsections, we extensively talk about malware. **Malware** or **malware sample** is an **executable** file that is capable of performing malicious or **unwanted** activity or actions. In this Thesis, we use the term *malware* as the opposite of **goodware** or **benign** software: executables that don't perform malicious activity or actions.

Malware analysis is a set of tools and methods aimed at understanding malware functionality and revealing **properties** or **features** common for many malware samples. Such features are then used in **malware detection** - methods aimed at distinguishing between malicious and benign executables. Moreover, such features can be used for **malware classification** - methods used to split malware into different categories.

Malware analysis is usually divided into **static** and **dynamic** analysis. Static

analysis is done without launching the malware, while dynamic involves it. While static analysis is relatively safe to perform without additional restriction, dynamic malware analysis is normally done in a safe and controlled environment. Often **virtual machines** or VMs are used to improve control and isolation of the analysis process. A desired operating system - **OS** is normally installed on the VM. We often refer to the OS as *system* and to the OS installed on the VM as **guest** system. The system that runs the VM is referred to as **host** system.

In order to perform malware detection, it is important to find which features can support the distinguishing between malware and goodware. For example, a presence of a certain URL or IP address among the strings found in the executable can be a sign of maliciousness of the file. For instance, a string *KRAB-DECRYPT.txt* found in the file can be a sign of this file being from a malware family *GandCrab*. To find the distinguishing features, they have to be extracted from benign and malicious files. This can result in large quantities of data that are not suitable for manual analysis. Thus, statistical methods such as **machine learning** (ML) are used to process and optimize such data for further use. ML methods are used to find the most useful features and train ML **models** that are later used for malware detection or classification. After the **feature extraction** and before ML models training it is often necessary to reduce the number of features by utilizing one of the **feature selection methods**. Selected features are then used as inputs for the ML models. Trained ML models are assessed using various **measures** such as true positive rate (TPR), false positive rate (FPR), accuracy and so on.

2.1.2 Computer operations

In this subsection, we describe the basic principles of computer operations. The majority of modern computers are based on the von Neumann architecture[33] and can be roughly depicted as shown in Figure 2.1. The von Neumann architecture implies the presence of the following components: input and output devices; processing unit with arithmetic logic unit and registers; control unit with instruction register and program counter; memory that stores data and instructions; external mass storage. These components are called *hardware*.

Nowadays, control and processing units are placed inside the CPU: Central Processing Unit. The CPU is responsible for arithmetic, logical, and control operations. Every CPU has a predefined set of operations that it can execute. These operations are often referred to as CPU instructions or *opcodes* - operational commands. In the CPU opcodes are stored in their binary representation which is often referred to as *machine code*. However, they have more human-understandable text representation: *mnemonic codes* or *assembly instructions*. The execution of opcode often involves operations on data. The data is normally stored in the main memory or CPU registers - special components of the processor used for data

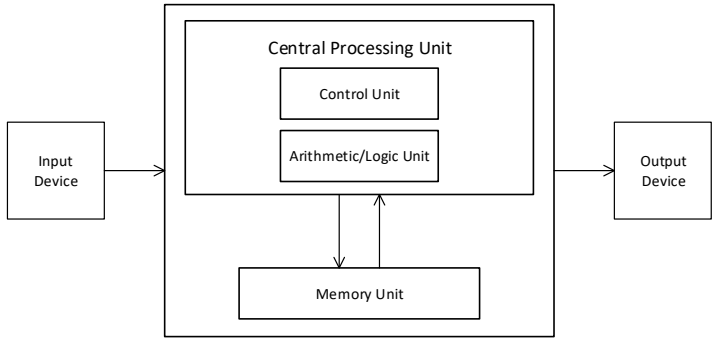


Figure 2.1: von Neumann computer Arcitecture

storage.

Data stored in the CPU registers is the nearest to the processor's control, arithmetic, and logical units. It is directly accessible for operations with no access latency[46]. The data in the registers can be either the result of the execution of previous opcodes or loaded from the main memory. Nowadays computers (still) have a dedicated hardware component that stores volatile data - Dynamic Random-Access Memory (DRAM). The latency for accessing this data is higher than the one in registers but faster than the one on the permanent storage components (SDD, HDD). As registers often can't store all the necessary data CPU often has to access memory. Despite all the technological progress, DRAM units are not fast enough to avoid performance bottlenecks. Therefore modern CPUs contain several levels of cache - intermediate storage components in the CPU. Their task is to store parts of the most accessed data from the main memory. The access latency of the cache is tens of times lower than the one of main memory[19]. For simplicity, the author of the Thesis often addresses main memory as just *memory*. Instructions that require interaction with memory are normally executed under the context of a certain process. A process has access to certain parts of the memory. These parts of the memory might not necessarily have a continuous range of physical addresses. To simplify the creation and execution of programs operating system provides a process with a *virtual memory*: a mapping between real physical addresses and a range of continuous range of virtual addresses. We briefly elaborate on the terms such as operating system or process further in this section.

When an opcode is executed in the CPU, it is translated into the set of *microoperations*: basic CPU instructions that are responsible for loading and storing data, interaction with the arithmetic logical unit, memory address calculation, and so on. Microoperations are executed on execution ports that are dedicated

to the previously mentioned functions of the microoperations. An opcode can have zero or several arguments which are used for the opcode execution. Arguments can contain data, registers, and memory addresses of where the data is stored, as well as register or address of the result buffer: a place where the result of the execution of opcode is stored. Malware, as well as any other executable file, contains binary representation of opcodes. Whenever an executable is running in the system, opcodes are first loaded into the memory, and then, in the order of execution, are loaded and executed in the CPU. Some opcodes use only CPU registers e.g. $XOR\ EBX, EBX$, while others use memory e.g. $MOV\ [memory\_address], EAX$. Opcodes that operate with memory use *load* and *store* microoperations. Execution of *load* microoperation will result in *read* from the memory, while execution of *store* - write to the memory. For the reader of this Thesis, it is important to understand, that presence of a cache is transparent to the opcodes. Thereby if necessary data is not stored in the cache and the memory management unit needs to access main memory, there will be still registered only one *read* memory access operation.

CPUs are built based on different architectures such as x86, x86_64 ARM, MIPS, PIC, and so on. An architecture, or *Instruction Set Architecture* (ISA), is a model of an abstract processor used to implement a CPU in hardware. The ISA describes various properties of a CPU such as: supported data types, available registers, input and output operations, memory management, and so on. Thereby, processors built with different ISAs have different instruction sets. At the same time, processors built with the same or compatible ISAs will have fully or partially compatible instruction sets. For example, an old Intel Pentium 4 CPU built with x86 Northwood architecture has its instruction set compatible with modern Intel Core i9 CPU built with x86_64 Comet Lake architecture. In this Thesis, we run our experiments on Intel x86_64 CPU.

The "basic" instruction set of a modern x86 compatible CPU contains different types of instructions such as logical, control, data handling and mathematical. Furthermore, modern x86 compatible CPU have various extensions of their "basic" instruction set which are used for specific tasks such as encryption, advanced data and mathematical operations, vector operations, random number generation and so on.

The opcodes, when executed sequentially for a specific task, form a computer program. Computer programs are written using programming languages: formal languages that contain various instructions or commands aimed to implement different algorithms or *logic*. Computer programs can be written directly in machine codes. Such program can be directly loaded into the CPU for execution. However, it is easier for human to use text representation of opcodes or assembly language. Program written in the assembly language is passed to the *assembler*: another pro-

gram that, besides other tasks, translates assembly instructions into the machine codes. Even though instructions allow to fully utilize capabilities of a CPU, it is quite challenging to implement complex programs purely in assembly language. For example, programming complex mathematical operations that are not implemented in hardware will require big amounts of opcodes to be written by a human. Assembly and machine codes are considered as *low-level* programming languages. Low-level programming languages provide almost no abstraction over the instruction set of a CPU. Thereby, low-level code may be difficult to understand, hence difficult to improve, modify or debug. That's why it is often more convenient to use *high-level* programming languages such as C, C++, C#, Java, Python, and so on. Such languages provide a high level of abstraction over the CPU instruction set. High-level programming languages provide a set of human-understandable instructions. High-level instructions can substitute many low-level instructions. Thereby, creation, modification, and debugging of high-level code is often faster and easier.

When a program is written in a high-level language, its instructions can not be loaded directly into the CPU. The program has to be *compiled* into the binary form or *executable*: a file that contains machine codes. An executable contains the logic conceived by the creator as well as some of the necessary resources and various directives needed for the execution. However, in order to start the execution of the compiled program, its instructions have to be loaded into the CPU. Nowadays, computers are operated by operating systems (OS). OS is the intermediate software that simplifies human interaction with hardware by providing various interfaces. The formats of executable files are often specific to the OS. Most of the executable files used on Windows are of a Portable Executable (PE) format. Files of PE format contain various headers and sections. Headers contain information necessary for the OS to properly load executables before launch and information about the sections. In their turn, sections contain binary machine codes, data, and resources used by the machine codes during the execution. Machine codes are normally stored in the executable section, while data and resources have separate dedicated sections as shown in Figure 2.2.

When a user wants to execute a program it is the task of the OS to "feed" it to the CPU. The part of the operating system responsible for this task is called *loader*. For example, as shown in Figure 2.3, the loading of an executable on Windows OS is a multistage process. First, the loader reads an executable file. There it finds and interprets various directives that are used to create a process object. Later, the loader fills the process object with data from an executable and prepares the execution environment. Sections of an executable are mapped into the virtual memory. Necessary OS resources and libraries are also loaded into the memory, while links to the respective memory regions are placed in the process object. Before launch-

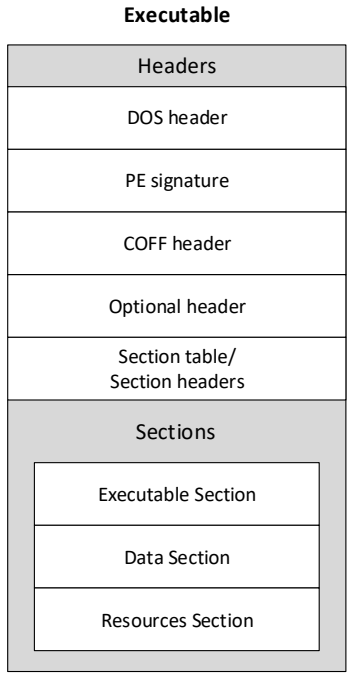


Figure 2.2: Portable Executable file format

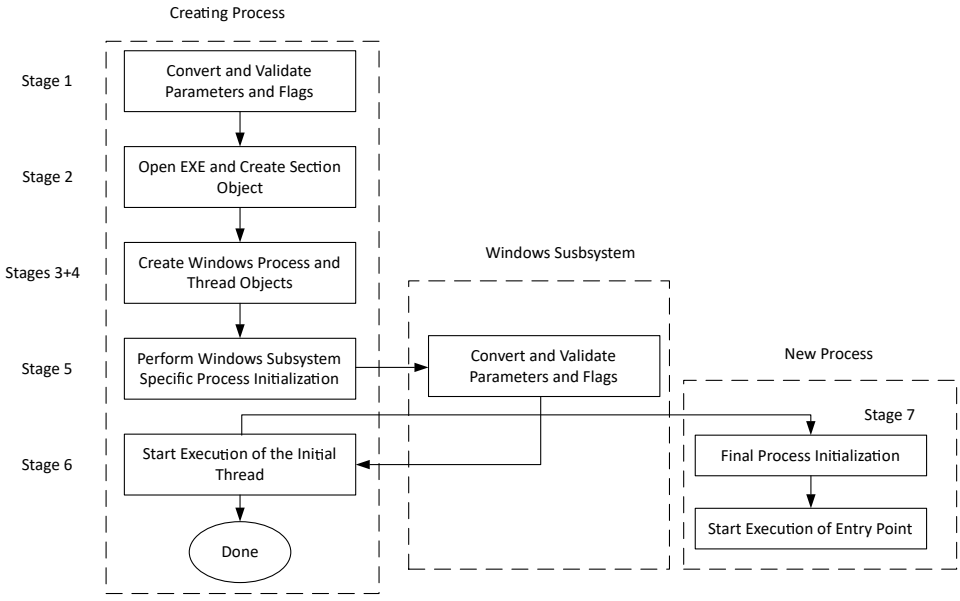


Figure 2.3: Windows process creation flow [14] (P4), inspired by [50]

ing a process, the loader creates and initializes a *thread*: the smallest object that contains executable instructions and can be managed by OS. Then, when the initial thread is initialized and necessary resources are loaded, a newly created process object is launched creating a *process*: an instance of a program that can comprise multiple threads. A new process performs final process initialization: e.g. loads necessary libraries and performs various additional checks[50]. Up to this point, none of the instructions from the executable were executed. In the end, the address of the first instruction of an executable is loaded into the CPU, and execution of the logic from an executable begins.

At this point CPU begins to execute instructions that represent the functionality of a computer program or *software*. Users of the computers mostly interact with software as it helps to input data into the hardware and receive an output - results of the hardware's operations. Software allows to use computers for numerous routine and sophisticated tasks such as text and image editing, solving math equations, controlling power plants, and so on. However, some software is created with malicious intentions: in the next subsection, we describe malware.

2.1.3 Malware

Computers are the pieces of complicated electronic *hardware* that are designed to perform numerous logical and mathematical operations faster and more efficiently than people. It is quite challenging for human to interact with hardware directly. Thereby, computer programs or *software* are designed to make human interaction with hardware simpler and more convenient. Most of the time software is written using one of the human-understandable programming languages such as C++, C#, Java, Python, and so on. By means of the programming language, a programmer introduces a certain *logic* into the software or program that is being created. The logic describes the behavior of software when it is launched. Each program is created to serve a certain purpose: to execute a certain sequence of operations and achieve certain results.

The functionality of the software can be differently assessed by people. When exposed to the results of the execution of the same piece of software, different people may perceive them as either *malicious* or *benign*. For example, when one launches a tool to wipe a hard drive with the intention to delete all data - the resulting "clean" hard drive is perceived as *wanted*, thus, benign result. On the opposite, if the software that wiped the hard drive was used to destroy important and sensitive data it is perceived as *unwanted* or malicious. Software that produces malicious and/or unwanted results is perceived as *malicious software* or *malware* and is such that contains a *malicious logic*[44]. In contrast, software that does not produce malicious or unwanted results is perceived as *benign* software or *goodware*.

Generally speaking, the software is perceived as malware if it performs certain

(unwanted by the user) actions without clearly notifying about them. Different methods are used to protect users and computers from malware. For example, restrictions on the types of files which can be downloaded from the Internet may significantly reduce the chance of malware appearing on the computer. However, such policies can make regular operations more cumbersome. Hence, specialized anti-malware software is nowadays used on most of the computers. Anti-malware, or *anti-virus*, software is created and used in order to detect the presence of malware in the system and limit the potential harm that it can cause. Anti-virus solutions use different approaches to detect malware such as signature-based or heuristics-based malware detection. The simpler and more straightforward one is *signature* based malware detection. It relies on signatures of previously known malware to find whether it is present in the system. Cryptographic hash functions are often used to generate signatures. They are used to map malicious files of arbitrary size to the unique fixed-size hash value or signature. For signature-based detection to work, a database of previously known malware has to be created. Such databases are essential parts of anti-virus vendors' businesses. A piece of software is registered as malware in such database if *community* agrees about its maliciousness. Here, by the community, we mean a collaboration between end-users and anti-virus vendors.

The main drawback of signature-based malware detection is its inability to detect previously unseen malware. Malware authors put significant effort into making new variants of malware, thus making old signatures useless. Signatures or hashes are generated based on the content of the entire malware sample and are designed in a way that two files', the contents of which are different by only one bit, will have completely different signatures. Different obfuscation techniques, such as polymorphism, metamorphism, or encryption are used to generate numerous variants of malware with similar functionality but different signatures[40]. To deal with this problem, anti-viruses incorporate *heuristics* malware detection. Heuristics malware analysis is aimed at finding parts of malware that remain relatively intact between the variants. For example, a newer variant of the same malware may use the same URL for communication, hence contain it in the file. Such an approach allows detecting of previously unseen variants of malware if they share at least some characteristics similar to previously known malware variants. To identify such characteristics and understand malware functionality one may perform malware analysis.

Malware analysis is the set of methods used to discover characteristics relevant for malware detection and classification. The simplest approach to find such characteristics rely on *static* malware analysis. Static characteristics or *features* are those that emerge directly from the file and can be extracted without a need to launch the potentially malicious executable. For example, such features as strings,

properties of Portable Executable (PE) header, names of sections, entropy, byte sequences frequencies, and so on are considered static features. Static features can be extracted from a file relatively easy and fast. However, the majority of them can be altered using obfuscation techniques. To overcome these limitations, dynamic malware analysis focuses on the feature produced during the execution of the malware. Moreover, malware becomes malicious only when it is executed[14]: an executable reveals its functionality when was launched. Thereby, *dynamic* malware analysis is considered as a workaround for the aforementioned problems. It is aimed at the discovery of the relevant features that emerge during the execution of malware.

Both static and dynamic features can also be used for malware categorization or *classification*. Malware classification is the task of attributing malware into different classes based on the similarities of their characteristics. Malware classification can serve different purposes such as improvement of pre- and post-attack actions[11], malware authorship attribution[6] and so on. For example, finding malware samples that make use of the same network address may point to the need of fine-tuning the settings of the firewall. Both malware detection and classification can be performed using dynamic characteristics[34]. As in this Thesis, we utilize the behavioral characteristics, in the next subsection we focus on dynamic malware analysis.

2.1.4 Dynamic Malware Analysis

Dynamic malware analysis is aimed at revealing and explaining the functionality of the malware[45]. The dynamic analysis allows to explain *what* malware is doing (which goals it is made to achieve) and *how* it achieves its goals. During the dynamic analysis, malware is launched in a controlled environment together with various monitoring tools. These tools allow tracking changes and actions in the system that occur during the execution of malware. For example, while analyzing a running malware it is common to track: file system, processes, memory, CPU, and network activity[45]. Thus, a task of dynamic analysis is to **describe changes** caused by malware in the system. Having such changes described, it is now possible to find how changes introduced by malware are different from those introduced by goodware. Thereby, dynamic analysis can be considered a source of features for malware detection.

Dynamic analysis can also be used to find behavioral features similar to the different malware samples. Such features may help to categorize malware based on its functionality. For example, an unknown variant or category of malware that achieves its goals similar to the known malware samples will be detected with features found using the dynamic malware analysis. In contrast to static features, dynamic features are more difficult to change. On a high level, the goals that

malware is made to achieve do not significantly change over time. Most of the malware samples are aimed at achieving one or several of the following goals: download and/or launch another executable, find and steal sensitive information, open a remote network connection, achieve persistence in the system, encrypt or destroy data. As operating systems do not significantly change over time, it means that malware will achieve similar goals with the use of similar mechanisms. For example, to achieve persistence on Windows systems, malware can use one of the Autostart Extension Points such as Microsoft\Windows\CurrentVersion\Run registry key. Thereby, tracking of certain activity patterns may help to detect potential malware. In other words, the **dynamic malware analysis involves interpreting the behavioral trace of malware**: a sequence of events recorded during the execution of malware. **Behavioral trace** represents events that occurred in the system as the result of malware's execution. For example, a behavioral trace may include but is not limited to: a list of file operations performed by malware, a list of launched processes and threads, and so on.

A process of the dynamic malware analysis can be described with the cycle shown in Figure 2.4. First of all, it is important to create a *baseline* - a base state of the guest OS. Once the analysis is done, it should be possible to restore the system after the malware launch. It is made to ensure similar conditions for the future runs of the same or different malware sample and avoid the influence of the changes in the environment on the results of the analysis. Once the OS is in its base state, we place a malware sample in the system. After that, we continue with pre-execution tasks. They include but are not limited to: launching monitoring tools, launch other software, etc. Later, we execute the malware sample. While it is running, the monitoring tools record the behavioral trace. After the execution, we perform post-execution tasks. They involve running analysis tools, dumping memory, gathering various information from the system, and so on. A cycle is repeated with each analysis to ensure consistency of the dynamic analysis results.

There are several challenges connected with dynamic malware analysis: analysis environment; type of events that has to be recorded; the amount of events that have to be recorded. First of all, one has to choose the analysis environment. It is necessary to decide in which *environment* analyzed malware sample has to be launched and which resources it needs to fully reveal its functionality. For security reasons, malware analysis is often performed in an isolated and controlled environment. It is highly not recommended to launch malware on live systems that contain sensitive data and have a connection to real networks. The most straightforward solution for malware analysis is to dedicate a separate computer where malware is launched and monitored. This solution provides a realistic environment for malware. However, it might be challenging and time-consuming to restore the system after the launch of malware. Thereby, malware analysts often utilize virtualization.

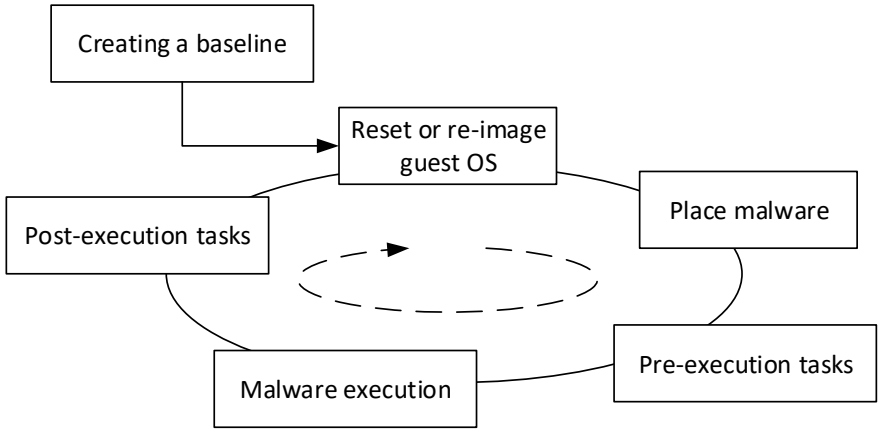


Figure 2.4: Dynamic malware analysis cycle. Inspired by [7]

With virtualization it is possible to launch a virtual machine on top of the host system. The VM is the virtualization or emulation of the computer system which has a full-fledged operating system installed. It allows having multiple virtual machines running in the same host system. Moreover, VMs allow launching malware in the different versions of OS, allowing for a more in-depth analysis of malware's capabilities under different conditions. Virtual machines provide the capability to make snapshots: preserved states of the same virtual machine. It is thereby relatively easy and fast to restore a VM to its baseline using a dedicated snapshot. However, running analysis in VMs has its drawbacks. For example, it is possible for the malware to use one of the anti-VM techniques to detect that it was launched in the VM and alter its behavior. To counteract such issues, analysts create virtual machines that "look" as similar to the real machine as possible. For example, emulating the network resources can make malware "believe" that it was launched in the real system. Other methods involve installing certain software, renaming and configuring virtual hardware, and so on. It is important to justify, that nowadays virtualized environments are more and more commonly used for normal operations. Thereby, anti-VM techniques are not that common in malware, since malware authors want their malware to be executed. Sometimes, malware requires additional resources or user interaction in order to fully reveal its functionality. Thus, it is the task of analysts to find out what kind of resources are missing (e.g. certain library) and which user input is needed (e.g. keyboard strokes, or certain mouse activity).

One of the crucial challenges is the amount and type of events that one is go-

ing to record and analyze. While performing dynamic analysis, it is important to decide, **how much data do we need to record**. Generally speaking, one has to decide how much time a certain sample will run during the analysis phase. The time can be restricted directly, for example, 1 or 5 minutes. Furthermore, the time can be restricted indirectly by setting the desired amount of certain events that have to be tracked. For example, one can wait until a malware sample encrypts the first hundred files. This is an open problem, as there is no correct answer due to the nature of certain malware samples which can wait in idle mode for extended periods of time. Thus, often it is the malware analyst who decides on the amount of time needed to record a behavioral trace.

However, the *types* of events that have to be recorded also pose issues for the dynamic analysis. As the task of dynamic analysis is to describe changes introduced by malware to the system, one has to decide on the level of data *granularity* used to create such a description. Before performing a dynamic analysis it is important to decide **how precise a description of malware behavior should be**. Thus, by granularity, we mean the number of details used to describe malware's functionality. A more detailed - low-level - description of the malware's behavior requires the use of a higher level of data granularity. In contrast, higher levels of the description require less data or lower level of data granularity. In Figure 2.5 we show an example of how the behavior of a malware sample can be described at different levels of granularity. For example, at the lowest granularity level, we can simply say that the malware sample is a Backdoor, and creates a remote connection. On the next level, we can provide a description of the used vulnerability and a particular exploitation technique implemented in the malware sample. At the highest level of data granularity in this example, we can describe function calls, network, and file activity caused by the execution of malware. We call the above-mentioned variants of behavior description *high-level*. High-level description requires recording of *high-level features*. High-level features include, but are not limited to: API and function calls, their arguments, file activity, network and registry activity, and so on. High-level features are relatively good for the detailed description of malware's functionality and behavior. However, in order to record such features, it is often necessary to have some sort of monitoring tools in the victim system which can be tracked and disabled by malware. Moreover, malware authors may try to hide malicious activity by utilizing various anti-dynamic analysis techniques such as anti-debug and anti-VM. Despite all the efforts, it is impossible to avoid execution on the system's hardware. Every action performed by any executable running in the system will result in *hardware activity*. The hardware activity is the source of *low-level* features - features that emerge directly from the system's hardware. Such features provide high granularity data about the malware's behaviour. In the next subsection, we describe hardware activity and

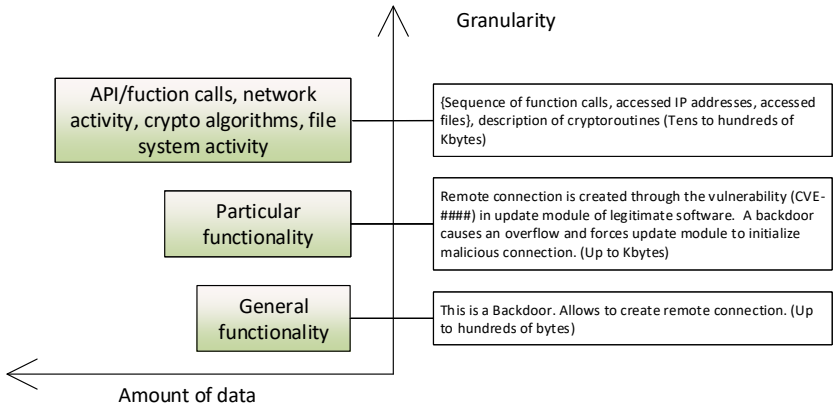


Figure 2.5: Description of behavior at the different granularity levels

low-level features in a more detailed manner.

2.1.5 Low-level behavior features

Execution of opcodes results in hardware activity: executed microoperations, opcodes themselves as well as memory operations emerge directly from the hardware. It means that with access to the hardware it is hypothetically possible to record such activities without a need to install monitoring tools on the OS. Information about executed opcodes and memory operations can be a rich source of information about the running process. However, there are more sources of information about the running process at the level of hardware. Modern CPUs often incorporate **hardware performance counters**: special counters that allow tracking of various events. For example, amounts of load, store, and branch retired instructions, misinterpreted branch instructions, Translation Lookaside Buffer accesses, cache access operations, Page Table Walks, and so on. The other sources of hardware activity can be a hard-disk activity, network card activity, GPU activity, and so on. In this Thesis, we focus on the analysis of the memory activity produced by malware.

There are several challenges linked to the study of low-level features in malware analysis. The first challenge is the difficulty of low-level feature extraction. Due to the nature of features, it is hypothetically possible to extract them directly from the hardware (in the form of electrical signals). However, due to the proprietary nature and simplified design of consumer-level computers, it is extremely challenging for regular researchers to perform such studies. One of

the workarounds is the emulation[39][36][28] of necessary hardware in, for example, Field-programmable gate array (FPGA). Such an approach allows, among the other benefits, to evaluate the potential computational overhead of malware detection caused by low-level detectors. However, such an approach requires additional competence from the researchers. Moreover, the phase of prototyping may require a significant amount of time since the architecture implemented in FPGA will likely be re-developed multiple times. Thereby, especially in the phase of prototyping, many researchers choose software-based methods of low-level feature extraction. Most of the common malware analysis and reverse engineering tools do not come with the functionality suitable for low-level feature extraction and recording. Thereby, quite often, researches[39][36][5] that use low-level features rely on dynamic binary instrumentation toolkits. Dynamic Binary Instrumentation (DBI) is the set of methods aimed at tracing and analysis of the behavior of binary executables during runtime on the level of opcodes. It is implemented through the injection of callbacks (instrumentation) and analysis instructions between the instructions of the original code. The newly generated code ensures transparency of the inserted instructions for the original code so that the functionality of inspected executable remains intact. DBI allows to inspect, record, and alter the state of the executable at every moment of its runtime. Thereby, DBIs are very powerful tools when it comes to the extraction of low-level behavioral traces. Most of the time, DBIs don't come with the desired ready-to-use functionality. Instead, they provide "basic" functionality such as: tracing of memory access operation, tracing of branch instruction, tracing of call instruction etc. Often, such DBIs as Intel Pin[4][31], Valgrind[32] or DynamoRio[20], come in the form of frameworks where users are given an API that allows for building of arbitrary tools. Thereby, DBIs provide enough flexibility for the instrumentation of executables and inspecting their low-level behavior. The results presented in this Thesis are based on the data recorded using the Intel Pin binary instrumentation framework.

2.1.6 Machine learning

Every day, thousands of new malware samples are detected. Thereby, malware analysis involves the processing and analysis of large quantities of data. Performing such processing manually will require infeasible amounts of time and human resources. Thus, **Machine Learning** methods are used to automate and speed up the processing and analysis of large quantities of data. Machine Learning methods are the computer algorithms that *learn* from data and improve their performance without being initially created to act this way. Machine Learning is often used for the *knowledge discovery*[29]. Knowledge discovery is the process of making new knowledge (information) from the data that would otherwise not be found. New knowledge can aid for a better understanding of data and therefore help to use it

more efficiently. Machine Learning involves *learning*: a process of improving the performance based on the input data.

Machine Learning (ML) methods can be roughly divided into *supervised* and *unsupervised*. Supervised methods include, but are not limited to, classification and regression, while unsupervised - clustering and associations. Supervised methods require input data to provide some sort of background knowledge and learning data. A supervised algorithm receives data, uses it to search a hypothesis space, and presents a final hypothesis as an output. Such a hypothesis presents a more efficient representation of the input data. For the supervised algorithm to work an input data should incorporate the description (attributes or features) of learning samples together with the supervising (target) variable. It is important to justify the term *feature* the way it is used the most in this Thesis. As provided in [29], a feature is a variable that can obtain a set of allowed values and represents a certain *property* of a learning sample. In this Thesis, we record long sequences (*traces*) of memory access operations performed by executables. Next, we split these sequences into n-grams. In our research we use these n-grams as features. The values of the features are either *1* or *0*. Respectively, these values reflect the fact of presence or absence of a certain n-gram in a certain trace. They reflect a *property* of a memory access traces to contain a certain n-gram. This way, the data that is used for training by the ML algorithm is essentially a matrix of zeros and ones. We refer to it as *bitmap of presence* in P1[15]. Basically, we are using *values* of the features to perform feature selection and to train ML models.

Target variable is unobservable in the real world and the task of the algorithm is to learn to predict it based on the attributes. For example, after processing behavioral traces of malicious and benign executables a Decision rules algorithm can hypothetically generate a set of rules that would help to distinguish between malicious and benign behavior. In this way, based on the behavior of a new executable it might be possible to predict whether it is malicious or not. Prediction of the type or *class* of the sample based on its features is called *classification*. It is important for the reader to distinguish between *classification* as ML task, and *malware classification*. In order to perform classification, the ML algorithm learns on the data with the target variable - class. As the result, a machine learning *model* is created as output. Such a model is considered to be *trained* for a certain task. The task of classifying a sample in one of the two classes is called two-class or binomial classification. In malware analysis, binomial classification is used for the task of malware detection. In this case, the ML model is trained to attribute samples into either benign or malicious classes. The task of classifying a sample in one of more than two classes is called multinomial classification. For example, malware classification involves the attribution of the malware in one of the many categories such as *malware families* or *malware types*. Thereby, multinomial classification is used

for the task of *malware classification*.

Unsupervised methods receive input data without the target variable. Their goal is to find relations and similarities among the learning samples. For example, a clustering algorithm can find similarities between behavioral traces of malware samples. This way, malware samples can be divided into subsets or *clusters*. Clustering may aid in malware categorization: attributing malware samples to groups based on their properties. With clustering it is possible to discover new categories of malware, thereby improving malware classification. Clustering can also be used to improve understanding of the relationship between existing malware categories. For example, it is possible to perform clustering on the data from malware samples where categories are previously known. Later, one may compare how previously known categories fit into newly created clusters.

ML model has to be tested in order to determine its quality and understand whether a certain type of features is suitable for the given task. The quality of the model is assessed based on its ability to correctly classify samples that were not used for training. In order to assess the quality of the model various quality measures (see Section 2.1.7) are used. There are two main methods for testing the ML model: k-fold cross-validation and percentage split. K-fold cross-validation is based on splitting the dataset into *k* chunks, out of which *k-1* are used for training and the remaining one is used for testing. The values of k are often set to be 5 or 10[11][14]. Training and testing are performed *k* times and the quality measures of the *k* ML models are averaged. Percentage split is based on splitting the dataset into two chunks. One is used for training and the other for testing. Often the ratio of a train to test set is 50/50, 60/40, or 70/30. One of the challenges in training the ML model is data preprocessing. Often, the raw data has to be cleaned or normalized. Moreover, the number of features of a chosen type can be too high which will lead to overfitting and models of lower quality. In order to decrease the number of features, reduce dimensionality, a set of methods called *feature selection* is used. Feature selection is aimed at selecting the most relevant for a given task features. Feature selection is normally done before training the ML model. The problem is that when feature selection is performed on the full dataset, it takes into account the properties of samples that are later used for testing. Thus, incorporating "the knowledge from the future" into the model making its quality assessment doubtful. We do not necessarily say, that performing the feature selection on the full dataset is erroneous. This helps to understand the capabilities of a combination of certain features and the ML method. However, as the amount of newly discovered malware is huge, in malware analysis it is especially important to test ML models against *previously unseen* samples: samples that did not contribute to the model training and feature selection. In this Thesis, we study how low-level features can be used to detect previously unseen malware.

In this thesis we utilize Machine Learning for binomial and multinomial classification. While using different ML methods to train models for different classification tasks it is important to assess the quality of trained models. This helps to understand which methods and features are better suited for certain tasks. In the next subsection, we discuss methods for assessing the quality of machine learning methods.

2.1.7 Assessing the quality of ML aided malware detection

Classification ML models are created or trained on the labeled data. Data that consists of information about the class of learning or train samples. The data used for training is called *training* dataset. To test the performance of the trained ML model a *test* dataset is used. It normally contains samples that were not used for training. In order to assess the performance of the model various *measures* are used for the assessment of the quality of ML models. Such measures are normally a numerical representation of the certain quality of the machine learning model. The numerical nature of the measures allows for a comparison of the quality of the different machine learning models. We consider the machine learning models different unless they were created by training the same ML algorithm on the same data with the same parameters. In this Thesis, we utilize ML for classification purposes. The quality measure of an ML model often represents the ability of the ML model to correctly classify samples from the test dataset. Thereby, in this subsection, we focus on several quality measures that are especially relevant for malware detection and classification. Before describing the measures we have to provide several basic values that are used to calculate the measure. These values are normally used for binomial classification, where one class is considered as True class while the other is False class:

- True Positive (TP) - the number of samples of the True class from the test set that were correctly classified as True class.
- True Negative (TN) - the number of samples of the Negative class that were correctly classified as Negative class.
- False Positive (FP) - the number of samples of the Negative class that were incorrectly classified as True class.
- False Negative (FN) - the number of samples of the True class that were incorrectly classified as Negative class.

These values can not be solely used for assessing the quality of ML models: the absolute numbers of correctly or incorrectly classified samples can be barely used for the comparison of different ML models e.g. due to different amounts of samples

in test datasets. Thereby, the following *measures* are used to provide normalized values suitable for comparison of the different ML models. The first one is True Positive Rate (TPR) is calculated as $TPR = \frac{TP}{TP+FN}$ and can take values from 0 to 1. TPR is sometimes called sensitivity or recall[29] and represents the ability of the model to correctly classify the sample of the True class. In malware analysis, high TPR means the good ability of the model to detect malware. In other words, the model with $TPR = 1$ is able to correctly classify all malware samples from the test set in thereby considered as the model with a good detection rate. The second quality measure is False Positive Rate (FPR) is calculated as $FPR = \frac{FP}{FP+TN}$ and can take values from 0 to 1. FPR reflects the tendency of the model to incorrectly classify samples from the Negative class as samples from the Positive class. For malware detection, a model with high FPR will generate a high amount of false alarms by labeling benign executables as malicious. ML model trained for malware detection task is considered good if it has high TPR and low FPR. The next quality measure is accuracy (ACC) and is calculated as $ACC = \frac{TP+TN}{TP+FP+TN+TF}$. It takes values from 0 to 1 and represents the ability of the model to correctly classify samples from both classes. This measure can also be used for multinomial classification by representing the fraction of correctly classified samples from all classes to the overall number of samples. There is one problem linked to the use of the aforementioned measures. They might not always correctly represent the quality of the models trained on *imbalanced* dataset. The imbalanced dataset contains different amounts of samples in different classes. For example, a dataset may contain 1000 malware and 10 benign samples. Thereby, ML model trained on such dataset might have difficulties to correctly predict a rare class. Furthermore, testing on the imbalanced dataset may lead to an erroneous assessment of the model's quality. For example, if the test set contains 10 times more samples of one class than of another, a model that will classify all samples from the test set to the majority class will achieve an accuracy of 0.9. Thereby, when providing such a measure it is important to provide the description of train and test datasets. However, there are several ways of dealing with imbalanced datasets. First of all, it is possible to apply sample weighting[29]. In this case, samples of a rare class receive higher weights, while those of a majority class - lower. In such a scenario, the correct classification of a sample from the rare class will contribute to the quality measure more than the correct classification of the majority class. Second of all, it is possible to use specialized measures that are believed to overcome the problem of imbalanced datasets. One such measure is *F1-measure* (F1M). It is calculated as $F1M = 2 \times \frac{PPV \cdot TPR}{PPV+TPR}$. Here $PPV = \frac{TP}{TP+FP}$ is a positive predictive value or *precision* that represents the proportion of correct positive classifications among the samples classified to the positive class. F1M is the harmonic mean of precision and recall. It is believed to represent models' accuracy while eliminating the

influence of TN, thereby contributing to better model assessment in the case of an imbalanced dataset.

In order to properly use these measures it is important to understand several basic principles. First of all, most of the quality measures have their limitation. And it is thereby important to take them into account while assessing the model. Next, while comparing two models it is important to understand *what* could lead to the different performance of the models. For example, models created with the same ML algorithm (e.g. Random Forest) can have different performances due to different parameters used for training (e.g. number of trees in Random Forest). Moreover, models trained or tested on different datasets may show different performances due to the nature of the data. It is generally considered that the more data is used for training the model - the better the performance of the model will be achieved. For example, a model trained on the dataset that has 100 malicious and 100 benign samples can potentially have lower performance than one trained on the dataset with 1000 malicious and 1000 benign samples. Furthermore, the same samples can be described using different features. Thus, models can have different performances because one type of features represents a target class better than the other. Thereby, quality measures should not be used without a proper assessment of their limitation.

2.1.8 Feature selection

In the previous subsection, we briefly mentioned the influence of the features on the ML model's performance. As we already mentioned, features carry information about certain properties of the samples. For example, the frequency of appearance of a certain sequence of opcodes in the behavioral trace of an executable can be potentially used to determine its maliciousness. The problem arises from the fact that the number of potential features or *feature space* can be large. For example, as described in article P2 [11] from Chapter 7, the amount of unique memory access patterns produced by the malware samples was more than 15 millions. Such numbers of features may lead to various problems for training, using, and assessing the ML models. First of all, large amounts of features require more time to process and thereby train, assess and use the ML models. Second, a large number of features or *high dimensionality* of the data may lead to the model *overfitting* the learning samples. For example, an ML algorithm trained on high-dimensional data can create precise hypotheses that perfectly describe learning samples. However, test samples may have slight variations of the feature values. Therefore, the overfitting model may not be able to correctly classify test samples and have lower classification performance. Lastly, even if the model trained on high-dimensional data shows decent performance it might be challenging to *interpret* the results of the classification. Interpreting or understanding the results of

classification is important in the various fields of ML applications. For example, it might be crucial to understand why a certain malware sample was not detected by the model. In order to interpret such an outcome, one may choose to analyse features and their influence on the classification. Thereby, a high-dimensional dataset can make such interpreting infeasible. Thus, in this subsection, we discuss Feature selection: a set of statistical methods used for the reduction of feature space.

As features describe certain properties of samples, a reduction of their number will likely lead to the loss of useful information. Thereby, the reduction of feature space may lead to the inability of ML methods to generalize over the data, hence making the quality of potential ML models lower. In contrast, some of the features may be irrelevant for the given task. Therefore, the process of feature selection is aimed at selecting features that are the most relevant for a certain task. For instance, the task of classification requires features that allow distinguishing samples of different classes based on the values of those features. Various algorithms can be used in order to select the most relevant features such as Information Gain, Distance measure, Gini Index, Relief[29], Correlation-based Feature selection (CFS)[23] and so on. Most of the feature selection algorithms utilize some sort of *feature quality measure*: a measure that helps to evaluate and rank the utility of the features for the given learning problem. Such methods are called filter methods.

In the algorithms such as Information Gain or Distance measure feature quality measures are calculated based on the *information content*: the amount of information [43] about the target value (class in the classification problem) that certain feature carries. Such methods are called myopic [29]. They do not take into account locality and the relationship between the features. Many of the feature selection methods look only at the relation between the feature and a target variable. During the feature selection process, such methods rank features based on their quality measure. For example, in such methods, quality measures represent the ability of a feature to describe a target variable. Later, a certain number of features are presented as the result of the selection process. In such cases, the best number of features to choose is normally decided on through a set of experiments. Different numbers of features are selected and then used for the training of ML models. Trained ML models are compared based on their performance measures and the best one is normally considered to be trained using the best set of features.

Some feature selection methods do not use any feature quality measures: wrapper methods. They use a certain ML algorithm to train a model using a certain subset of features. Basically, various combinations of features are used to train models. These models are later compared based on their performance measures, and the best one is considered to be trained with the best feature subset. Such an approach allows finding the best feature set for a given ML algorithm. How-

ever, the number of possible feature combinations of various sizes may be large. Thereby, wrapper methods are more computationally intense than filter methods. Thus, in the case of large initial amounts of features, wrapper methods have limited usability.

As we already mentioned, some feature selection methods rely solely on the information about the target variable that a certain feature carries. These methods tend to select features with the highest information value. However, features that correlate the most with the target variable may also correlate with each other. Thereby, the best features may not complement to each other. They will carry redundant information which won't improve the overall quality of the feature set consequently decreasing the ability of the ML model to perform classification. Thus, methods that aimed on overcoming this problem exist. For example, ReliefF ranks features based on *locality*: their ability to distinguish between learning samples that lay close in the feature space. A feature that helps to distinguish between close samples of the different classes gets its score increased. In its turn, a feature that differentiates close samples of the same class gets its score decreased. This way, ReliefF ranks features in the context of other features. However, it can still select features that carry redundant information[47]. To overcome the problem of redundancy a feature selection method should take into account dependency or *correlation* between the features. Correlation-based feature selection method[23] (CFS) was designed to deal with correlating features. It is important to note, that CFS was originally designed to select a subset of features of user-defined size. However, the current implementation in Weka[24] is made to gradually add features to the feature sets until further improvement of the quality of the best feature set is not possible. CFS calculates the correlation between all features (internal correlation) and between features and a target variable (external correlation). It later ranks subsets of features based on their average internal and external correlation. Basically, a rank of a good feature subset has maximized the trade-off between internal and external correlations. The more feature correlates with the target variable the higher the rank of a subset, whilst the more features correlate between each other the lower the rank of the subset. Even though CFS has great capabilities to reduce feature space in several orders of magnitude [11][14][13] (P2,P4,P6) it comes with its limitations. First of all, a need to compare *all* possible subsets of a size *k* from a full set of size *n* can result in high computational costs as the number of combinations will be $\frac{n!}{k!(n-k)!}$. High *ns* can make search very computationally complex. Moreover, the large size of the full feature set will result in a need to calculate and store the values of n^2 correlations. With feature numbers that can reach millions, such a task will require a lot of computational and memory resources. Therefore, when choosing CFS one has to take into consideration its limitations.

As we said in the beginning, the reduction of feature set size reduces the amount of information about the target variable that ML algorithm can use for learning. Thereby, it sometimes happens (especially with myopic methods) that a bigger feature set allows for achieving a better performance of the ML model. However, lower number of features makes the analysis of the ML model performance simpler[11][12][14] (P2,P3,P4). Furthermore, a lower number of features improves the interpretability of the model and helps in the explanation of found phenomena. Thereby, especially for research purposes, it might be better to use a smaller feature set even if ML models trained with it underperforms compared to the bigger feature sets.

2.2 Related works

In this section, we provide an overview of the works that elaborate on the topic of application of low-level features for malware analysis. Authors of [5] were one of the first who proposed malware detection using DBI. In their work, they used Intel Pin to record behavioral traces that contain: calls to and arguments of "exec" function, system or library calls that involve any file system modification, calls to functions that create hard and symbolic links as well as instructions that perform memory read and write operations. They split the execution trace based on the basic blocks and store the aforementioned information for each executed basic block. This data was later used to generate regular expressions which, on their turn, were used to create policies necessary for malware detection. To evaluate their approach they used "original" and intentionally obfuscated malware. After a set of experiments on Windows and Linux OSes authors concluded, that their approach allows to achieve 100% detection rate and low FN and FP rates. Even though authors of [5] did not focus solely on low-level features, their experiments showed that the application of DBI has potential in malware analysis.

The next paper where authors used DBI and low-level features for malware detection is [28]. There authors used Intel Pin to track low-level behavior of executables. They utilized the following features: frequency and presence of opcode in each of the Intel x86 architecture categories; memory reference distances; total number of memory reads or writes; total number of unaligned memory accesses; total number of immediate and taken branches. After a set of experiments they showed, that their approach allows to achieve detection accuracy of more than 96% for offline and 92% for online detection. Moreover, using FPGA they implemented the proposed detector in hardware and evaluated its overhead. They showed, that such detector causes up to 9.83% slowdown. They also made an important notice, that collection of memory related behavioral events caused most of the slowdown: when they did not collect memory related events the slowdown was under 2%. The topic of hardware implementation of malware detectors were later expanded

in [35]. The authors proposed Malware Aware Processors (MAP). MAP is a "processor augmented with a hardware-based online detector"[35]. They trained their classifiers using low-level behavioral events recorded by Intel Pin. Authors recorded features similar to the previously mentioned features from [28]. They showed, that it is possible to have relatively low overhead and achieve good detection performance at the same time.

Other authors suggest utilization of other low-level features for malware detection and classification. For example, authors of [9] use hardware performance counters to track branch, store and load instructions being retired and mispredicted branch instructions. With their approach authors achieved detection rate of more than 92% and accuracy of more than 96%. On their turn, authors of [17] suggested the use of opcode sequences as a source of features for the prediction of maliciousness of executable. They performed analysis of opcode frequencies within behavioral traces of malicious and benign executables. Author of [17] concluded, that difference in opcode frequencies between malware and goodware has a potential to be used for malware detection. As it was previously mentioned, often opcodes use CPU registers as input arguments. Thereby, authors of [30] proposed a malware detection approach that uses general-purpose registers as a source of information. They utilize spatial and temporal properties of eight registers and compare their malware detection potential to other low-level features such as opcode existence and frequency of opcodes. With such an approach they managed to achieve an accuracy of more than 97%.

Papers that we mentioned so far either did not use low-level memory activity or used only amounts or facts of presence of memory access operations in the combination with other features. As to the author's knowledge, the first paper where memory accesses were solely used for malware detection was P1 [15] which is presented in this Thesis in Chapter 3. Together with other papers of Banin et.al. it presents a thorough evaluation of memory access patterns capabilities in malware detection, classification, and analysis. In these papers authors utilized sequences of memory read and write operations as features and achieved detection accuracy of more than 99%[14] (P4). Other authors explored memory activity in a set of different ways. For example, authors of [49] used memory access histograms and achieved detection rate of more than 99%. Authors of [51] extended a technique proposed in P1 [15]. They used memory address, address of executed operation, its number in the sequence and the type of memory access operation. They successfully used these features to find similarities between various malware samples.

Chapter 3

Summary of published articles

The main part of the research resulted in 6 research papers:

P1 S. Banin, A. Shalaginov, and K. Franke, "Memory access patterns for malware detection" Norsk informasjons sikkerhets konferanse (NISK), pp. 96-107, 2016

P2 S. Banin and G. O. Dyrkolbotn, "Multinomial malware classification via low-level features" Digital Investigation, vol. 26, pp. S107-S117, 2018

P3 S. Banin and G. O. Dyrkolbotn, "Correlating high-and low-level features," in International Workshop on Security, pp. 149-167, Springer, 2019

P4 S. Banin and G. O. Dyrkolbotn, "Detection of running malware before it becomes malicious" in International Workshop on Security, pp. 57-73, Springer, 2020

P5 S. Banin, Malware Analysis using Artificial Intelligence and Deep Learning: "Fast and straightforward feature selection method: A case of high dimensional low sample size dataset in malware analysis." Springer, 2020

P6 S. Banin and G. O. Dyrkolbotn, "Detection of previously unseen malware using memory access patterns recorded before the entry point," The 4th International Workshop on Big Data Analytic for Cyber Crime Investigation and Prevention, 2020

The author of this Thesis was also involved in the writing of the following paper:

S1 A. Shalaginov, S. Banin, A. Dehghantanha, and K. Franke, "Machine learning aided static malware analysis: A survey and tutorial" in Cyber Threat Intelligence, pp. 7-45, Springer, 2018

Research of the question RQ1 did not directly result in a publication of a separate article. However, this research question was addressed in Section 2.2. Moreover, it was necessary to have a good understanding of the best practices to put the overview of the related works into papers **P1-P6**.

3.1 General overview of the used datasets

Before giving condensed descriptions of each of the papers we want to provide a reader with a short overview of the datasets used in papers **P1-P6**. The following list contains descriptions of the datasets, whilst more detailed descriptions are present in the respective articles. In total, we have used three different datasets **D1-D3**.

D1: Used in **P1**. In this paper, we use malicious PE32 executables found on VirusShare [48], and benign PE32 executables found on clean installations of Windows XP, 7, 8 and 10. We chose samples of the smallest file size that contain GUI. We have not performed any other filtering. As in this Thesis, we utilized dynamic malware analysis, the data used for training the ML models obviously came only from samples that managed to start on our VM. The main limitation of this dataset is that benign executables were created by the same company which could have affected the validity of the results. However, as we have later shown in P4 our approach works similarly well on benign samples from different companies.

D2: Created for the use in **P2** and reused in P3. The dataset used in these papers is a part of a larger dataset[42] created under the initiative of the Testimon [22] research group. Malicious executables were collected from various available collections available online. As we focused on multinomial malware classification in P2 we chose files that belonged to the ten most common families and ten most common types of that dataset. We also chose the executables that have GUI and do not contain AntiDebug and AntiVM techniques. Similarly to P1 we focused on the smallest files and did not perform any other filtering. The main limitations of this dataset might be the number of samples in the categories as well as an unintentional selection of the samples that are not representative of their category.

D3: Created for the use in **P4**, reused in P5 and P6. Malicious executables were found in the *VirusShare_00360* collection. We chose only executables that were labeled as malicious by 20 or more AV engines on VirusTotal. The only other filtering that was done is choosing only samples that belong to the ten most common malware families present in the dataset. Benign executables were taken from the *PortableApps*: an online resource of free and

open source software. We chose only samples that were not labeled as malicious by any of the VT engines present in VirusTotal at that moment. The main limitation of this dataset is the number of samples in categories. As it is hard to avoid the potential unintended selection of samples that are not representative of their categories.

During our research, we did not take into consideration the presence or absence of various packing and obfuscation techniques in both benign and malicious executables.

In order to record memory access traces, executables from D1,D2, and D3 were launched from the same folder on the virtual machine together with a specially crafted IntelPin-based tool. During the research presented in P1, all samples were transferred to the VM before the baseline snapshot was made. This was done because VirtualBox API at that time had bugs that made automated file transfer impossible. Samples from dataset D2 were transferred separately before their launch together with IntelPin tool. This became possible because VirtualBox API has been fixed by the developers. Samples from D3 were also transferred to the folder on the VM individually. There is a difference in the use of benign samples between D3 and D1. Benign samples from D1 were copied by themselves without their external resources if such were present. This partially lead to the fact that most of the selected benign executables didn't start. To eliminate this problem, we decided to transfer executables from D3 together with their external resources. One may assume, that benign executables accessing resources in its folder may generate specific activity and make them more distinguishable from malware. On the other hand, presence of these resources makes behaviour of benign executables more realistic. As the absence of necessary files may lead to similar exceptions being thrown by executables. Which again may make benign executables stand out from malicious ones.

Any research dependent on real-life samples is limited by the quality of the dataset. The results present in this Thesis may have lesser validity, as any used dataset can be criticized for not being representative if compared to the real-life scenarios. Therefore, in future work, it is worth trying to construct a larger and more thoroughly selected dataset.

3.2 Memory access patterns

Our first paper was aimed at answering the RQ2:

- How can low-level features be used for malware detection?

After a literature review we found that, despite several papers mentioning usage of memory activity for malware detection, no one has used *memory access patterns*

for malware detection and analysis.

In the paper **P1** we showed, that it is possible to distinguish between malware and goodware using memory access patterns. In that paper, we explored basic principles of malware detection using memory access patterns. We proposed to record a sequence of all memory access operations from the launch of the process and split it into a set of overlapping n-grams (patterns). These patterns were later used as features for the training of ML models. In that work, we studied how long the sequence of memory access operations needs to be in order to provide enough information for the classification. We also explored how the size of the pattern influences classification performance. In that paper, we found, that it is enough to record 1M of memory access operations. We also found that such sequence has to be split into the set of n-grams of the size 96. In the end, we concluded, that 800 of the *best* 96-grams selected from the sequences of the first 1M of memory access operations produced by all samples in the dataset allow us to achieve binary classification performance of more than 98%. In that paper, we also face a problem of high dimensionality: the number of unique 96-grams was too high for common ML packages. Thus, we had to implement a novel feature selection method. This feature selection method is later explored in more detail in the paper **P5**.

3.3 Malware classification

As paper **P1** showed that it is possible to detect malware using memory access patterns we focused on answering the **RQ3**:

- How can low-level features be used for malware classification?

In our next paper **P2** we studied how memory access patterns can be used for multinomial malware classification. In that paper, we used an already known combination of 1M of first memory access operations and n-grams of the size 96. We trained ML models on two different datasets. The first one consisted of 10 malware types, while the second of 10 malware families. First of all, in this paper, we had to give a definition of malware types and families as there were no consistent definitions in the literature. Second, we found that it is easier to distinguish between malware families than between malware types. This finding complied with the proposed definitions of families and types. We achieved 84% classification accuracy for 10 malware families and 68% for 10 malware types. In this paper, we first utilized the two-step feature selection method. On the first step, we select 50K,30K,15K,10K, and 5K of the best features with Information Gain feature selection. In the second step, we select the best feature subset from the 5K IG selected features with Correlation-based Feature Selection (CFS) from Weka. Such an approach allowed us to reduce feature space from millions of features to just 29. In the end, we performed an analysis of the classification performance.

One of the valuable contributions of that paper was an analysis of subcategories. For example, we looked at how samples that belong to a certain malware type are labeled with malware families labels and vice versa. This allowed us to explain why certain malware categories were more likely to be misclassified than others.

3.4 Correlating high- and low-level features

During the work on papers P1 and P2 we found, that memory access patterns are not human-readable: it is hardly possible for a human analyst to understand what exactly an executable was doing when produced a certain memory access pattern. Thus, we focused on answering the **RQ4a**:

- How can low-level features improve understanding of malware detection and analysis?

Our third paper **P3** was dedicated to the attempt to correlate low-level features (memory access patterns) and high-level features (API-calls). We used the best memory access patterns and best API calls n-grams selected similarly to P2. Under our experimental design, we were not able to find any significant correlation. However, we showed, that multinomial malware classification can be improved, if API calls and memory access patterns are combined. Thereby, in **P3** we also answered the **RQ4c**:

- To what extent low-level features can improve malware classification accuracy?

This showed that memory access patterns and API calls we used under our experimental design did not correlate. Hence, did not bring the redundant information that allowed to improve the classification performance. It is important to note, that we did not use arguments passed to API functions in our research. The reason for that was it was technically challenging way to locate and decode arguments passed to the functions during dynamic analysis. But this is definitely worth looking into in the future work. While analyzing the results of that paper we found, that most of the memory access operations that we recorded originated from before the Entry Point part of process execution. Basically, we discovered that we can potentially detect and classify malware based on the activity produced by the process before it has any chance to cause any harm to the victim system.

3.5 Improved malware detection before the Entry Point

In our next paper **P4** we in detail studied the possibility of malware detection and classification based on the activity that the process produces before the first instruction from the main module of an executable is executed. With **P4** we answered **RQ4b**:

- How can low-level features improve malware detection capabilities?

In that paper we proposed a novel approach in dynamic malware analysis: the BEP-AEP approach. With this approach we separated behavioral activity produced by a process *before* it has reached an Entry Point and *after*. The Entry Point drew our attention after analysing the results obtained in **P3**. We have chosen Entry Point as a separation point due to its crucial meaning in the execution of a newly created process. It separates memory activity generated during finalization of process creation and memory activity generated by the logic that was put into an executable by its creator. A more detailed description of the process creation flow is provided in the paper **P4**. Thereby findings from **P3** naturally drew our attention. During our research, we did not consider studying the applicability of any other possible milestones in process execution flow as separation points. In paper P4 we found, that under our experimental design it is possible to detect malware based on only the BEP memory access patterns with an accuracy of more than 99%. We also found that a similar amount of memory access operations produce AEP allows to derive enough information to detect malware with an accuracy of more than 99%. We also discovered that it is easier to distinguish between malware families using the AEP memory access patterns than BEP. In that paper, we also found, that most of the memory access patterns selected as best features originated from *RtlAllocateHeap* routine of the *ntdll.dll* Windows library. Based on these findings one may assume, that *what* and *how* is being allocated in the memory upon startup of the process may depend on certain static properties of an executable. In order to address this assumption one may need a better understanding of process creation flow and specifically the RtlAllocateHeap function. Unfortunately non of the sources available at the time of research provided a detailed description of the aforementioned things. Without this knowledge, such study will be as limited as the search for correlation between API calls and memory access patterns described in **P3**. We, therefore, leave this part of the study for future work.

3.6 Intersection Subtraction feature selection

In our paper **P5** we studied how well the feature selection method proposed in P1 performs if compared to the more common IG feature selection method. Our feature selection method is called Intersection Subtraction (IS) feature selection method. The basic principle of the IS feature selection is the following.

1. First we decide on the desired amount of features m that we want to be selected by the IS feature selection algorithm.

2. From the memory access trace of each sample in the dataset we construct a set of unique memory access n-grams. Every n-gram found within a trace is therefore recorded only once in this set.

3. For each of the classes (benign and malicious) we construct a vector of n-grams which contain every unique n-gram found in all traces of all samples of a particular class. Therefore, here every n-gram is also present only once.

4. For each of the vectors we calculate class-wise frequencies of the n-grams. Basically, if an n-gram is found in 50% of the traces within a class its class-wise frequency will be *0.5*. For example, vectors of 3-grams can have the following class-wise frequencies:
$benign\_3gram\_vector$=[[WWR,1.0], [WRW,0.87], [RRR,0.66],...]
$malicious\_3gram\_vector$=[[WWR,1.0], [RWR,0.92], [RRW,0.57],...]

5. With two vectors constructed from traces of benign and malicious samples we remove those n-grams that are found in both vectors regardless of their class-wise frequencies. We obtain two *clean* vectors by *subtracting* their *intersection* from them. For example, the above-mentioned vectors contain one 3-gram in their intersection. After removing it the vectors look like this:
$benign\_3gram\_vector\_clean$=[[WRW,0.87], [RRR,0.66],...]
$malicious\_3gram\_vector\_clean$=[[RWR,0.92], [RRW,0.57],...]

6. From both of the clean vectors we select *m/2* n-grams with the high class-wise frequencies. We use them to construct an n-grams vector of the length *m*. Thereby the final feature set contains equal amount of n-grams unique to each of the classes.

7. This vector becomes our feature set that is used in the training of the ML model. It is important to note, that class-wise frequencies are used only during the feature selection process and not used for training the ML model. ML algorithm receives a "table" where rows represent samples, columns represent n-grams from our newly constructed feature set and cells contain values *1* or *0* depending on the fact of presence of a particular n-gram in the particular sample.

In P5 we compare custom implementations of IS and IG methods by using them to select features suitable for malware detection. First of all, we showed, that IS worked at least 3.8 times faster than IG. Second, we showed, that features selected by IG allow to train ML models that achieve almost the same classification performance as those trained using features selected by IG. We also found that feature sets selected by IS and IG we almost completely different. This provides a valuable finding in the field of machine learning and might serve as a stepping stone for future research in the area of feature selection. In the end, we concluded, that even though IS has its own disadvantages it can help researchers to use a faster feature selection method on high-dimensional datasets in order to see whether a certain classification problem can be solved with a certain type of features at all.

3.7 Detection of previously unseen malware

In our last paper **P6** we continued to work on **RQ4b** and explored how the BEP memory access patterns can be used to detect *previously unseen* malware. The crucial difference between that paper and paper P4 is the following. We arrange our dataset based on VirusTotal's first submission dates of the samples. We then split the original dataset into train and several test sets. In P4 we used cross-validation in order to assess the quality of ML models *after* the feature selection was done. In contrast, here samples from test sets were not used for feature selection and were newer than those used for training. In that work we found, that models trained on BEP memory access patterns are capable of detecting previously unseen malware with high detection rates. However, most of the models developed high FPR as test sets became more distant in time from the train set. One of the valuable contributions of P6 is the thorough analysis of the classification performance. We explored, whether the following characteristics could explain the performance of the models: distribution of malware families within the train and test sets, amount of selected features, the novelty of features, the way features represent classes, and feature space. Even though such an analysis approach did not help us to explain all classification performance questions, we believe that it may help other researchers to address their problems.

3.8 Survey paper on static analysis techniques

The paper **S1** presents an overview of the static malware analysis techniques and their comparison on two large malware datasets. In that work, we explored how different static characteristics of malware can be used for malware detection. There, we used the following static features: PE header, bytes n-grams, opcodes n-grams and API calls n-grams. In that paper, we used two sets of malware with roughly 41000 and 58000 samples in them and one set of benign executables with roughly 16000 samples. The results of that work showed, that Decision Trees and k-Nearest Neighbors algorithms perform better than the other methods.

It is important to outline, that the contribution of the author of this Thesis to the paper S1 consists of writing the description of ML and feature selection methods; conducting the experiments with byte and opcode n-grams. The S1 paper does not contribute to the main research topic of this Thesis. However, it provides a valuable example of a machine learning application in malware research.

Chapter 4

Contributions

This thesis contributes towards an improved understanding of the applicability of low-level features for malware analysis. The following contributions are present in this research:

- **Malware detection**: Contribution towards utilizing the memory access patterns for malware detection[15] P1 . We have shown, how much low-level data has to be collected to provide a decent detection rate. We also studied, how collected low-level data has to be preprocessed. This contribution shows that it is possible to perform high-accuracy malware detection based on memory access patterns.

- **Novel and improved detection capabilities:** Contribution towards utilizing memory access patterns to detect malware on launch before the execution reaches the main module[14] (P4). We have shown, that memory access patterns allow to detect launched malware before the Entry Point. We also contributed towards the understanding of how the performance of classifier trained on older samples changes when given newer samples to classify[13] (P6). Under our experimental design, we show, that it is possible to detect previously unseen malware based on the BEP low-level activity for at least 11 months since the update of the model. We have also discovered, that combining high- and low-level features improves classification performance.

- **Feature selection**: Contribution towards faster feature selection method for high-dimensional datasets[10] (P5). We have created and tested a novel feature selection method: Intersection Subtraction feature selection. We have shown, that it works faster than Information Gain. Also we have shown, that under our experimental conditions IS performs similarly to the IG. Thereby, we increased the speed of feature selection while maintaining its quality.

- **Low-level features decoding**: Contribution towards methods for finding the high-level counterparts of low-level activity patterns[12] (P3). We have created a method to find which high-level events are related to a certain low-level activity pattern. During our study, we discovered, that many of the *relevant* memory access patterns originate from the same high-level API call.

- **Better possibilities for threat analysis: Novel malware classification capabilities** Contribution towards better threat analysis using memory access patterns to classify malware into malware families and malware types[11] [14][12] (P2,P3,P4). We have shown, that it is possible to classify malware into families and types using low-level features.

- **Better understanding of the phenomena:** Contribution towards methods for analysis of ML classification performance and feature selection results. We showed how one can analyse the results and performance of ML-based malware detection. We believe that our approach to the analysis of classification and feature selection results can help other researchers and contribute to a more concise use of ML in malware-related research.

4.1 Malware detection

When an executable is launched in the system, it produces a set of behavioral characteristics that can be used to identify the executable and its category. Some of these characteristics are suitable for distinguishing between malicious and benign executables. During our research, we have several times shown, that it is possible to detect malware using only memory access patterns. This is a valuable achievement since malware executing on modern computers always invoke the inevitable activity in the virtual memory. In our first work, it has been shown, that the first 1M of memory access operations produced by an executable gives enough information to distinguish between malware and goodware[15] P1. We have also found, that the first 100K of memory access operations is not enough to achieve satisfactory detection accuracy. In that paper we found, that splitting a sequence of memory accesses into n-grams of a size 96 allows to **detect malware with an accuracy of more than 98%**.

4.2 Improved detection capabilities

While analyzing the data we recorded for the study presented in P3 [12] we found, that most of the recorded memory access patterns emerged before a newly launched process begins to execute commands from its main module: before the

Entry Point (BEP). We used this finding to conduct a set of more detailed experiments in P4 [14]. There we found, that using only the memory access patterns produced BEP allows to achieve a detection accuracy of more than 99%. **This finding allows to stop malware upon startup: before it has a chance to perform any malicious actions**. We have also explored malware detection capabilities of memory access patterns produced after the Entry Point (AEP). Counter-intuitively, we found, that on our dataset memory access patterns produced BEP allows to achieve slightly higher detection accuracy when compared to AEP. However, this can be a result of certain properties of our dataset. We have also combined memory access patterns produced BEP and AEP, which allowed for slightly higher detection accuracy. Moreover, during our research, we found that **one may need only 9 memory access patterns** to train the ML model capable of malware detection with an accuracy of 99.7%.

When exploring the capabilities of a novel malware detection approach, it is important to test it against "previously unseen" malware. As thousands of new malware samples are discovered every day, it is important to understand the robustness of trained models against samples that did not contribute to the model *and* features. In our paper, P6 [13] we shown, that models trained on older data degrade over time. Detection accuracy drops, as test samples become more distant in time from train samples. However, even though the accuracy drops, TPR remains quite stable and high. In other words, we show, that the ML model trained on the samples from just the two first months **can detect most of the unseen malware samples for at least 11 following months**.

4.3 Feature selection

At the beginning of this research, we faced an unexpected problem. The number of unique features was very high making the use of commonly available ML packages impossible. This forced the author to implement a fast, straightforward but yet efficient feature selection method. In paper P5 [10] we describe an Intersection feature selection method (IS) that was first used in P1 [15]. It has been shown, that this method allows to process millions of features **faster than commonly used Information Gain (IG)**. In our experiments, we show, that models trained on features selected by IS perform slightly worse than those trained on features selected by IG. However, this difference is very small. Thus, we have contributed towards faster feature selection which can help other researchers to assess the quality of their data faster.

4.4 Low-level features decoding

A typical memory access pattern we use in our research can look in the following way: *[WWWRWRW...RRWRRR]*. Such a pattern can be one of the few features needed to detect malware with high accuracy. However, it might be important to understand *what* exactly is happening when the process produces a certain pattern. The aforementioned pattern can not be interpreted by a human analyst, thus, we performed an attempt to correlate sequences of API calls and memory access patterns. Unfortunately, under our approach, it was impossible to find any meaningful correlation[12]. The approach published in P3 [12] is based on the assumption, that the best low-level features should correlate with the best high-level features: a *best-to-best* approach. In that paper, we have shown, that such an approach does not work for our experimental conditions. After P3 [12] was published we made an additional round of search utilizing *best-to-all* approach: we searched for the correlation between best low-level features and all high-level features. During the search we found, that a method for finding a correlation between high- and low-level features is correct and provides consistent results. E.g. a certain memory access pattern will always be found within the same or similar API-calls n-gram.

Nonetheless, suggested in P3 [12] method allowed us to make an important finding later in P4 [14]: many of the features selected by feature selection methods originated from *RtlAllocateHeap* routine from *ntdll.dll* standard Windows library.

4.5 Better possibilities for threat analysis

Often it is not only important to detect malware, but also to detect its category: type and family. The knowledge about malware type allows to understand *what* malware is doing and which measures should be taken to reduce threat brought by such samples. In its turn, the knowledge about malware *family* allows to understand *how* malware achieves its goals and what has to be done to restore the system after the attack[11] (P2). We have successfully shown, that memory access patterns allow to distinguish between malware types and families. Moreover, in P4[14] we show, that it is possible to train an ML model that can distinguish between 10 malware families and goodware based on the memory access patterns produced by BEP. Together with BEP malware detection, this finding creates a basis for future research, where it is important to find ***what* exactly makes BEP behavior of malicious and benign processes different**.

4.6 Better understanding of the phenomena

During the work on this Thesis, we heavily utilized various Machine Learning techniques. Often, the results we found contained various phenomena that had to be explored in a more thorough manner. While performing the analysis of these

phenomena we often had to use analysis approaches that, to the author's knowledge, are unique to our work. Sometimes (like in P6) our analysis did not support the hypotheses. However, they showed that the real reason for specific classification performance has to be explained with a different cause. For example, in order to explain classification performance from P1 we had to analyse intersections of feature vectors, the sparseness of data, and Area under Feature Class-wise frequency charts. The analysis of intersections of feature vectors was also applied in P5 and P6. Later, we analyzed the influence of "hidden" subcategories within classes to explain results obtained in P2 and P6. For the comparison of different feature selection methods, we used Difference ratio - a measure that shows how different are the feature sets selected by different methods. Properties of feature sets were also used in an attempt to explain classification performance from P6. There we showed, that number of selected features, the relative placement of samples in feature space, or feature-class representation imbalance can not explain several classification phenomena. We believe, that analysis methods of ML performance presented in our papers can help other researchers to achieve a better understanding of their results.

Chapter 5

Discussion

In this section, we elaborate on the theoretical and practical implications made in this thesis. We also outline potential limitations of the results and provide suggestions for future research.

5.1 Theoretical implications

Dynamic malware analysis is based on the study of the execution trace produced by executables. A perfect execution trace will incorporate all changes in the system caused by running an executable. However, it is barely feasible to trace and process *all* changes in the system. Instead, it is a common practice in malware analysis research to focus on one or few activity types. Fewer types and sources of activity may help to create malware detection models that are simpler, faster, and less susceptible to noise. However, deliberate reduction of the amount of data used to record an execution trace reduces the accuracy of this trace. Thus, simpler execution traces may result in weaker results in malware analysis and detection. In this Thesis, we show the capabilities of a relatively simple execution trace in malware detection and classification.

Malware analysis using low-level features may require the application of appropriate methods suitable for such task. The development of such methods requires a fundamental understanding of underlying processes and objects. It is necessary to utilize the knowledge of Portable Executable format, Windows process model, and computer architecture in order to fully leverage the advantages of low-level features in malware analysis. In this Thesis, we focus on the analysis of memory access traces generated by running Windows executables.

An executable running on the computer generates large amounts of various high- and low-level activity. In order to analyze such activity, one has to record or *describe* it. From the theoretical studies, it is possible to understand, that the *description* of a hardware-based trace of an executable will take more space than

that of a high-level. Execution of one API call involves the execution of hundreds of opcodes on the CPU. In their turn, modern x86 compatible CPUs can execute hundreds of different opcodes. From this, it is easy to derive, that the full low-level description of an execution trace may present a significant complexity for the analysis. In contrast, memory access operations have only two possible values: *read* and *write*. Execution of some opcodes does not involve memory access operations. Thus, using memory access patterns can decrease the accuracy of an execution trace. Nevertheless, memory access sequence allows for relatively simple analysis and provides great detection capabilities. In this Thesis, we show, that patterns derived from memory access sequence can be successfully used for malware detection and analysis. Moreover, a sequence of memory access operations can be considered as a binary sequence: elements of the sequence can take only two possible values. This opens a possibility for the future implementation of hardware-enabled malware detection, where all data is transformed into binary form.

In this Thesis we analyze memory access sequences by splitting them into the n-grams. In our first paper, we have shown, that increasing the size of n-gram results in higher classification accuracy. While increasing the size of an n-gram we increased the potential feature space. As the n-gram size reached 96, potential feature space became as big as 2^{96}. This raised the problem of feature selection. Our typical memory access sequence consists of around 1M of memory access operations. Under such conditions, each new trace added to the database can potentially increase the feature space of the model on 1'000'000-95 new features. Thereby, the simplicity of a trace came with a cost of the potential complexity of the feature space. In order to show that memory access patterns can be realistically used in malware detection, we had to utilize feature selection. This allowed us to create models based on only tens of features instead of the millions from the potential feature space. Thus, we showed that memory access operations can serve as a source of information sufficient for malware detection.

Malware analysis based on the low-level features brings logical desire to build a semantic gap between hardware and high-level activity. For example, it is natural for a human analyst to try to understand when exactly a certain memory access sequence is generated during the execution of an executable. While analyzing our traces we found, that most of the recorded data emerge before a newly launched executable reaches its Entry Point. This means that we are capable of detecting the running malware before it has a chance to perform any malicious actions in the system. We made an attempt to understand this phenomenon with a help of Windows documentation but found no good answers available in the available sources. Therefore, it is important to find this explanation in future work.

Findings present in this thesis show that:

1. simpler trace may result in bigger feature space;
2. it is possible to reduce a feature space from potentially millions of features to only tens while keeping the quality of the model on the appropriate level;
3. it is important to have a full understanding of Windows process model;
4. execution trace generated before the Entry Point can be successfully used for malware detection;

In summary, the theoretical implications of this thesis serve as a methodological basis for hardware-based malware analysis and detection.

5.2 Practical considerations

In this section, we describe practical considerations that have to be taken into account in order to perform research similar to the one described in this Thesis. In order to perform research on the topic of malware analysis, it is, in the first place, necessary to acquire malware samples. There are different ways of getting the malware: free and open collections, online sandboxes, and anti-virus vendors. There exist open repositories on Github or standalone websites (e.g. VirusShare) where one can freely download batches of malicious executables. Such resources may have different malware collections of different sizes. However, it might be problematic to find a specific malware type or family, since free resources often provide samples "as is" while malware-specific search capabilities are very limited. On the other hand, online sandboxes and malware analysis platforms (e.g. VirusTotal, Hybrid Analysis) often allow downloading samples from their collection. They also provide additional information about the samples that can be useful for the researchers: upload date, results of detection from various anti-virus vendors, results of basic static and dynamic analysis, etc. These systems normally have advanced search capabilities making it easier to find samples from a specific malware category or with certain characteristics (e.g. samples that contain certain strings). However, such systems often have download limitations for non-commercial users. It is either only possible to download one sample at a time or the number of downloads per day is very limited. The third option for getting the malware samples is a collaboration with anti-virus vendors. Part of their business is collecting the malicious executables which were found on the systems of their clients. Thereby, such companies have extensive collections of malware. However, they might not want to share some parts of their collection. For example, for security reasons they might not want to share the newest samples, as it might be wise not to reveal that some malware samples were already detected[45]. Thereby, the first step in malware-related research is obtaining the collection of malware of

the desired size.

When studying novel malware detection methods it is also important to acquire enough benign samples. It might come as a surprise, but getting enough goodware might be more challenging than getting enough malware. First of all, since malware is not protected by intellectual property laws it is easier from a legal point of view to share it. Furthermore, for some reason, there are very few places where one can get a batch of benign executables in an easy and straightforward manner. There are several options for getting benign Windows executables: open-source software repositories, clean Windows installation, free or portable applications repositories. Some researchers get benign samples from Github. However, it might be challenging to find repositories with executables (and not just a source code). Moreover, to get an adequate number of goodware samples one has to clone many repositories which have to be found in the first place. It is also important to check whether the found executables are benign. Other researchers use executables found in the clean installations of Windows. Windows comes with a variety of executables that are considered benign. However, getting many executables from one software company might skew the results of the analysis: there might be similarities between the samples introduced by coding and compiling practices used by a company. It might also be challenging to publish all of the findings since it can be considered as a violation of intellectual property laws. Another workaround for getting benign samples is the repositories of free or portables applications (PortableApps, portablefreeware, etc.). Some of them have tools that allow to download many free applications in a fast and straightforward manner.

The amount of malicious and benign samples depends on the purpose. However, it is generally considered that more samples allow to train better ML models. Thereby, it is recommended to acquire and use in the analysis as many samples as possible. On the other hand, analysis time grows with the number of samples. In the case of dynamic malware analysis, every sample requires a full cycle of analysis in the controlled environment what significantly increases analysis time. Thereby, it is important to adequately assess available resources (computing and time) when making a decision on the number of samples to be used in the analysis.

Having enough samples is only the first step. As we trained ML models to distinguish between benign and malicious it is necessary to properly label samples. For example, it is often necessary to check whether the free software that was downloaded from one of the repositories is actually benign. For these purposes, online sandboxes such as VirusTotal come into the serve. It might happen, that some of the anti-virus engines will detect such executables as malicious. And it is thereby important to decide whether to keep such sample as benign, drop it from the dataset, or move it to the malicious category. Moreover, some of the malware samples might be considered malicious by only a few anti-virus engines. Thereby,

it is necessary to decide which samples should be considered as truly malicious or truly benign. For example, to perform experiments presented in P4[14] we considered the sample as benign if none of the AV engines available at VirusTotal detected them as malicious. There were no issues with our malware samples, but we have anyway checked them for being labeled as malicious by at least 20 AV engines from VirusTotal. When studying the possibility of malware classification it is necessary to decide which AV engine will be used as a source of labels. It is important to understand, that different AV vendors use different names and naming schemes while arranging malware into categories. It often happens, that according to different AV vendors the same sample can belong to the different malware families. Moreover, some vendors provide either the name of the malware family or the name of the malware type. For our research, we decided to use labels assigned by the Microsoft AV engine. They follow the CARO[1] naming scheme that incorporates information about malware type, platform, family, and variant.

In order to perform dynamic analysis of many executables, it is wise to have a dedicated computer or server where one can automate data collection and processing. We performed our experiments on the Virtual Dedicated Server with 4-cores Intel Xeon CPU E5-2630 CPU running at 2.4GHz and 32GB of RAM with Ubuntu 18.04 as a main operating system and 3TB of storage. Data collected during dynamic analysis can require a significant amount of storage. For example, behavioral traces collected for papers P4, P5, P6 [14][10][13] could take as much as 1GB of storage per sample. This fact brings attention to another practical consideration. Before running full-scale experiments one should decide on the amount and type of events included in the behavioral trace. In our case, some of the data that we collected was not used to obtain results. For example, we did not use recorded opcodes, but they can be used in future research. In its turn, the data processing might put additional requirements for the analysis platform. For example, while performing feature selection we often used up all of the 32GB of RAM. Thereby, it is recommended to have as much RAM as possible as the number of unique low-level features generated by several thousand samples might be as big as tens of millions.

To make a collection of low-level features possible it is necessary to use appropriate tools. For our research, we used Intel Pin[27] DBI tool. It provides extensive functionality for the control and analysis of the running executable. Intel Pin provides an API that allows the creation of tools with the desired functionality using the C++ programming language. The only problem with Intel Pin is that its documentation is not detailed enough. Thereby, we recommend looking into the open-source tools that come with the Intel Pin to make yourself familiar with the usage of various Pin's functions.

Studying low-level features enabled malware analysis requires a significant

amount of preparations. For most of the papers presented in this Thesis preparations for data collection (finding benign and malicious samples, creating Intel Pin tools), data collection (running samples in VM), and processing (feature selection) took significantly more time than training ML models and analysis of the results.

5.3 Ethical and Legal aspects

Every research work involves using the *methods* that lead to the *results*. Both methods and results can cause certain ethical and legal issues. For example, methods of obtaining experimental data can be both illegal and unethical. Furthermore, information and knowledge presented as the results may in some cases be used for malicious intentions. Thereby, it is considered a good practice to elaborate on such issues. In this section, we elaborate on possible ethical and legal issues linked to the research presented in this Thesis.

5.3.1 Ethical aspects

There are several ethical issues that are invoked by the mere fact of publishing of our research. In our work, we disclose details of the novel malware detection approach. The cybersecurity landscape is a continuous arms race. When the defense against certain adversarial activity is created, adversaries try to invent new methods to achieve their goals. For example, malware authors may investigate the possibilities and limitations of anti-virus solutions in order to make new variants of malware more detection-proof. Thereby, by showing a novel method for malware detection we might unintentionally help malware authors to take it into account while creating new variants of malware. However, we believe, that fine-tuning memory access activity is too complicated for the majority of malware writers.

In each of the papers presented in this Thesis we use Machine Learning for the analysis of our data. The ML algorithms and models have their weaknesses, exploitation of which is called *adversarial learning*[25]. One of the tasks of adversarial learning is to fit a certain input (e.g. malware) so that the ML model will classify it as goodware. It is considered, that the more information about ML-enabled system adversaries have - the easier it is to conduct adversarial learning[16]. As we disclose which ML algorithms and data we used for malware detection, a potential adversary may use this information for malicious purposes. However, as we are not presenting the implementation of the real malware detection system - this issue should not be considered severe.

The last ethical issue arises from the general idea of this Thesis. We state, that it is hypothetically possible to detect malware based on its hardware activity. Thus, in the future, it might be possible to implement specialized hardware module capable of malware detection. However, we have to inform the reader that even existing hardware-based security solutions (e.g. Trusted Platform Modules, Intel

Software Guard Extensions) are known to have their own vulnerabilities and be susceptible to the attacks[18]. Thereby, it is important to understand that hardware may have its own security flaws.

5.3.2 Legal aspects

In this Thesis, we use reverse engineering methods in order to get the necessary data from benign and malicious executables. Reverse engineering is aimed at revealing the internal structure of the software and presenting it in high levels of abstraction. While analysing benign executables it is possible to stumble upon proprietary software which is often protected by various intellectual property laws and regulations. Thus, revealing the results of reverse engineering can be potentially seen as a violation of intellectual property[21]. From one point of view, it might be possible for the researchers to use only open-source and free software where reverse engineering is not forbidden by laws. However, this can result in a less valid result of the research, since the datasets will be less real-life ones. Moreover, in our research, we present findings of the dynamic analysis of standard Windows libraries. When an executable is launched on Windows, it is impossible to avoid the execution of the binary code from libraries responsible for process creation and its launch. Thereby, as we published memory access patterns that emerged from windows libraries one may consider this as a violation of intellectual property. However, there are several arguments on our side. First of all, the amount of published memory access patterns is relatively small and can't be used for the disclosure of the internal structure of proprietary libraries. Moreover, specific patterns may be specific to the systems with Intel CPUs, thereby different on systems with different CPUs. Furthermore, reverse engineering made on fair use principles is allowed under various circumstances[2][3][37]. Among the others, fair use principles include: reverse engineering for research purposes; avoiding unnecessary reverse engineering of the whole product; impossibility to obtain information by means other than reverse engineering. This thesis complies with all of the above-mentioned principles of fair use.

The real-world implementation of the system that utilizes methods proposed in this Thesis can be potentially seen as a system capable of data interception. Potentially, memory access patterns can be used to detect malicious activity from outside of the VM where the web application is running. Thereby, an Internet hosting provider that monitors memory activity on the user servers for security purposes may reveal sensitive and confidential information. However, such issues should be solved at the stage of the agreement between the user and provider.

5.4 Limitations and Future work

In this section, we describe the limitations present in our methodologies and results. First of all, as in any research that utilizes Machine Learning methods, it is important to understand that classification performance and results of feature selection might be influenced by our datasets. For example, it might be possible to find a dataset on which the malware against goodware classification performance will be significantly worse than in paper P1[15]. In contrast, it can be possible to create a dataset that will allow achieving 100% multinomial classification accuracy using the approach presented in P2[11]. Thereby, to eliminate potential flaws in the validity of the results it might be necessary to conduct large-scale testing of the proposed methods.

The other limitation is brought by the design of the experiment presented in P3[12]. There we had a hypothesis, that best memory access patterns have to correlate with best API calls n-grams: we called it *best-to-best* approach. However, we proved this hypothesis wrong. In P3 we outlined, that it might be necessary to repeat the study, but using a *best-to-all* approach, where one would explore potential correlations between best memory access patterns and all API calls n-grams. Such a search is quite time-consuming, thus we were not able to finish it by the time of writing the P3 paper. However, we performed that search for parts of our data later and found, that memory access patterns appear within certain API call sequences in a quite consistent manner. Thus, in future work, it is important to find all consistent correlations between low- and high-level features. It is also important to take into consideration the BEP-AEP approach. So that a potential correlation between memory access patterns and API call sequence is presented under the context of either BEP or AEP behavioral activity.

One of the biggest limitations in the analysis of memory access patterns is the amount of data that has to be processed. As we used only the type of memory access operation, the number of features often reached millions. Thereby, we had to perform feature selection in order to make training of ML models possible. However, we often had to perform a two-stage feature selection process: first, go down to 50K of features with IG and later use CFS to decrease the number of features to several dozens. Unfortunately, we could not use CFS on the full feature sets since it would require an infeasible amount of computational resources. Thus, we might have missed some of the good combinations of features that could help us to improve the classification performance of ML models.

We performed all our experiments on Windows operating system. Thereby, our conclusions are limited to this system. In the future, it is necessary to test our approach on operating systems other than Windows. We also used the same server with the same Intel CPU. It is hypothetically possible, that running similar experiments on systems with different CPUs (AMD, ARM, etc.) can show results

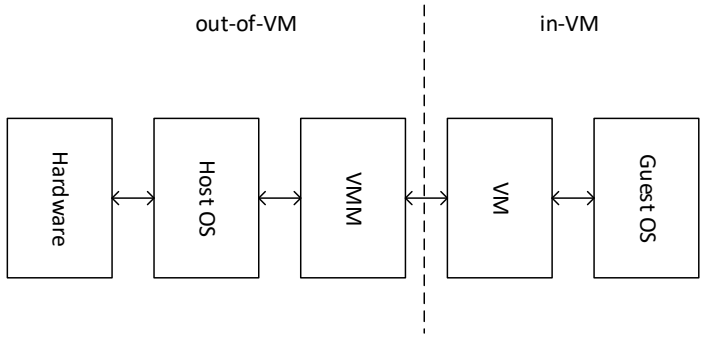


Figure 5.1: Out-of- and in-VM sources of information

different from those presented in this Thesis. Thus, in the future, it is important to test our approach on systems with CPUs other than Intel.

In this Thesis we collected low-level or hardware-based features using software tools launched in the same VM as samples from the dataset. As we stated above, malware can have a functionality aimed at disrupting of the analysis and detection mechanisms. Hypothetically, any monitoring tool that runs in the kernel or user level in OS can be detected by malware. Thus, malware can try to disrupt the operations of such tools thus thwarting protection and analysis. Moreover, the detection of monitoring tools may be used by malware as a reason to stop running, hence not revealing functionality and thwarting analysis. We believe, that low-level features have the potential to be used for out-of-VM malware analysis and detection (see Figure 5.1), as it is very unlikely that out-of-VM tool can be detected from inside the VM. Thereby, in the future it is necessary to: a) test the possibility of out-of-VM malware detection based on low-level features; b) test the possibility of a hardware-based solution that makes malware detection using low-level features possible.

5.5 Bibliography

[1] A new virus naming convention. http://www.caro.org/articles/naming.html. accessed: 20.04.2021.

[2] European software directive, art. 6(2), 1991 o.j. (l 122) at 45, 1991.

[3] Limitations on exclusive rights: Fair use, 17 u.s.c. § 107 (2012), 2012.

[4] Pin 2.14 user guide 2016: Memory reference trace (instruction instrumentation). https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html#MAddressTrace, 2016. accessed: 2016-4-14.

[5] Najwa Aaraj, Anand Raghunathan, and Niraj K Jha. Dynamic binary instrumentation-based framework for malware defense. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 64–87. Springer, 2008.

[6] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. On the feasibility of malware authorship attribution. In *International Symposium on Foundations and Practice of Security*, pages 256–272. Springer, 2016.

[7] AndyNor.net. Df2: Reverse engineering part 2. https://andynor.net/blog/471/, 2015. accessed:2016-4-15.

[8] AVTEST. The independent IT-Security Institute. Malware. https://www.av-test.org/en/statistics/malware/, 2020.

[9] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 703–708. IEEE, 2014.

[10] Sergii Banin. *Malware Analysis using Artificial Intelligence and Deep Learning: Fast and straightforward feature selection method: A case of high dimensional low sample size dataset in malware analysis*. Springer, 2020.

[11] Sergii Banin and Geir Olav Dyrkolbotn. Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107–S117, 2018.

[12] Sergii Banin and Geir Olav Dyrkolbotn. Correlating high-and low-level features. In *International Workshop on Security*, pages 149–167. Springer, 2019.

[13] Sergii Banin and Geir Olav Dyrkolbotn. Detection of previously unseen malware using memory access patterns recorded before the entry point. *The 4th International Workshop on Big Data Analytic for Cyber Crime Investigation and Prevention*, 2020.

[14] Sergii Banin and Geir Olav Dyrkolbotn. Detection of running malware before it becomes malicious. In *International Workshop on Security*, pages 57–73. Springer, 2020.

[15] Sergii Banin, Andrii Shalaginov, and Katrin Franke. Memory access patterns for malware detection. *Norsk informasjonssikkerhetskonferanse (NISK)*, pages 96–107, 2016.

[16] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[17] Daniel Bilar. Opcodes as predictor for malware. *International journal of electronic security and digital forensics*, 1(2):156–168, 2007.

[18] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, et al. Advances and throwbacks in hardware-assisted security: Special session. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 1–10, 2018.

[19] Bevin Brett. Memory performance in a nutshell. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html. accessed: 28.11.2022.

[20] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[21] Julie E Cohen. Lochner in cyberspace: The new economic orthodoxy of" rights management". *Michigan Law Review*, 97(2):462–563, 1998.

[22] Testimon Research Group. Testimon research group. https://testimon.ccis.no/, 2017.

[23] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[24] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[25] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58, 2011.

[26] Eric M Hutchins, Michael J Cloppert, Rohan M Amin, et al. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1(1):80, 2011.

[27] IntelPin. A dynamic binary instrumentation tool, 2017.

[28] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2015.

[29] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[30] Fang Li, Chao Yan, Ziyuan Zhu, and Dan Meng. A deep malware detection method based on general-purpose register features. In *International Conference on Computational Science*, pages 221–235. Springer, 2019.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[32] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[33] John von Neumann. First draft of a report on the edvac. Technical report, 1945.

[34] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.

[35] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[36] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry V Ponomarev. Hardware-based malware detection using low level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[37] Vinesh Raja and Kiran J Fernandes. *Reverse engineering: an industrial perspective*. Springer Science & Business Media, 2007.

[38] Reuters. Ukraine's power outage was a cyber attack: Ukrenergo. https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA, 2017.

[39] Hossein Sayadi, Nisarg Patel, Sai Manoj PD, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[40] Mike Schiffman. A brief history of malware obfuscation: Part 2 of 2, 2010.

[41] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. Machine learning aided static malware analysis: A survey and tutorial. In *Cyber Threat Intelligence*, pages 7–45. Springer, 2018.

[42] Andrii Shalaginov, Lars Strande Grini, and Katrin Franke. Understanding neuro-fuzzy on a class of multinomial malware detection problems. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 684–691. IEEE, 2016.

[43] Claude E Shannon. A mathematical theory of communication, part i, part ii. *Bell Syst. Tech. J.*, 27:623–656, 1948.

[44] R. Shirey. Internet security glossary, version 2. RFC 4949, RFC Editor, August 2007. http://www.rfc-editor.org/rfc/rfc4949.txt.

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

[46] Cornell University. Code optimization: Memory access times. https://cvw.cac.cornell.edu/codeopt/memtime. accessed: 28.11.2022.

[47] Ryan J. Urbanowicz, Melissa Meeker, William La Cava, Randal S. Olson, and Jason H. Moore. Relief-based feature selection: Introduction and review. *Journal of Biomedical Informatics*, 85:189–203, 2018.

[48] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

[49] Zhixing Xu, Sayak Ray, Pramod Subramanyan, and Sharad Malik. Malware detection using machine learning based analysis of virtual memory access patterns. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 169–174. IEEE, 2017.

[50] Pavel Yosifovich. *Windows Internals, Part 1 (Developer Reference)*. Microsoft Press, may 2017.

[51] Çağatay Yücel and Ahmet Koltuksuz. Imaging and evaluating the memory access for malware. *Forensic Science International: Digital Investigation*, 32:200903, 2020.

Part II

Publications

Chapter 6

P1: Memory access patterns for malware detection

Sergii Banin, Andrii Shalaginov, Katrin Franke

Abstract

Malware brings significant threats to modern digitized society. Today exist many malware detection techniques, yet malware developers put in significant efforts to evade detection and remain unnoticed on their victims' computers, such as through encryption and obfuscation that tend to eliminate known and noticeable traces in memory, network or disk activities. Because of this, there remains a strong need for new malware detection methods, especially ones based on Machine Learning models, because processing of large amounts of data are not suitable task for human. This paper presents a novel method that could potentially detect zero-day attacks and contribute to proactive malware detection. Our method is based on analysis of sequences of memory access operations produced by binary file during execution. In order to perform experiments we utilized an automated virtualized environment with binary instrumentation tools to trace the memory access sequences. Unlike the other relevant papers, we focus only on analysis of basic (Read and Write) memory access operations and their n-grams rather than on fact of presence or overall number of operations. Additionally, we performed a study of n-grams of memory accesses and tested it against real-world malware samples collected from open sources. Collected data and proposed feature construction methods resulted in accuracy of up to 98.92% using such Machine Learning methods as k-NN and ANN. Thus, we believe that our proposed method will serve as a stepping stone for better proactive malware detection techniques in the future.

6.1 Introduction

Malware is the malicious software designed to perform illegal or unwanted activity on a victim's system. The VirusShare database [23] contains 25,072,568 malware samples as of 9th May, 2016. When a new malware sample is detected there is a time gap between the moment antivirus vendors can analyze it and the moment they update their databases for their customers. To thwart detection, malware developers develop additional techniques to evade detection by antimalware software through the use of different obfuscation techniques such as encryption, polymorphism, metamorphism, dead code insertions, and instruction substitution [20] in order to change the appearance of a file and its static characteristics. For example, it is possible to change hash sums used as file signatures (such that SHA-1 or md5) by simply changing different strings in the file. Further, dead code insertions can be used in executables to change opcode sequences, making detection troublesome.

There are two main approaches for malware analysis that can be found in the literature [3, 11]: *static* and *dynamic*. *Static analysis* is done on a malicious file without its execution and aimed at collecting various static characteristics such as bytes, opcodes and API n-grams frequencies, Portable Executable header features, strings and others [20, 21, 8]. *Dynamic analysis* is based on running a malicious executable in a controlled environment and tracking its activity within the system. Such activities include network, registry and disk usage patterns, API-calls monitoring, instruction tracing, memory layout investigation and others [5]. To collect such information one can use either specialized sandboxes like Cuckoo [9] or utilize any Virtual Machines such as VirtualBox accompanied by a debugger or other watchdog software. Despite the fact that some authors consider [24, 18] disk and network activities crucial for malware detection, few authors have outlined the utility of memory properties analysis [1, 12].

This paper presents a novel methodology for malware detection. It is based on the extraction of the memory access sequences (further called *memtraces*) for both benign and malicious executables using the dynamic binary instrumentation tool Intel Pin [10]. This dynamic binary instrumentation tool is used for live analysis of binary executables and allows for analyzing different properties of execution such as memory activity, opcodes, addressing space, etc. Our proposed methodology is based on the assumption that similar opcodes with similar arguments will result in nearly the same memtraces, which is explained later in the paper. Thus, we apply an n-gram technique to extract features from memtrace sequences in order to perform benign against malicious classification. Moreover, we used specially-tuned feature selection to be able to verify classification accuracy while adhering to a set of community-accepted Machine Learning (ML) methods. It will be shown that our method can find an application in proactive malware analysis with reliable

results using only fraction of the execution records of the malware sample. Unlike the other related works, where authors worked on specific ML methods and other dynamic features, we focus primarily on memory access sequences and patterns within them. So, this paper contributes to a new malware detection methodology and test it against real-world samples.

The reminder of the paper is organized as following: 6.2 presents an overview of the dynamic malware analysis, including existing behavioural characteristics as well as how memory activity can be used for identification of malicious activities. Further, 6.3 presents our contribution towards the memtraces analysis for malware detection. Description of the collected malware samples and analysis of the results are given in the 6.4 section. Finally, 6.5 contains our final remarks and conclusion.

6.2 Memory patterns in malware detection

In this section we provide a short overview of existent studies that are related to our, because there dynamic binary instrumentation tools and memory activity analysis were also used. Dynamic malware analysis involves malware execution in controlled environment with further investigation of its activity and any possible traces that can be found in the system. In the Malware Analysis Cookbook, Ligh et al. [14] defined the following automated procedure for dynamic malware analysis covering a set of predefined operations ranging from VM start up to traces collection.

According to SANS [3] one may conclude that a number of behavioural characteristics can be used to identify whether or not an executable file has some malicious functionality. With a use of such dynamic malware analysis, it is possible to collect different types of features such as file system events, registry changes, API and DLL calls, network and memory activity [24]. In the paper [18], the authors claim that memory analysis without ground-truth can't be considered trustworthy, especially on proprietary operating systems (e.g. MS Windows). They investigate the accuracy and efficiency of traversal-based and signature-based memory analysis tools (Volatility [7] framework and its plug-ins), which are designed to gather information about processes, modules, files etc. Further work [18] also examined accuracy and efficiency of robust field- and graph- based signature schemes SigField [4] and SigGraph [15]. They compared results from binary analysis tool and Volatility over the Virtual Machine memory, claiming that traversal-based and signature-based methods tend to produce less accurate results.

A methodology for malware analysis with using Intel Pin was discussed earlier [1]. The model was first tested in virtual environment and afterwards in a real environment with Windows XP or Xen Linux installed. They extracted the following characteristics: system or user API calls if any file or folder was modified, calls that create hard or symbolic links, calls or arguments of function *exec()* and in-

structions that performed memory operations *read* and *write*. Unlike in this paper, where we focus on single memory access operations generated by single opcodes, they utilized *basic blocks* of a program. The basic block is an instruction sequence which is executed between control flow transfer instructions. Among other features, the authors used fact of presence, size of transferred data and memory range of memory access operations within the basic blocks. Recording the execution trace they generated regular expressions and security policies, which then were used for malware detection. As the result, they achieved 100% detection rate for original and obfuscated malware samples on both Windows and Linux. The authors claim that their system is capable of accurate malware detection with 93.68% code and path coverage of input-dependent executables. Finally, it is worth mentioning the work [12] that proposed the ensemble learning technique of malware detection based on a number of features extracted with Intel Pin [10]: frequency of opcode occurrence, presence of particular opcode, difference between frequency of opcode in malware and benign executables, distance and presence memory reference and total number of load and store memory operations as well as branches. For each executable they collected a feature vector for every 10,000 committed instructions and achieved classification accuracy up to 95.9% with a specialized ensemble classifier.

6.3 Memtraces for malware detection

The proposed method based on memtraces is described below. Steps from characteristics collection and feature construction for future use in ML are similarly presented.

6.3.1 Collecting memory access

Opcode (API calls) n-grams have consistently been successfully utilized as reliable features for malware detection [16]. No matter the programming language or frameworks used for developed programs, a compiled PE32 executable can be represented as a sequence of opcode instructions. Opcodes (or assembly commands) are basic commands executed on the hardware level. Some operate only with CPU's registers, and as such $XOR\ EAX, EAX$ or $MOV\ ESI, EBX$ won't have any interaction with virtual memory, while others can generate sequences reading from and writing to virtual memory operations, for instance $MOV\ EBX$, VAR_NAME which reads from memory and $MOV\ [VAR\_NAME]$,
110 which writes to it.

In this paper we analyse sequences of basic memory access operations which are *R* for Read and *W* for Write operations. Our goal is to record a sequence of memory access operations, or *memtraces*, and analyze this sequence. The majority of modern desktop computers utilize x86 compatible architectures that were

introduced in order to implement pipelines and, as a result, increase execution speed. Modern x86 compatible CPUs translate opcodes into a sequence of micro-operations (or *uops*) responsible for loading and storing data, interacting with arithmetic logical units, branching, and so on, each *uop* executed on the specific port. Some authors collected information about number and types of micro-operations used by CPUs in order to execute certain opcodes, as in [6] where such information was collected for Intel architectures ranging from Pentium to the Skylake architecture. For example, in Sandy Bridge architecture *port p23* stands for memory read or address calculation, and *p4* for memory write.

To be more specific, we focus on opcodes that allow interaction with memory such as MOV, AND, XOR, ADD etc. It was found that, regarding the number of *load* and *store* micro-operations, the opcodes were similar to those presented in the book [6]. Our scope was confined to solely memory-related activity, and we looked for a number of micro-operations going to a memory read port (e.g. *p2* or *p3* for Ivy Bridge and Skylake architectures) or a memory write port (e.g. *p4* for Ivy Bridge and Skylake architectures). No similar information was found for AMD CPUs, so the scope was also limited to Intel processors. With the help of Intel Pin, we checked data that are usually transferred by the detected read and write operations where we found that most of the memory operations involve transfers of 4 bytes. (1- 2- 8- and 10-byte memory accesses were also found.) This means that Intel Pin is capable of detecting memory operations on the level of separate opcodes and has a desired granularity. From the results of this study we concluded that opcode with similar parameters will generate similar memory access sequence regardless of the overall task of the executable and Intel CPU model. To verify this we tried, with help of Intel Pin, to make the output contain executed opcodes and its (if exists) memory operations. The examples of opcodes and memtraces captured from *calc.exe* benign executable taken from Windows 7 are given below.

```
[mov edi, dword ptr [ebp-0x20]]
R
[add dword ptr [eax], ecx]
RW
[mov dword ptr [ebp-0x8], edx]
W
```

This sequence can be explained as following:

- *mov edi, dword ptr [ebp-0x20] reads* from memory, and writes this information to the *edi* register. According to [6] instructions of type *MOV Register, Memory* for all addressing types involves 1 read operation (1 microoperation for the port p23).

- *add dword ptr [eax], ecx reads* data from address that is previously calculated (by the memory read port which has additional function of address

calculation), then it calculates the sum and later data is *written* to the already calculated address. Instructions of type *ADD Memory, Register* involves 2 microoperations to the store port (one for address calculation and one for reading) and 1 microoperation for writing [6].

- Instruction *mov dword ptr [ebp-0x8], edx* generates *W* (*Write*) because *MOV Memory, Register* is a memory writing.

Based on what was said above, we highlight two hypotheses: (i) Opcode n-grams are reliable features for malware detection as described earlier, and (ii) Opcodes with similar arguments will produce similar memory activity. As results, memtraces can be used as robust features for identification of malware samples. It is hard to say how many memtraces are required for good detection rate; this will have to be studied later on. To start with however, we decided to restrict length of recorded memtrace sequence to 10 millions of records. According to our measurements, in order to perform 10,000,000 memtraces, an executable (from our dataset) spends about 0.053 seconds on our hardware. Time was recorded with a use of *chrono* a C++ lirary. This makes our system potentially applicable for the systems which require near real time malware detection, because classification of a software sample will take less than a second. Afterwards, the original memtrace sequence is pruned to get first 100,000 and then 1,000,000 memtraces in order to study influence of the memtraces sequence length on the accuracy.

6.3.2 N-gram as feature extraction

In order to detect malicious executable we record the sequence of memory access operations(memtraces). We define original memtrace sequence $S_{original}$ as a set of memory access operations: $S_{original} = (m_1, m_2, ...m_l)$ where *l* is the number of memtraces recorded during execution of a program and m_i is either Read or Write memory access operation. Memtrace sequence *ms* is defined as subgroup of original memtrace sequence where $ms \subseteq S_{original}$. Here *ms* is constructed from memtraces: $ms = (m_{k+0}, m_{k+1}...m_{k+p-1})$ where $k \in [1, l-p+1]$ is the starting position of certain memtrace sequence, and $p \in [1, l]$ is the length of memtrace sequence. So, the memtrace sequence of length *p=n* is called n-gram. We use only R for read and W for write operation regardless to size of the transmitted data. Then we extract n-grams of preferred size from the original memtrace sequence. While authors who apply opcode n-grams for virus detection usually use *n=1, n=2* [19] *n=3, n=4, n=5* [22], we could not use *n* of such small sizes. Here are the reasons:

- Executable may contain hundreds of different opcodes. Thus, sequences of *OR,OR,OR* and *ADD,ADD,ADD* represent different features for opcode based methods.

- We trace only memory accesses. Thus, both *OR,OR,OR* and *ADD,ADD,ADD* (*Memory, Register* operands) could be recorded as *RWRWRW* and *RWRWRW*.

- Memtraces sequence is a binary sequence, because it contains only two symbols. For the length of 1 or 10 million, class-wise frequency for memtrace n-gram of size 6 will be close to uniform, and there is a high likelihood, that among benign and malicious classes there will be no a single unique n-gram for particular class.

During initial experiments we also found that there is no class-unique n-grams up to the size of 12 for the dataset used for experiment. It can be explained by the fact that not all the opcodes generates memtrace activity. From one perspective it makes less data to work with, but from another it could probably result in lower classification accuracy. So, we decided to start from n-gram size of *16*, and proceed with *20,24,36,48,72,96* to cover different n-grams and have feasible processing and analysis overhead. Another reason of increasing n-gram size is that probability of particular n-gram to occur in a sequence of memtraces is higher for smaller n values, thus utilizing small n values can result in impossibility of finding unique features and low classification accuracy. We limit n-gram size to 96 because our scripts were not able to finish bitmap construction for 10,000,000 memtraces due to out of memory error.

6.3.3 Feature Selection

To use extracted memtraces for training ML models, we need to extract features that will provide the best description of classes [13]. Ideal feature for classification task is the feature, that exists only in particular class and is present in all instances of this class. However, in real-world problems, it is usually very hard or even impossible to find such features. So, the task is to find features that fit previously stated request better than others using the following steps:

1. For each class (benign and malicious) construct vector of n-grams (which are unique within the class) and their class-wise frequencies, e.g.: $[[WWR, 1.0], [WRW, 0.87], [RRR, 0.66], ...]$

2. Having two vectors, one for benign and one for malicious executables, we delete those n-grams that are present in both vectors, regardless to their class-wise frequencies. In other words, we subtract intersection of two sets from each of them and get two *clean* vectors.

3. From each of clean vectors we select particular amount (e.g. 100,200,400) of n-grams with highest class-wise frequency and combine them into the final feature vector of length 200, 400 or 800. The numbers 100, 200, 400

were chosen, because many researchers used to utilize feature numbers from 100 to 1,000. So, this a common baseline for similar researches, yet we need to take into account ML software performance.

However, to be able to apply ML classification we need to build a bitmap (matrix) of presence, where "1" is placed if particular instance contains particular feature and "0" if not. Such bitmap is used later to train ML methods. In order to assess model quality we will use 5-fold cross validation. For results assessment we will use classification accuracy because it shows how well model performs on the whole dataset. Finally, newly built feature vector is then used for building the bitmap that is later used in Machine Learning algorithms.

6.4 Experiments & Results

This section is devoted to experiments design and analysis of achieved results of the proposed method.

6.4.1 Computing Environment

All our experiments were performed on Virtual Dedicated Server (VDS) with Intel(R) Core(TM) CPU @ 3.60GHz, 4 cores, SSD RAID and 48GB RAM. Ubuntu 14.04 64 was installed with MySQL 5.5, PHP 5.5.9 and VirtualBox 5.0.16. Windows 7 32-bit was used as guest OS, because of its wide spread [17] and the fact that malware written for 32-bit OS's will run on 64-bit as well. Another reason to use 32-bit version of Windows 7 is that our VDS was not capable of running newer or 64-bit versions of Windows due to virtualization issues.

6.4.2 Malware & data collection

Benign files were collected from clean installations of 32 bit versions of Microsoft Windows OS (XP, 7, 8, 10). The reason of such choice is that there is no publicly available datasets with big amount of benign files. Malicious files were taken from *VirusShare* repository [23] (*VirusShare_00207.zip*). This archive contains collection of PE32 files, which is a very popular executable format due to strong legacy of 32-bit operational systems and compatibility issues. Both benign and malicious files were unsorted and uncategorised. In order to avoid duplicates, files were renamed with their *md5* sums. Having information gained from *peframe* [2] those files, that contain *GUI* field in the peframe output, were selected. The reason of this kind of filtering is that many malicious executables, that don't have GUI, can switch to idle mode soon after start, so, this will bring significant problems to automated dynamic analysis since it could produce very small amounts of data or it will require an unreasonably long waiting time. After files were filtered

by presence of GUI, the smallest 1,000 files from each of the datasets were selected. File sizes of selected benign and malicious executables vary from 115.4 KB to 152.1 KB and from 976 bytes to 28.7 KB respectively. The experiments setup included automated execution of malware as specified in the Figure 8.2 according Lingh et al. [14].

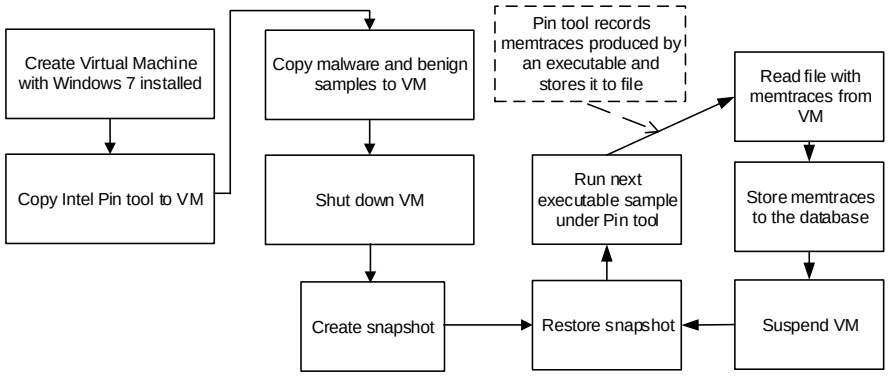


Figure 6.1: Automated malware analysis cycle using Intel Pin for metraces sequences extraction

During the experiments only 445 benign and 759 malicious files managed to start since some had *anti-debug* or *anti-VM* features. Final dataset contains 1,204 files and corresponding MySQL table of raw memtraces table occupies 6.9 GB of storage space and table of 96-grams for memtrace sequence length of 10,000,000 occupies 32.2 GB.

6.4.3 Results

We used memory access sequence of lengths 100,000; 1,000,000 and 10,000,000, n-gram sizes of 16,20,24,36,48,72,96 and feature set sizes of 200,400, and 800. The following ML methods were trained: NB (Naive Bayes), BN (Bayesin Network), J48 (C4.5), k-NN (k-Nearest Neighbours), ANN (Artificial Neural Network) and SVM (Support Vector Machine). In this paper we present only the most outstanding achieved results.

Influence of the memtrace sequence length and n-gram on classification accuracy

Considering three lengths mentioned before, we can say, that usage of the first 1,000,000 memtraces is enough to achieve good malware detection rate together with the number of selected features equal to 800. The results of experiments are given in the Table 6.1. Other configurations gave lower or equal results.

We can see that k-NN and ANN provide best classification accuracy of **98.92%**

n-gram size	Machine Learning method					
	NB	BN	J48	k-NN	ANN	SVM
100,000 memtraces						
16	60.22	60.47	**64.29**	64.20	63.54	63.54
20	60.88	62.71	65.03	**65.61**	63.54	63.54
24	62.46	63.95	67.19	**67.28**	62.13	63.54
36	59.72	60.63	67.61	**67.77**	62.54	63.62
48	59.72	59.72	68.77	**68.94**	62.96	64.45
72	64.12	64.12	**71.26**	**71.26**	64.70	67.28
96	67.03	67.03	70.76	**71.01**	63.87	68.77
1,000,000 memtraces						
16	60.71	61.30	82.97	**83.80**	**83.80**	70.02
20	55.32	55.32	83.89	**84.88**	61.38	77.74
24	56.15	56.81	79.49	**79.57**	61.38	76.50
36	57.64	57.64	78.16	**78.32**	65.70	76.16
48	73.34	73.34	85.71	**85.88**	65.86	85.05
72	**92.11**	**92.11**	90.95	91.20	92.03	92.03
96	94.44	94.44	98.84	**98.92**	**98.92**	98.51

Table 6.1: Accuracy %, for 800 features

for 1,000,000 memtraces, 800 features and 96-grams. k-NN also shows most of the row-wise best results in the tables. However, the point where k-NN and ANN reached best accuracy could not be stated as optimal condition for malware-benign classification task. Several results were not gained due to out-of-memory problem, but better results could probably be achieved in future work. From one point of view k-NN is a good algorithm, because it doesn't require actual training phase. However, it makes no generalization about processed data. This can result in overfitting and classification time growth. Among others J48 and ANN showed good results and could be considered as reliable candidates for malware-benign classification task if k-NN is not suitable for some reason. As we can see from the Table 6.1, the more memtraces we have in the original sequence - the higher accuracy we gain. This is natural dependency, because bigger length of memtrace sequence gives us more information about an executable. As we can see, n-grams (48, 72, 96) of bigger size gives us better accuracy. Finally, we have also studied influence of the number of features on classification accuracy. Similarly to memory access sequence length, overall dependency shows growth of accuracy with growth of feature number as shown in the Figure 6.2.

Analysis of the classification accuracy

The non-trivial factors that influence classification accuracy are given in the Table 6.2 first. During the feature selection process we used to calculate intersection of unique n-grams sets from benign and malicious executables. This intersection then was subtracted from both benign and malicious unique n-gram sets. From

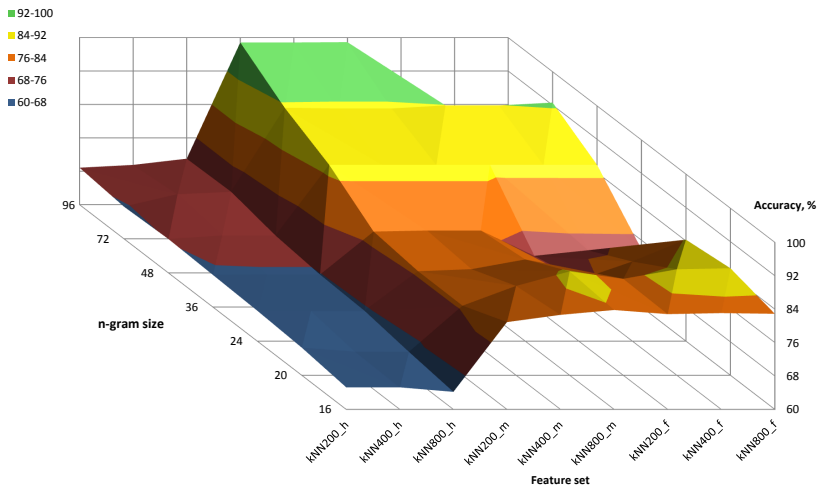


Figure 6.2: Accuracy depending on n-gram size and number of features (200-800) for all memtrace sequences lengths

the Table we can see that characteristics of intersection under different memtrace sequence lengths and n-gram sizes. It contains the following columns: *n-gram size* - the size of n-gram, *Isec* - the number of unique n-grams shared between benign and malicious executables (intersection size), *B_unique* - the number of unique n-grams found in benign executables, *M_unique* - the number of unique n-grams found in malicious executables, *Ratio* - intersection ratio, shows similarity between malicious and benign n-grams sets. Calculated as $Ratio = \frac{Isec}{(B\_unique + M\_unique)}$ The Table 6.2 contain intersection ratios calculated for the 1,000,000 memtraces.

n-gram size	isec	B_unique	M_unique	Ratio
16	50,751	53,587	56,722	0.46008
20	156,553	209,037	240,098	0.34857
24	234,725	364,926	440,593	0.29140
36	372,336	670,314	1,009,935	0.22160
48	481,347	910,659	1,655,659	0.18756
72	694,570	1,384,718	2,944,111	0.16045
96	918,076	1,884,036	4,201,454	0.15086

Table 6.2: Intersection size and ratio for unique benign and malicious n-grams for 1,000,000 memtraces

6.4.4 Interpretation of achievend results and findings

In the Figure 6.3 we show dependency between accuracy rate and intersection ratio for 1M of memtraces and all lengths of feature vector. Accuracy rates were taken from k-NN column of corresponding result table, because k-NN has shown most of the best results for particular n-gram size. Logarithmic trendlines were added to every series for easier understanding. We will use k-NN's accuracy to illustrate other findings and tendencies for the same reason.

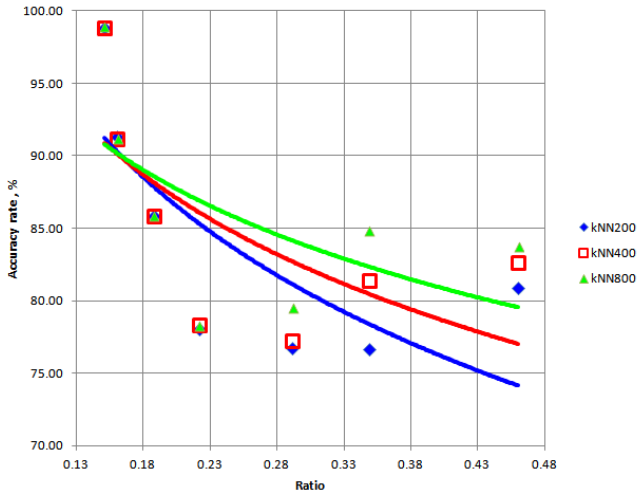


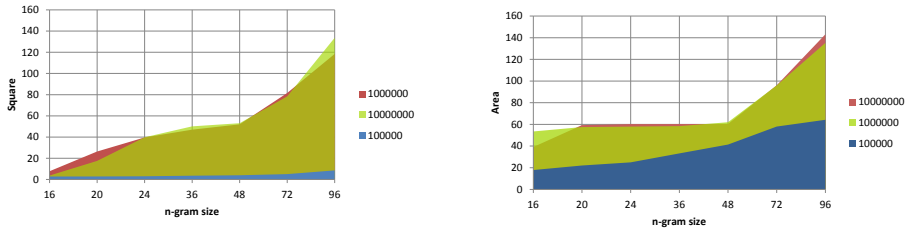
Figure 6.3: Accuracy vs intersection ratio for 10^6 memtraces

The results achieved by k-NN under all conditions are shown in the Firure 6.2: x-axis is for feature set, y-axis is for n-gram size and z-axis is for accuracy. Labels on x-axis are named as *XXX_type* where *XXX* stands for feature number and *type* for memtrace sequence length: **h** for 100,000, **m** for 1,000,000 and **f** for 10,000,000. It is easy to see that growth of feature vector and memtrace sequence length (x-axis) generally results in accuracy growth. n-gram size and feature number increase results in accuracy growth, but now it is easier to see that area around n-gram size of 24 contains descending of accuracy (as well as weak matrix sparseness fading and area under class-wise frequency chart growth).

Worth to mention that there is visible correlation between intersection ratio and accuracy rate. The smaller ratio implies the bigger accuracy. This can be explained in the following way. Bigger intersection ratio means fewer features for feature selection. This results in smaller class-wise frequency of a particular feature, hence bigger sparseness of presence matrix. Sparse matrix can worsen generalization of dataset and even result in zero-filled rows, which definitely will

decrease accuracy. As we utilize bitmap of presence, and our feature selection method is aimed on selecting only class-unique features, zero-filled row means that particular sample could be difficult to correspond to one of the classes. And high sparseness of a matrix means that many features are not very efficient. So, it will lower ability of ML methods to generalize through the data. We conducted the study of class-wise frequencies of features selected in feature vectors, and found that there is no n-grams with frequency *1.0*. This means that either there does not exist a single n-gram that describes just malicious or just benign executables or n-grams with class-wise frequency *1.0* were rejected during feature selection as those present in both classes.

In the Figures 6.4, *a* and *b* Areas Under Feature Class-wise frequency charts (AUFC) are visualized for both malicious and benign executables. Having charts of this kind built, we can claim that there is a positive correlation between n-gram size and area. Another natural finding, is that the more features we have - the more samples we can cover with them, this brings us better accuracy. Also, we found, that AUFC for memtrace sequence length of 100,000 keeps growing on the n-gram size from 24 to 48, while for other lengths growth is almost stopped. We should notice that AUFC for malicious features are bigger than similar areas for benign features. This means that benign files are more different from each other in terms of n-grams than malicious ones. It can be explained as benign executables were made for bigger variety of purposes, while malicious are aimed on performing malicious *activity*.



(a) Benign class **(b)** Malicious class

Figure 6.4: Area under class-wise frequency chart for 200 features

From one point of view AUFC is a good measure for estimation of feature selection efficiency, but since ML algorithms work with presence bitmap, it is better to use another measure, which will directly show quality of extracted data. As it was said earlier, presence bitmap is a matrix of zeros and ones, so, if it has many zeros it will be hard to generalize data, hence ML algorithms will produce lower accuracy. Let's use *sparseness* as a measure of zeros percentage in matrix. Sparse-

ness is a ratio between numbers of zeros in matrix to number of cells and could be expressed as $sparseness = \frac{number\ \ of\ \ zeros}{width\ \ of\ \ matrix\ \cdot\ length\ \ of\ \ matrix}$. Moreover, we also performed sparseness calculations for all bitmaps. Using sum of benign and malicious AUFC and sparseness measures for all bitmaps, we built charts in Figure 6.5 for 800 features. As it can be seen from this chart, sparseness of presence bitmap is descending, while overall AUFC grows, proving earlier statement.

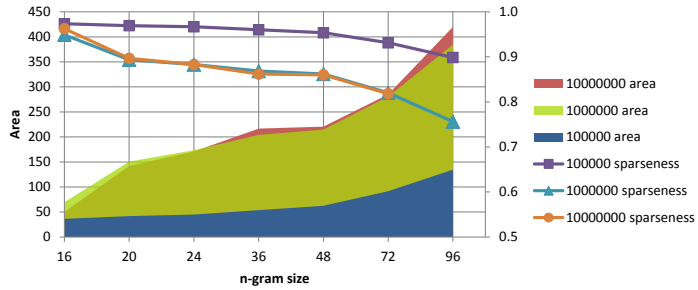


Figure 6.5: Area under class-wise frequency chart for 800 features

6.5 Discussions & Conclusion

This work targets malware detection using memory access patters based on a sequence of read and write operations, also called memtraces. From the literature we can see that many authors target dynamic malware analysis due to comprehensiveness of the collected behavioural features, including and not limited to disk, network and memory patterns. Yet, only few consider memory access operations as a reliable sources for malicious activities identifiers. We believe that memtraces can be highly relevant for proactive malware analysis. This is because it is the result of opcode execution, which produces consistent operations, yet may slightly vary for different arguments. We proposed a method for fast malware identification according to a presence or non-presence of a specific read and write pattern in its memory access sequence. For our experiments we used $10^5, \ldots, 10^7$ memtraces and 200-800 features extracted using $12, \ldots, 96$ n-gram size. It was found that 10^6 memtraces with 800 features and 96-grams gives a robust classification accuracy up to 98.92% using ML methods. In addition to this we studied a range of aspects and found that such method reveals a set of useful statistical properties that can be further applied for threat identification and ML-based malware detection. We believe that our work will contribute to proactive malware analysis in future.

6.6 Bibliography

[1] Najwa Aaraj, Anand Raghunathan, and Niraj K Jha. Dynamic binary instrumentation-based framework for malware defense. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 64–87. Springer, 2008.

[2] Gianni Amato. Peframe. https://github.com/guelfoweb/peframe. accessed: 27.10.2016.

[3] Dennis Distler and Charles Hornat. Malware analysis: An introduction. *SANS Institute InfoSec Reading Room*, pages 18–19, 2007.

[4] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577. ACM, 2009.

[5] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[6] Agner Fog. Technical university of denmark. *Instruction tables Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2016.

[7] VOLATILITY FOUNDATION. Volatility. http://www.volatilityfoundation.org/, 2015. accessed:2016-4-15.

[8] Lars Strande Grini, Andrii Shalaginov, and Katrin Franke. 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*, pages 337–350. Springer, 2018.

[9] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. The cuckoo sandbox.(2012). *URL https://www.cuckoosandbox.org*, 2012.

[10] IntelPin. A dynamic binary instrumentation tool, 2017.

[11] C. McMillan K. Kendall. Practical malware analysis. In *Black Hat Conference USA*, 2007.

[12] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2015.

[13] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[14] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. Wiley Publishing, 2010.

[15] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

[16] Bin Lu, Fenlin Liu, Xin Ge, Bin Liu, and Xiangyang Luo. A software birthmark based on dynamic opcode n-gram. In *Semantic Computing, 2007. ICSC 2007. International Conference on*, pages 37–44. IEEE, 2007.

[17] Netmarketshare. Desktop operating system market share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpcustomb=, 2016. accessed: 2017-22-11.

[18] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. On the trustworthiness of memory analysis #x2014;an empirical study from the perspective of binary execution. *IEEE Transactions on Dependable and Secure Computing*, 12(5):557–570, Sept 2015.

[19] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[20] Mike Schiffman. A brief history of malware obfuscation: Part 2 of 2, 2010.

[21] Dolly Uppal, Rakhi Sinha, Vishakha Mehra, and Vinesh Jain. Malware detection and classification based on extraction of api sequences. In *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 2337–2342. IEEE, 2014.

[22] P Vinod, Vijay Laxmi, and Manoj Singh Gaur. Reform: Relevant features for malware analysis. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 738–744. IEEE, 2012.

[23] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

[24] Jun Yang, Jiangdong Deng, Baojiang Cui, and Haifeng Jin. Research on the performance of mining packets of educational network for malware detection between pm and vm. In *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 296–300. IEEE, 2015.

Chapter 7

P2: Multinomial malware classification via low-level features

Sergii Banin, Geir Olav Dyrkolbotn

Abstract

Because malicious software or *("malware")* is so frequently used in a cyber crimes, malware detection and relevant research became a serious issue in the information security landscape. However, in order to have an appropriate defense and post-attack response however, malware must not only be detected, but also categorized according to its functionality. It comes as no surprise that more and more malware is now made with the intent to avoid detection and research mechanisms. Despite sophisticated obfuscation, encryption, and anti-debug techniques, it is impossible to avoid execution on hardware, so hardware *("low-level")* activity is a promising source of features. In this paper, we study the applicability of low-level features for multinomial malware classification. This research is a logical continuation of a previously published paper [4] where it was proved that memory access patterns can be successfully used for malware detection. In this research we use memory access patterns to distinguish between 10 malware families and 10 malware types. In the results, we show that our method works better for classifying malware into families than into types, and analyze our achievements in detail. With satisfying classification accuracy, we show that thorough feature selection can reduce data dimensionality by a magnitude of 3 without significant loss in classification performance.

Keywords: Information security, Malware detection, Malware classification,

Multinomial classification, Low-level features, Hardware activity

7.1 Introduction

Malware detection is an important part of information security. Recently there were several major cyber attacks that influenced power grids, banking and transportation systems, manufacturing facilities and so on [33], [41] and all of them used malware for achieving their final goals. Despite the use of anti-virus solutions, complicated anti-detection techniques allowed adversaries to avoid defense mechanisms. This fact points out a need for improvements in malware detection.

Malware is used for different purposes: to show ads to users, spread spam, track user activity, steal data, create backdoors and so on. Malware is often not created with a single specific purpose, but rather as a part of more advanced threats. APT or Advanced Persistent Threat is a human being or organization [11] that operates a campaign of intellectual property theft, the undermining of a company's or country's operations through stealthy, targeted, adaptive and data focused [7] attack techniques. *Something* has to exploit a victim's weaknesses, *something* has to aid in the installation of persistence tools, *something* has to communicate with command and control servers, and *something* has to perform actions in the victim system. Even though specific actions might be launched manually from the command and control server, they may rely on remote access trojans and backdoors [34] present in the victim system. As we can see, malware could be used for different purposes and goals.

Because of the variety of malware functionality, it is important not only to detect malice (*malware detection*), but to differentiate between different *kinds* of malware (*multinomial malware classification* or *malware classification*) 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.

Malware *classification* or *categorization* is a common problem that is analyzed in many research articles [40], [36]. There are two widely used malware categorization approaches: *malware types* and *malware families*. However, literature studies show that authors rarely provide proper definitions of these terms. This can lead to the various misunderstandings and non-valid comparisons. E.g. in [40], authors mention *viruses, backdoors, trojans* etc. while talking about classifying *malware types* and *families*. Another example of inconsistent terminology can be found in [36]. In this paper, authors claim that their system is capable of detecting the *malware families* (in their case *trojans, backdoors, worms*). Nevertheless, they compare their results to the results from other papers where research was done on the *malware types*. Authors of [35] attempted to elaborate on the definition of the term *malware*; however, later on they use term *malware family* when talking about

viruses, trojans, worms and other *malware types*. It might happen, that the use of inconsistent terminology is more common among academics and not malware analysis practitioners. Therefore, we must emphasize, in this paper that we use the following definitions created after reviewing descriptions of malware categories provided by well-known vendors (e.g. Microsoft, Symantec etc):

> **Malware type** is assigned according to general functionality.
> Malware is grouped into a **malware family** according to its particular functionality.

Where general functionality is about *what* malware does (which goals it pursues), and particular functionality is about *how* malware acts (which methods it uses in order to achieve its goals).

As it appears, it is insufficient to know that some malware is affecting operations: knowledge about its category (*family* or *type*) can aid in restoring a system's state as well as in developing new security mechanisms to prevent similar problems in the future. This necessitates standard definitions of different malware kinds and methods that allow the effective categorization of detected malware.

To avoid detection, malware creators develop additional evasive methods to thwart detection by antimalware software. They utilize various obfuscation techniques such as metamorphism, polymorphism, encryption, dead code insertions, and instruction substitution [37]. Such methods allow altering the appearance of a file and its static characteristics. The basic example is changing hash sums (such that SHA-1 or md5) used as file signatures by means of changing different strings in the file. Moreover, dead code insertions will change opcode sequences in the executable, making detection more difficult.

There are two main ways to perform malware analysis which are widely used and described in the literature [9, 20]: *static* and *dynamic*. *Static analysis* is performed without execution of a malicious file. The main purpose of this approach is to collect different static properties: bytes, opcodes and API n-grams frequencies, properties of Portable Executable header, strings (e.g. commandline commands, URLs etc) and others [37, 44]. *Dynamic analysis* is done by executing malware in a controlled environment (a virtual machine or emulator) and recording actions it has done in the system. These include patterns of a registry, network and disk usage, monitoring of API-calls, tracing of executed instructions, investigation of memory layout and so on [10]. Specialized sandboxes like Cuckoo [8] or other Virtual Machines can be used. They might be assisted by a debugger or other tracing software. Some authors assume [10, 30] disk and network activities are essential for malware detection, but few authors explored the capabilities of memory properties analysis [1, 21].

Though malware creators use a variety of sophisticated evasive techniques

[34], it is impossible to avoid execution on the system's hardware. Earlier low-level (or hardware) activity has proven to be efficient in malware detection [4]. In this paper, we use a similar technique for multinomial malware classification. Achieved results and findings will be used in future work, where combinations of high- and low-level activity will be used for malware categorization according to the specific context.

In this paper, we use sequences of memory access operations generated by a set of malicious executables as a source of features for machine learning algorithms. We apply dynamic analysis inside the virtualized environment as it is a safe (we don't let real malware samples spread outside of our environment) and time-efficient solution (experiments on physical machines would take significantly longer). We find the best features for distinguishing between ten predefined malware families and ten types. However, our models should be simple enough so that we can build a connection between low-level and high-level activity in the future work. Therefore, we may choose less accurate but simpler models to make analysis easier. Our initial hypothesis predicts that since malware types and families have a valuable difference in high-level behavior, we might be able to find distinctive low-level behavior patterns among malware categories. In the future work, we will test our models on the dataset of newer malware in order to check their capabilities against previously unknown (as for the models) malware. Our second hypothesis is that since malware families are assigned according to their particular functionality (e.g. exploiting of a certain vulnerability), they might generate more explicit activity that allows distinguishing better between families than between types.

The remainder of the paper is arranged in the following order: Section 7.2 contains State of the Art, Section 7.3 describes our methodology, Section 7.4 describes our results, Section 7.5 presents analysis of the results achieved, Section 7.6 presents a series of short remarks, conclusions, and a projection of future work.

7.2 State of the art

As was written above, in order to perform appropriate counteractions (to prevent) or postactions (to recover), we need additional information about malware category. With knowledge about malware types, we can apply appropriate defense mechanisms: e.g. in order to protect against Ransomware, we should keep an up-to-date backup of the data, while defense against self-replicating (Viruses) malware could be implemented with a thorough managing of a network traffic and removable media. In addition to knowledge about malware type, knowledge about malware family can help to set up appropriate defense mechanisms. Moreover, information about malware family can serve well in incident response actions: proper definition of malware family points to the potentially affected system components.

Many authors have performed research on malware classification. Different techniques and features are used to classify unknown malware into known malware categories or to detect outliers and perform a thorough analysis of such anomalies. For example, the authors of [23] combined different types of malware attributes (opcodes, API calls, flags, registers etc) in order to classify malware into 11 families. They used discriminant distance metric learning and pairwise graph matching in ensembled classifier to create an efficient framework that is capable of detecting previously unknown samples. Authors of [42] used a length of functions for classifying Trojans into 7 different families. They created pretty fast ($O(n)$ training and classification time) and relatively accurate (around 80% average accuracy) method for malware classification. They also warn, that their approach might not be as successful on other malware types such as Viruses, where malicious code is difficult to extract. The same authors in their newer paper [43] used API calls and their parameters as features for malware detection, and classification of 10 malware families. They managed to achieve up to 97% accuracy in malware detection, and up to 95% accuracy in malware classification.

Nevertheless, malware analysis always challenges. Authors of [5] did a thorough review of anti- debug, disassembly and VM techniques on the dataset of more than 4 million malware samples. As was shown, around 34% of malware is packed, while most of the packers listed in the paper contain some kind of anti-debug or anti-reverse engineering techniques. Moreover, among samples considered non-packed more than 68% contain obfuscation, 43% contain anti-debugging and 12% contain anti-disassembly techniques. This gives a clear view of a need of advanced dynamic analysis. However, more than 81% contain anti-VM techniques. The presence of anti-VM techniques might cause some problems for dynamic analysis. The authors didn't mention how they created their dataset and how the distribution of anti- techniques might be different from a real world. For example, Symantec published a paper where they claim that around 18% of malware stop execution when detected while being launched on a virtual machine [45]. Also, they say that a significant amount of organizations were planning to use server virtualization by the end of 2015. This means that malware may run on virtual machines or even created specifically to act on VMs and use their vulnerabilities [45]. Thus dynamic malware analysis, which is often performed in virtualised environments [13], is a relevant and promising research topic.

Dynamic malware analysis could be done on the different levels regarding to how "far" the features are from the hardware. For example, API calls or network analysis can be considered as high-level features and were proved to be reliable features for malware analysis [13]. On the other hand, memory activity [4] [1], opcodes [21], file system activity [22] and other hardware-based features[28] can be used for malware detection and considered as low-level features.

When studying memory access traces, we use Intel Pin [19], a binary instrumentation tool that allows us to capture detailed information about every single access to memory. Malware analysis usingf Intel Pin was described earlier in [1] and [4]. Authors of [1] tested model in a virtual environment and in a real environment with installed Windows XP or Xen Linux. They recorded the following features: API calls (both system or user) if any file or folder was modified, calls which created symbolic or hard links, calls and arguments passed to function *exec()*, and instructions that executed memory operations such as *read* and *write*. While in our paper we target separate memory access operations generated by separate opcodes, authors of [1] used *basic blocks* of a program. The basic block is a sequence of instructions executed between conditional branching instructions. Together with other properties of memory access operations in the basic blocks, authors studied memory range, the presence of certain operations and the size of transferred data. Using records of the execution trace, it was possible to create regular expressions and security policies, to use them for malware detection. Finally, 100% detection rate was achieved for both Windows and Linux on the original and obfuscated malware samples. Authors also state that their approach allows one accurately detect malware, and achieved 93.68% code and path coverage of input-dependent executables. Ensemble learning method for malware detection with a use of a number of properties extracted with Intel Pin [19] was proposed in the [21]. Authors used the frequency of opcode occurrence, presence of a particular opcode, difference between the frequency of opcode in malware and benign executables, distance and presence of memory references, and total number of load and store memory operations as well as branches. For each sample in their dataset, they recorded a generalized feature vector for every ten thousands of executed instructions, reaching up to 95.9% of classification accuracy.

Paper [4] is worth special attention, since the authors used memory access traces for malware detection. Their initial goal was to show that low-level features (memory access patterns in their case) are applicable for malware detection tasks. They used a virtualized environment and Intel Pin [19] to record memory access operations produced by malicious and benign executables. Using Machine Learning, they achieved more than 98% of accuracy for malicious against benign classification. Even though they used a different feature selection method, this work created a baseline for our research. However, the authors of [4] didn't take into account malware categories present in their dataset which points to one of the main goals of our research: testing whether memory access operations applicable for multinomial malware classification.

Additionaly, behavior analysis has its disadvantages: it might be vulnerable to anti-emulation, when malware is created with capabilities not to reveal it's functionality in emulated and virtual environments. Even though it might not be the

biggest problem, behavior based detection methods have another disadvantage: malware cannot be detected being executed [34]. The speed of detection depends on the features used for detection. E.g. if we use n-grams (or 1-grams) of API calls (or any other high-level event), our detection system will not make a decision before a certain API call is executed. However, single API call invokes the execution of many (rough analysis allows us to say hundreds) of opcodes with different parameters. So it is hypothetically possible to detect a needed high-level event before it is completed, since opcodes provide better data granularity. This is yet another reason to study low-level features in malware context. However, we need significantly more storage to store information about executed opcodes and their parameters than for API calls: from what was said above, it is easy to see that amount of data (to store and to analyze) can be larger in several magnitudes larger. In it's turn, memory access sequence takes less space and can be stored as a sequence of binary elements (*R* for *read* and *W* for *write* operation). It therefore simplifies process of pattern search and matching. Memory accesses potentially provide granularity better than opcodes: it is therefore possible to detect execution of opcodes sequence before it is finished. Moreover, since not all opcodes generate memory activity [4] this method should create smaller performance overhead while giving detection system more time to make a decision.

Taking into account the presence of anti-debug techniques mentioned above, as well the contiguous growth of virtualization solutions' market share [31] our research can aid for out-of-VM security solutions. Since many virtualized solutions might contain sensitive information, vendors won't always have access to the systems, while malware capable of escaping virtual environment can undermine not only host system [45], but other guest systems as well. Methods that allow monitoring the state of guest system from outside a virtual machine can improve the security of a virtualized environment without breaking ethical and privacy policies. In their paper [18], the authors designed and implemented a VMM-based hidden process detection system. Their system is placed outside of a protected virtual machine and interacts with a virtual machine manager. During the virtual machine introspection they inspect low-level state of protected virtual machine and track presence of hidden processes or lack of critical processes. In [15], authors created a system, that can detect OS type with a use of fingerprints extracted from virtual machine memory without false positives. Authors of [28] proposed Malware-Aware Processors, where they suggest hardware-based online monitoring of malware. As a features for malware detection they use frequency of memory read and write operations, memory address histogram, frequency and existence of opcodes and instruction categories. Moreover, hardware manufacturers tend to invest in hardware-based security solutions [28]. Because of everything written above, the results of our research may contribute to different aspects of digitized society,

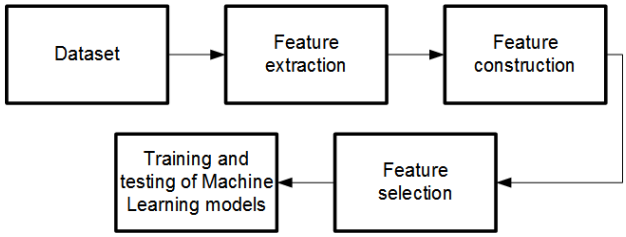


Figure 7.1: Simplified experimental flow

from improving the security of operations to helping security measures agree with ethical and privacy considerations.

7.3 Methodology

In this section we describe our experimental flow and explain details about dataset, feature selection, chosen machine learning methods, and hardware. We also outline several phases of analysis that we perform on the achieved results.

In our study we followed the scheme provided in Figure 7.1. We first created two datasets: one for malware families, and another for malware types. We then extracted features by recording memory access traces from each sample. Afterwards, we constructed n-grams of a size 96 for each sample. Lastly, we performed feature selection and trained Machine Learning Models. A detailed scheme of our experimental flow is shown in Figure 8.2 and described below in this Section.

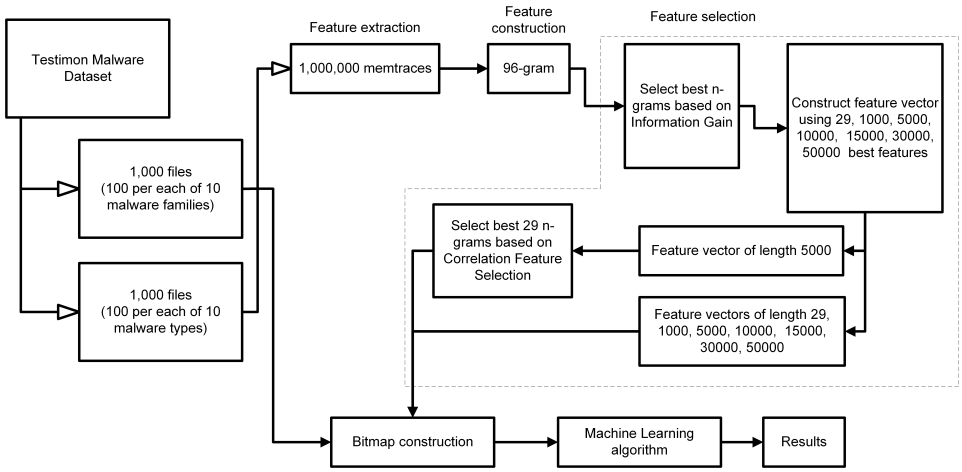


Figure 7.2: Detailed experimental flow

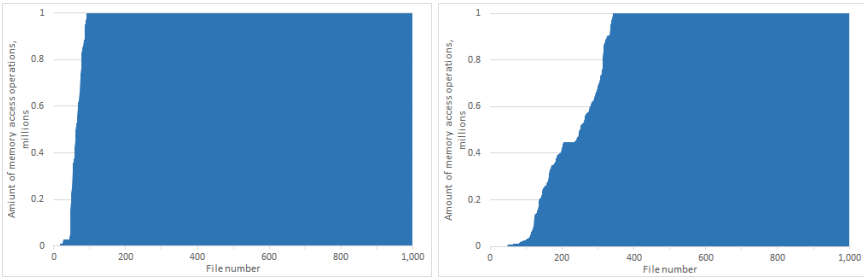
7.3.1 Dataset

The initial dataset was created under the initiative of the Testimon [14] research group and consisted of 400k malware samples. All malware samples were PE32 executables. This dataset was previously used for research purposes and described in more details in [38]. The malware that we used in our research was selected under the following criteria: the file should not be a DLL (only EXE files), it should not contain AntiDebug or AntiVM features, it contains GUI and files were sorted ascending according to a size of a file. Information about file type, AntiDebug, AntiVM and GUI were gained through the use of peframe [3]. As our research is aimed on proof of concept, dealing with DLLs and AntiDebug features was not a case, so we eliminated potential problems by filtering such things out (though we argue that study of AntiDebug influence will be an important part of future work). As we described in previous sections we can skip dealing with AntiVM, however we should remember this for assessing the results. We selected malicious files where *peframe* detected a presence of GUI for a simple reason: malware samples without GUI can fall into idle mode soon after starting, making it hard to collect enough data and increasing the time of dynamic analysis [4]. The presence of GUI should not significantly influence the results because it is present in every single sample. Because if something influences every sample we might assume, that results will be equally biased. We also decided to select small files because our goal was to prove a presence of features that can help to distinguish between 10 malware categories. If we used big files with long execution times it would be more likely to find a unique feature for each malware sample, which is good for classification accuracy, but won't contribute to understanding our findings and won't prove that our hypothesis works.

The main goal of this research is to check how memory access patterns can aid in malware classification. In order to do this, we created two datasets: the first contain 10 malware types and the second contain 10 malware families. We decided to choose the following malware types: *backdoor, pws, rogue, trojan, trojandownloader, trojandropper, trojanspy, virtool, virus, worm*. Additionaly we chose the following malware families: *agent, hupigon, obfuscator, onlinegames, renos, small, vb, vbinject, vundo, zlob*. The reason for such choice was that these types and families were prevalent in our malware dataset. We tried to create a balanced dataset, so each malware category contained around 100 samples. However, not all of the files launched, so they were rejected before analysis. Our datasets contained 952 files for malware types and 983 files for malware families. We can therefore assume that our datasets are approximately balanced, and we don't need to analyze the influence of sample distribution on the final results.

7.3.2 Feature construction and selection

The first task is to record a sequence of the first 1'000'000 (one million) memory access operations performed by an executable. We record only the type of operation: *W* for *Write*, and *R* for *Read*. This length of a sequence was chosen based on results from previous research [4] where it provided the best accuracy for malware against benign classification. We also found, that not all the executables can produce a greater or equal amount of memory access operations. On the Figure 7.3, the charts show the distribution of memory access operations gained from types and families datasets. We analyze all samples regardless the amount of memory access operations they produced. We do not truncate or fill missing operations with zeros: instead we work with the available amount of data. We explain our choice in the following paragraphs.



(a) Families **(b)** Types

Figure 7.3: Memory access operation numbers for families and types

As was stated in the Section 7.2 in some scenarios it might important to detect malicious process as fast as possible. Also in Section 7.1 we stated that our models should be simple enough to perform high level analysis of the findings in future work. So we need to find features that do not rely on how long the process is executed. The sequence of memory access operations is later on divided into overlapping n-grams of a length 96. An n-gram is a sub-sequence of length n of original sequence of length L. For example if an original sequence of length *L=6 [WRWWRW]* is divided into n-grams of length *n=4* (4-grams) then our n-grams set will look the following way: {WRWW,RWWR,WWRW}. Each n-gram starts from the second element of the previous one: they overlap on the n-1 elements as it is shown on the Figure 7.4.

This n-gram size was also chosen due to findings published in previous research [4]. We might notice, that out of 2^{96} possible n-grams we have to select the most relevant, thus significantly reduce feature space. N-grams are later stored for feature selection. Some malware researchers use file-wise frequency of features as

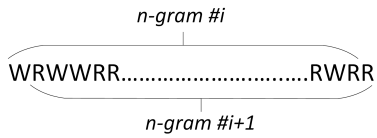


Figure 7.4: Example of overlapping n-grams

feature values: file-wise frequency is a ration between number of observations of a certain feature in the file and overall number of all observation of all features. In our case n-grams are stored without file-wise frequency for two reasons. First, we can not guarantee the amount of memory access operations (how long the file will run before stop) produced by a random file. Second, if we are able to find unique memory access patterns that comply with our classification goal, we can continue with more in-depth analysis of results, provide better high-level description of low-level findings.

As numbers of features are too big to just simply feed them to the machine learning models additional feature selection methods are therefore required. We obtained more than 15M of features for malware families dataset, and more than 6M of features for malware types dataset. The numbers are big but not surprising: sequence of memory access operations is basically a binary sequence with two possible elements *R* or *W*, so each sequence on 1M operations can potentially contain up to 1M-96+1 different 96-grams. However, during preliminary experiments we found that such amounts of data are too big to use in general-purpose machine learning libraries. Also models built on high-dimensional data provide results that are harder to interpret by human analysis. We used a feature selection method based on Information Gain. Information gain is an attribute quality measure based on class entropy and class conditional entropy given the value of attribute [24]. We ranked all features according to their Information Gain and selected 50000 with highest rank. We chose this number for several reasons. First, is computational complexity while training ML models, and second because we know from previous research [4], that we need around 400 features for class to get good classification accuracy. This reason is however more empirical since in this paper we study multinomial classification. After feature selection we use several conventional Machine Learning models in order to check our hypothesis, quality of feature selection and get some additional findings. To study the classification performance dependency on number of features we also selected best 5, 10, 15 and 30 thousands of features. We also performed correlation-based feature selection (CfSubsetEval [16] from Weka [17]) on the best 10000 features. This method selects features based on their correlation between class and other features. In simple words: the best feature is the one that correlates with classes and does not

correlate with other features (does not bring redundant information). This gave us the 29 best features, and we will show in Section 7.4 that they perform almost as good as thousands of features. However, it was impossible to get a larger number of features from correlation-based feature selection due to computational issues. On the Table 7.1 and on the Figure 7.5 we also present results for best 29 features selected by Information gain, as so we can compare feature selection performance on similar feature numbers. We omit results for feature numbers between 29 and 5000 to simplify presented material as they don't add any significant information to the reader.

7.3.3 Machine Learning algorithms

As a machine learning (ML) methods we chose the following: k-Nearest Neighbors (kNN), RandomForest (RF), Decision Trees (J48), Support Vector Machines (SVM), Naive Bayes (NB) and Artificial Neural Network (ANN). The following parameters (default for Weka package) were used for ML algorithms: kNN used *k=1*; RF had 100 random trees; J48 used pruning confidence of 0.25 and minimum split number of 2; SVM used radial basis as function of kernel; NB used 100 instances as preferred batch size; ANN used 500 epochs, learning rate 0.3 and a number of hidden neurons equal to half of the sum of a number of classes and a number of attributes. The results for ANN are shown only for the smallest amount of features since machine learning software Weka [17] was not able to finish training of such big neural networks. This fact can be explained by means of time complexity for training. According to [12] computational complexity of ANN is $\mathcal{O}(nMPNe)$ where *n* is a number of input variables (size of a feature set), *M* number of hidden neurons, *P* number of output values (10 in our case, since we have 10 classes), *N* number of samples, *e* number of learning epochs. Artificial Neural Networks built by Weka by default has 1 hidden layer, when the number of hidden neurons is taken as $M = (P + n)/2$ and *e* equals 500. For the dataset with 29 features, 10 classes, around 1000 samples and 5-fold cross validation it took $5 \times 9seconds \approx 45seconds$ to train models. The time complexity in this case is $\mathcal{O}(3 \cdot 10^8 operations)$.For example, for 5000 features the time complexity would be around $\mathcal{O}(6 \cdot 10^{13})$ what will take roughly 10^5 times more time to complete a task which is not suitable for our purposes since $45s \cdot 10^5 \approx 52days$ of training time.

We held our experiments on Virtual Dedicated Server (VDS) with Intel Core CPU running at 3.60GHz, 4 cores, SSD RAID storage and 48GB of virtual memory. As a main operating system Ubuntu 14.04 64bit was used. Additionally, MySQL 5.5, PHP 5.5.9 and VirtualBox 5.0.16 were used. Windows 7 32-bit was installed on the VirtualBox virtual machine as a guest OS. It is widely spread [27] and malware written for 32-bit OS's will run on 64-bit OS as well as well. We also

met some virtualization problems and were not able to run VM with 64-bit OS installed.

7.3.4 Analysis

During the analysis stage we try to explain achieved results in terms of numbers and words. We perform two types of analysis: statistical and context using sub-categories of our two datasets (different than original 10 classes). In statistical analysis we look into per-category classification accuracy and use statistical measures to explain differences in performance of machine learning models for different malware categories. During context analysis we are seeking an understanding of classification performance with a use of malware functionality description. As a results of analysis we not only understand how distribution of subcategories influence on per-category classification accuracy, but also show how human understandable explanation of malware functionality can contribute to an explanation of malware classification performance. With these findings we contribute to our future work where we are going to correspond low- and high-level activity and make results achieved with low-level features more understandable.

We also compare our results with results from a paper [38] where authors used similar malware categories but did static analysis and used different ML algorithms.

7.4 Results

In this section we present results and key finding of our experiments. In order to test the quality of our ML models we used 5-fold cross validation. As a classification quality measure, we use accuracy: it allows us to compare results in this paper with results from previous study published in [4]. It also shows how many instances have been correctly identified in our multinomial classification problem. In the Table 7.1 we present results for classification accuracy of different ML algorithms as a function of number of features. Each cell contains the accuracy that a certain ML method achieved with a given number of features. The last row shows accuracy that given ML method achieved with 29 features selected based on correlation [16]. We separated it from other feature sets as here we used different feature selection method.

The results are also presented in Figure 7.5. As we can see SVM and NB, in general, showed lower accuracy than other methods. This is interesting, since SVM performed pretty well in previous studies [4]. Additionally, in general, neither SVM nor NB improve their performance with an increased number of features. We can also see other ML methods (kNN, RF and J48) slightly improve their performance as number of features increase. However, using 50000 features instead of 5000 to gain a few extra percents of accuracy is not necessarily an effi-

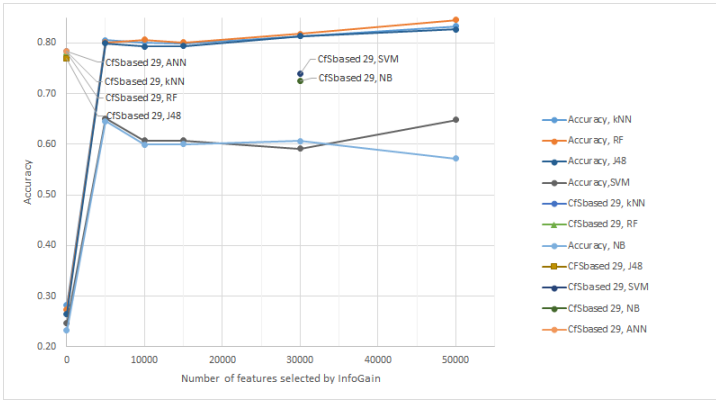
cient method, when our goal is to better understand how low-level features can be used for multinomial malware classification. Therefore we might put emphasis on a little bit less accurate but more understandable model. Because of this we decided to compare ML methods performance when only 29 features used. On the charts, these results are shown with separate points aligned to the most representative results on the horizontal axis. With 29 features (selected by corellation-based feature selection) given, ML methods such as kNN, RF and J48 show either small drop in performance or even some increase when compared to 5000 features. Other ML methods such as SVM and NB show significant grows of classification accuracy when given fewer features. One of the possible reasons could be that SVM is not originally designed for multinomial classification, it means that in order to deal with more than 2 classes it has to build several one-versus-all or one-versus-one classifiers. SVM is also known to have problems in so-called HDLSS datasets. High Dimension Low Sample Size dataset is a dataset, where the number of features is much bigger than the number of samples (it is true for most of our datasets, where number of samples is no bigger than 1000, while feature number starts from 5000). This fact was pointed out in different studies such as [2] and [26]. Naive Bayes classifier on its turn assumes that features are independent. But when we used Information Gain for feature selection, we can have a lot of potentially correlated features, thus Naive Bayes showed low classification performance in comparison to dataset where features where selected with respest to their mutual independence. Such behavior of Naive Bayes was studies in [32]. Also it is worth mentioning, that all the ML methods showed poor performance when 29 features selected by Information Gain were used.

	Accuracy for families						Accuracy for types					
Number of features	kNN	RF	J48	SVM	NB	ANN	kNN	RF	J48	SVM	NB	ANN
29	0.282	0.274	0.265	0.246	0.232	0.271	0.201	0.204	0.2	0.201	0.198	0.206
5000	0.806	0.802	0.800	0.651	0.646	N/A	0.642	0.637	0.623	0.468	0.430	N/A
10000	0.802	0.807	0.793	0.607	0.599	N/A	0.663	0.678	0.648	0.461	0.412	N/A
15000	0.800	0.802	0.795	0.607	0.600	N/A	0.665	0.661	0.645	0.455	0.415	N/A
30000	0.814	0.818	0.814	0.591	0.606	N/A	0.673	**0.688**	0.666	0.419	0.412	N/A
50000	0.833	**0.845**	0.827	0.648	0.572	N/A	0.668	0.675	0.665	0.375	0.386	N/A
CfSbased 29 features	**0.784**	0.781	0.769	0.740	0.724	0.783	**0.668**	**0.668**	0.626	0.584	0.498	0.617

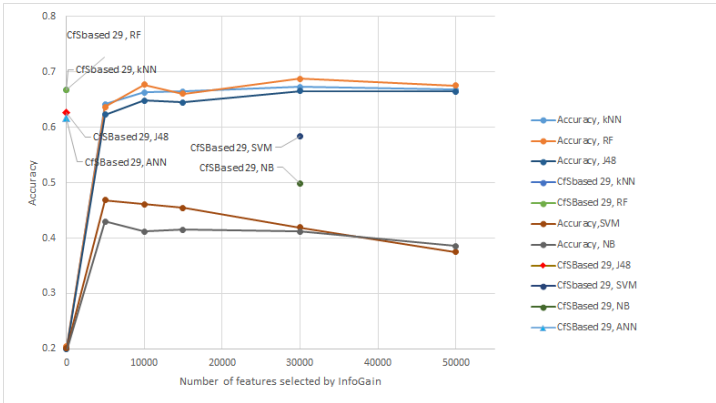
Table 7.1: Classification performance for families and types datasets

As we are able to see, classification performance are better when our algorithms are used for distinguishing between malware families and worse for malware types. We can explain this fact by referring to Section 7.1, where we provided definitions for term malware family and malware type. From the definitions it is easy to understand, that since malware sample is put into malware type according to general functionality it might be harder to distinguish between such categories, since a goal that malware achieves can be achieved by different methods. On the other

hand, malware families are about particular functionality, which means that methods used by samples within family should be more similar. This interpretation can be strengthened by the following observations: from malware families dataset we were able to extract more than 15 millions of uniques features, while types dataset gave us "only" 6 millions of such. It is worth mentioning that it took 1.66 hours (5987 seconds) to run through more than 15 millions of features extracted from malware samples divided in families, and select 50000 based on Information Gain. And it took 1.72 hours (6192 seconds) to run through more than 6 millions of features extracted from malware samples divided in types, and select 50000 based on Information Gain. In the next section 7.5 we will provide more analysis of the achieved results and describe some valuable findings.



(a) Families



(b) Types

Figure 7.5: Classification performance for families (a) and types (b) datasets

7.5 Analysis

In this section we analyze our findings. First, we analyze our results by means of statistics, e.g. we use our posterior knowledge of achieved accuracy and additional subcategories in order to explain why some malware categories are easier to classify than other. After we perform context analysis and try to show how human understandable description of malware categories can assist us in analyzing classification performance. Later we compare our results with results presented in [38] since authors used similar malware categories for their research.

7.5.1 Statistical analysis

For the analysis we will focus on the classification results from kNN algorithm. As it can be seen from Table 7.1 kNN provided best classification accuracy for both families and types when given 29 features selected with correlation-based [16] features selection. Also we chose this feature set for deeper analysis since following a rule of a thumb "less is more" we think that smaller feature set is much easier to analyze and it complies with our goals from Section 7.1. The question about a trade-off between model complexity and accuracy is not properly studied in the literature. However, the authors of [6] state that best models usually rely on a few features.

For our analysis we performed the following steps.

1. We recorded per-sample classification results and created a table where information about classification of each sample in our dataset is stored.

2. To this table we added a column where information about additional subcategories of each sample is stored. For samples in the malware types dataset, we added information about malware families, and vice versa.

3. As table from Step 1 allows us to calculate per-family (or per-type) classification accuracy we examined the influence of additional subcategories on the efficiency of our machine learning model to detect a certain malware type or family.

In the following paragraphs we describe our analysis workflow in a more detailed manner.

As we obtained our results based on average from a 5-fold cross validation, we decided to run 5-fold cross validation 5 times. Each cross-validation was done with a different random seed value in Weka [17]. This allowed us to make each sample to be in the test set (in other words - not used in model generating) more than once. We combined achieved results and took final information about classification result for a certain sample by the majority of results from all 5 runs.

In order to analyze the influence of additional subcategory on classification performance, we calculated entropy and coefficient of unalikeability of subcategories for each malware family or type. The (informational) entropy [39] is calculated as $H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$ where X is a variable (subcategory in this case), x_i is an i^{th} value of a variable, and $p(x_i)$ is a probability of a variable X to obtain value x_i. In simple words, entropy is often used to show randomness of a certain variable. It is also used in static malware analysis for detection of packers [25]. So in our case higher entropy will be a sign that certain category (type or family) are more diverse in terms of subcategories. In the matter of interest, we also used a coefficient of unalikeability [29]. It is an index of qualitative variation that measures variance of a nominal attribute (like our subcategory). It is calculated as $u = \frac{\sum_{i \neq j} c(x_i, x_j)}{n^2 - n}$ where $c(x_i, x_j) = \begin{cases} 1, & x_i \neq x_j \\ 0, & x_i = x_j \end{cases}$. It is a very simple coefficient, however it efficiently reflects variance of a nominal variable: if all the data are equal (variable obtain a certain value for all positions) than unalikeability is 1, and 0 if all positions are different. As we will show later unalikeability has a strong negative correlation with entropy.

After looking at results from Section 7.4 and taking into account our initial hypotheses our first guess was that the more subcategories are found within a specific class the more difficult it is to generalize over that class. However pure number of subcategories will not reflect their real distribution, and that was an important reason to introduce some more advanced measures described above. On the Tables 7.2a and 7.2b we present analysis of subcategory distribution on the classification accuracy.

As we can see, entropy in terms of subcategories in general is higher for malware types than for malware families. This can be easily explained by the fact that malware types samples are represented by higher number of subcategories. In order to illustrate findings from Tables 7.2 we will use charts on Figure 7.6. As we have written above, we can see a strong negative correlation between entropy and accuracy. However accuracy does not strongly dependent on neither unalikeability nor entropy. As we can see for both families and types datasets we can find classes with relatively high and low accuracy regardless the fact they share similar amount of subcategories and similar value entropy.

On the Tables 7.3a and 7.3b we show Pearson's correlation between corresponding columns in Tables 7.2a and 7.2b respectively. As we can see from these tables, accuracy is strongly affected by the number of subcategories or entropy in types dataset. This is yet another proof that families are assigned due to particular functionality, thus more alike within one family, and more diverse within several families. Table 7.3 also shows that entropy and coefficient of unalikeability has very strong (close to -1) correlation. However it is also worth to analyze several

class	acc.	unalike.	entropy	subN
agent	0.56	0.23	2.43	8
vbinject	0.59	0.98	0.08	2
obfuscator	0.64	0.98	0.08	2
hupigon	0.69	0.88	0.34	2
vb	0.75	0.36	1.83	8
small	0.84	0.73	0.92	7
vundo	0.88	0.94	0.22	3
renos	0.91	1.00	0.00	1
onlinega.	0.99	1.00	0.00	1
zlob	0.99	0.90	0.29	2

(a) Families

class	acc.	unalike.	entropy	subN
worm	0.43	0.02	5.69	63
pws	0.54	0.06	4.50	40
trojan	0.54	0.12	4.14	37
trojandr.	0.62	0.22	3.35	26
backdoor	0.67	0.11	4.18	40
trojanspy	0.71	0.27	2.92	22
trojando.	0.74	0.27	2.75	20
virtool	0.77	0.24	2.53	15
virus	0.81	0.02	5.42	55
rogue	0.86	0.31	2.08	9

(b) Types

Table 7.2: Accuracy (acc.), unalikeability (unalike.), entropy and number of subcategories (subN) for malware families (a) and types (b). Onlinega. stands for onlinegames, trojandr - for trojandropper, trojando. - for trojan-downloader.

specific cases in order to understand the nature of different classification accuracy for different malware categories.

7.5.2 Context analysis

In this subsection we analyze how human understandable description of additional subcategories influence classification performance. Within each (families and types) dataset we compare categories with high and low per-category accuracy by means of their two most frequent subcategories.

First, we will take a look at families dataset. For example malware family *zlob* has high classification accuracy (0.99) and relatively low entropy (0.29). It belongs

	acc.	unalike.	entropy	subN
acc.	1.0	0.43	-0.44	-0.37
unalike.	0.43	1.0	-1.00	-0.92
entropy	-0.44	-1.00	1.0	0.93
subN	-0.37	-0.92	0.93	1.0

(a) Families

	acc.	unalike.	entropy	subN
acc.	1.0	0.59	-0.60	-0.61
unalike.	0.59	1.0	-0.98	-0.96
entropy	-0.60	-0.98	1.0	0.99
subN	-0.61	-0.96	0.99	1.0

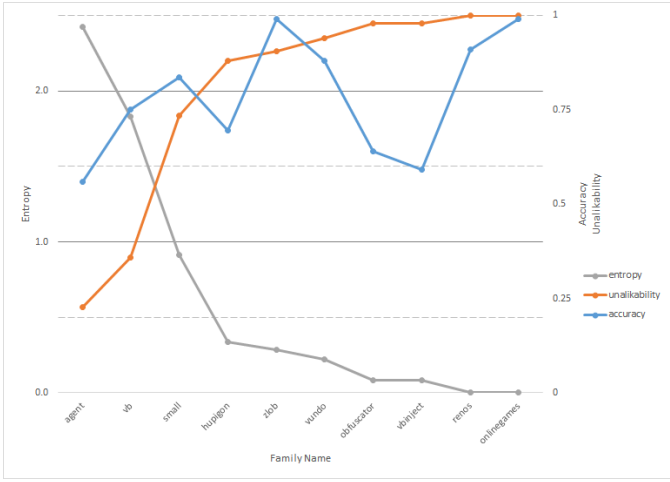
(b) Types

Table 7.3: Correlation between accuracy (acc.), unalikeability (unalike.), entropy and number of subcategories (subN) for columns of Tables 7.2a (a) and 7.2b (b)

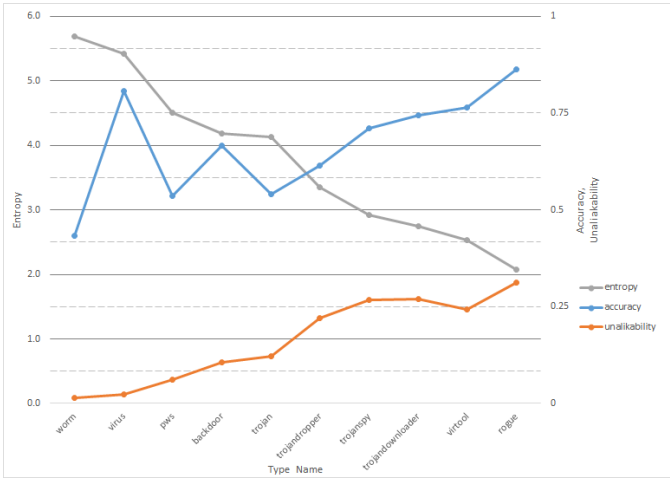
to two types (subcategories) such as *trojandownloader* and *trojan* with classwise frequencies of 0.95 and 0.05 respectively. However, another malware family *vbinject* has even lower entropy (0.08) but much lower accuracy (0.59). It also belongs to two types such as *virtool* and *trojan* with classwise frequencies of 0.99 and 0.01 respectively.

Second, let's take a look at types dataset. Malware type *virus* has relatively high entropy (5.42) and relatively high accuracy (0.81). Samples of this type belong to 55 different families (subcategories). And two most frequent are *small* and *radix* with classwise frequencies of 0.12 and 0.05 respectively. On its turn, malware type *rogue* has the highest accuracy of 0.86 with way lower entropy of 2.08. Samples of this type belongs to only 9 families, two most frequent of which are *fakexpa* and *internetantivirus* with classwise frequencies of 0.43 and 0.35 respectively. On the other hand, malware type *worm* has entropy (5.69) slightly higher than virus, but almost twice lower accuracy (0.43). Samples of worm type belongs to 63 families, two most frequent of which are *roram* and *kelvir* with classwise frequencies of 0.08 and 0.05 respectively.

In the first case we might admit, that trojandownloader and trojan families might have relatively similar behaviour, because first downloads and installs another malicious software, while others are trojans by itself. Yet they share a similar feature: they might look legitimate, and trick user to download and/or run them. On the other hand virtools are aimed on modification of other malicious software in order to hide them from antivirus software. At a first glance it might look like trojandownloaders and trojans are more similar than virtools and trojans. However



(a) Families



(b) Types

Figure 7.6: Per-family (a) and per-type (b) entropy (left vertical axis), unalikeability and accuracy (right vertical axis)

in both cases trojans make up only small amount of all samples. This is a very important finding and we will return to it later.

In the second case we should also study what our subcategories are. *Small* malware family are multipurpose malware, that is often used for downloading and executing additional files. They used in the initial infection of visitors to websites They also tend to drop and use kernel mode driver for its purposes. *Radix* on its turn is a mass-mailing malware that propagates by send a copy of itself via e-mail

with a use of its own SMTP engine. *FakeXPA* and *internetantivirus* are programs than pretend to scan systems for malware and display fake warning about malicious programs found on victim system. After that they ask you to pay for removing fake threats. There is no surprise that they have similar functionality and is a part of malware type *rogue*. *Roram* spreads via IRC channels. *Kelvir* also spreads via chat programs, but instead of IRC it uses MSN or Windows Messenger. To sum up this paragraph: *small* and *radix* are pretty different by functionality, while *fakexpa* and *internetantivirus* are way more similar. Yet malware type to which they belong are easy to generalize over. At the same time *roram* and *kelvir* are different only by the name of the chat program they use for proliferating. However, we might assume that our methodology is not capable of generalizing over such functionality.

7.5.3 Classification performance comparison

It is worth comparing our work with a paper by Shalaginov et al. [38] where the authors used malware dataset with a similar malware categories. It the Table 7.4 per-category True Positive and False Positive rates from [38] and our work are present. They did not include accuracy measure in their work, so we compare our results using True and False Positive rates. Authors of that work used a Neuro-Fuzzy approach for malware classification, while we will use results achieved by kNN because, as it was said earlier, it brought us best results in case of 29 features. As we can see, in most cases TP rate from our work is higher, and FP rate is somewhat lower than in [38]. However, for such categories as *hupigon* or *trojan* our results are worse. Authors of paper in interest used different features: they used static features from PE header. And Table 7.4 is yet more proof that static analysis may be outperformed by dynamic analysis. Specifically we can explain our relative success by the fact, that malware categories (both families and types) in use are assigned based on the *functionality* (dynamic characteristics), so our dynamic approach may be more suitable for such tasks. It is also worth mentioning that they had bigger dataset, so our results might be influenced as well if we increase number of samples. Authors of [38] used up to 20 features to complete their goals, what makes their work useful for malware analysts. This fact ensures us that using smaller (even though a bit less accurate) models gives better contribution for the scientific community. Finally, they show overall classification accuracy around 39.6% while ours is around 78.4%.

Shalaginov et al	Family	vb	hupigon	vundo	obfuscator	agent	renos	small	onlinegames	vbinject	zlob
	TP rate	0.3595	0.8080	0.5405	0.1222	0.1633	0.3276	0.5229	0.6084	0.2076	0.4295
	FP rate	0.0226	0.2033	0.0233	0.1185	0.0341	0.0101	0.1397	0.0303	0.0261	0.0262
	Type	trojan	pws	trojando.	worm	virtool	backdoor	virus	rogue	trojandr.	trojanspy
	TP rate	0.6084	0.1954	0.1385	0.1608	0.2112	0.3392	0.0857	0.0000	0.0744	0.0769
	FP rate	0.5220	0.0432	0.0517	0.0097	0.0614	0.1528	0.0098	0.0000	0.0193	0.0152
Our work	Family	vb	hupigon	vundo	obfuscator	agent	renos	small	onlinegames	vbinject	zlob
	TP rate	0.75	0.695	0.879	0.639	0.56	0.91	0.835	0.99	0.586	0.99
	FP rate	0.028	0.036	0.02	0.061	0.015	0.009	0.046	0	0.02	0.003
	Type	trojan	pws	trojando.	worm	virtool	backdoor	virus	rogue	trojandr.	trojanspy
	TP rate	0.542	0.535	0.745	0.433	0.765	0.667	0.808	0.864	0.615	0.71
	FP rate	0.046	0.046	0.026	0.049	0.033	0.057	0.038	0.02	0.026	0.029

Table 7.4: Comparison of our results to the results from [38]

As we shown in this Section, subcategories might be a key to explain classification performance of our malware classification approach, but only of the many. As we shown in previous paragraphs, subcategories does not directly influence classification accuracy neither by means of variety, nor by their amount. However analysis of subcategories pointed us to a very important finding: our approach is better in detecting and generalizing over a certain types of behavior, and worse for others. It means that in order to improve classification accuracy, in the future work we have to study how usage of context as ground truth will influence classification accuracy. We will elaborate on this more in the next section.

7.6 Conclusion and Future Work

In this paper we showed that patterns of memory access operations can be used for malware classification. We tested our method over the datasets with malware types and families. At a first glance, an achieved accuracy of 0.688 and 0.845 is not that high, however it is important to remember that in our case we have 10 classes and random guess on the balanced dataset will not exceed 0.1. So our results are way better than theoretical random guess generator. It is also important to notice that we went down from millions of potential features to 50000, and from them extracted 29 best features that allowed us to have compact yet relatively accurate models.

We have also shown that our approach performs better in some conditions, while worse in others. As we stated before, ground truth and context might help to improve classification accuracy. As a ground truth we might use high-level activity, to do so we should study what high-level activity (e.g. which API calls) are represented by certain memory access patterns. This study might also bring additional meaning to memory access sequences, because now a 96-gram similar to *WWWRWRW...WWRWW* does not say anything to a human analyst. While context analysis might involve capturing opcodes as well as the content of the memory. It can also be useful to use variative n-gram length, however, it might be extremely time and memory consuming. We also plan to evaluate models built with our ap-

proach against previously unkown, or assumed to be new, malware samples Nevertheless, our research topic is shown as promising, capable of bringing valuable findings and worth of further studies.

7.7 Bibliography

[1] Najwa Aaraj, Anand Raghunathan, and Niraj K Jha. Dynamic binary instrumentation-based framework for malware defense. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 64–87. Springer, 2008.

[2] Jeongyoun Ahn. High dimension, low sample size data analysis. 2006.

[3] Gianni Amato. Peframe. https://github.com/guelfoweb/peframe. accessed: 27.10.2016.

[4] Sergii Banin, Andrii Shalaginov, and Katrin Franke. Memory access patterns for malware detection. *Norsk informasjonssikkerhetskonferanse (NISK)*, pages 96–107, 2016.

[5] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*, 2012.

[6] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 122, 2012.

[7] Eric Cole. *Advanced persistent threat: understanding the danger and how to protect your organization*. Newnes, 2012.

[8] Cuckoo Sandbox. Cuckoo sandbox: automated malware analysis. https://www.cuckoosandbox.org/, 2015. accessed: 2016-4-15.

[9] Dennis Distler and Charles Hornat. Malware analysis: An introduction. *SANS Institute InfoSec Reading Room*, pages 18–19, 2007.

[10] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[11] What APT Means To Your Enterprise and Greg Hoglund. Advanced persistent threat.

[12] Scala for Machine Learning. Time complexity: Graph & machine learning algorithms. https://github.com/guelfoweb/peframe. accessed: 23.11.2017.

[13] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.

[14] Testimon Research Group. Testimon research group. https://testimon.ccis.no/, 2017.

[15] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. Os-sommelier: memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 5. ACM, 2012.

[16] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[18] Qiang Hua and Yang Zhang. Detecting malware and rootkit via memory forensics. In *Computer Science and Mechanical Automation (CSMA), 2015 International Conference on*, pages 92–96. IEEE, 2015.

[19] IntelPin. A dynamic binary instrumentation tool, 2017.

[20] C. McMillan K. Kendall. Practical malware analysis. In *Black Hat Conference USA*, 2007.

[21] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2015.

[22] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium*, pages 287–301, 2014.

[23] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

[24] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[25] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007.

[26] James Stephen Marron, Michael J Todd, and Jeongyoun Ahn. Distance-weighted discrimination. *Journal of the American Statistical Association*, 102(480):1267–1271, 2007.

[27] Netmarketshare. Desktop operating system market share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpcustomb=, 2016. accessed: 2017-22-11.

[28] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[29] Mike Perry and Gary Kader. Variation as unalikeability. *Teaching Statistics*, 27(2):58–60, 2005.

[30] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. On the trustworthiness of memory analysis #x2014;an empirical study from the perspective of binary execution. *IEEE Transactions on Dependable and Secure Computing*, 12(5):557–570, Sept 2015.

[31] PRNewswire. Virtual desktop infrastructure market to see 27.35% cagr driven by byod to 2020. http://www.prnewswire.com/news-releases/virtual-desktop-infrastructure-market-to-see-2735-cagr-driven-by-byod-to-2020-566513421.html, 2016. accessed: 2017-9-29.

[32] Jason D Rennie, Lawrence Shih, Jaime Teevan, and David R Karger. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 616–623, 2003.

[33] Reuters. Ukraine's power outage was a cyber attack: Ukrenergo. https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA, 2017.

[34] Ethan Rudd, Andras Rozsa, Manuel Gunther, and Terrance Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys & Tutorials*, 2017.

[35] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub. A survey on malware and malware detection systems. *International Journal of Computer Applications*, 67(16), 2013.

[36] V Sai Sathyanarayan, Pankaj Kohli, and Bezawada Bruhadeshwar. Signature generation and detection of malware families. In *Australasian Conference on Information Security and Privacy*, pages 336–349. Springer, 2008.

[37] Mike Schiffman. A brief history of malware obfuscation: Part 2 of 2, 2010.

[38] Andrii Shalaginov, Lars Strande Grini, and Katrin Franke. Understanding neuro-fuzzy on a class of multinomial malware detection problems. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 684–691. IEEE, 2016.

[39] Claude E Shannon. A mathematical theory of communication, part i, part ii. *Bell Syst. Tech. J.*, 27:623–656, 1948.

[40] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[41] The Verge. The petya ransomware is starting to look like a cyberattack in disguise. https://www.theverge.com/2017/6/28/15888632/petya-goldeneye-ransomware-cyberattack-ukraine-russia, 2017.

[42] Ronghua Tian, Lynn Margaret Batten, and SC Versteeg. Function length as a tool for malware classification. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 69–76. IEEE, 2008.

[43] Ronghua Tian, Rafiqul Islam, Lynn Batten, and Steve Versteeg. Differentiating malware from cleanware using behavioural analysis. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 23–30. IEEE, 2010.

[44] Dolly Uppal, Rakhi Sinha, Vishakha Mehra, and Vinesh Jain. Malware detection and classification based on extraction of api sequences. In *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 2337–2342. IEEE, 2014.

[45] Candid Wueest. Threats to virtual environments. *Symantec Research. Mountain View. Symantec*, 2014.

Chapter 8

P3: Correlating High- and Low-Level Features: Increased Understanding of Malware Classification

Sergii Banin, Geir Olav Dyrkolbotn

Abstract

Malware brings constant threats to the services and facilities used by modern society. In order to perform and improve anti-malware defense, there is a need for methods that are capable of malware categorization. As malware grouped into categories according to its functionality, dynamic malware analysis is a reliable source of features that are useful for malware classification. Different types of dynamic features are described in literature[4][3][12]. These features can be divided into two main groups: high-level features (API calls, File activity, Network activity, etc.) and low-level features (memory access patterns, high-performance counters, etc). Low-level features bring special interest for malware analysts: regardless of the anti-detection mechanisms used by malware, it is impossible to avoid execution on hardware. As hardware-based security solutions are constantly developed by hardware manufacturers and prototyped by researchers, research on low-level features used for malware analysis is a promising topic. The biggest problem with low-level features is that they don't bring much information to a human analyst. In this paper, we analyze potential correlation between the low- and high-level features used for malware classification. In particular, we analyze n-grams of memory access operations found in [4] and try to find their relation-

ship with n-grams of API calls. We also compare performance of API calls and memory access n-grams on the same dataset as used in [4]. In the end, we analyze their combined performance for malware classification and explain findings in the correlation between high- and low-level features.

Keywords: Malware analysis, Malware classification, Information security, Low-level features, Hardware-based features

8.1 Introduction

Malware, or malicious software, is one of the threats that modern digitized society faces every day. The use of malware ranges from showing ads to users, spreading spam and stealing of private data, to attacks on power grids, transportation and banking facilities[22][18]. The more severe consequences of malware use, the more likely they are a part of malicious campaign performed by an APT: Advanced Persistent Threat[7], an organization or a human that performs stealthy, adaptive, targeted and data focused [6] attack. APTs utilize different methods, tools and techniques to achieve their goals. Malware can be used at the different steps of APT kill-chain[4]: from reconnaissance and denial-of-service attacks to data stealing and creation of backdoors (for remote access) in the victim system. Since malware can be used for the variety of purposes, it is not only important to detect it, but also to be able to categorize it into different categories based on certain properties.

Malware classification (categorization) is an important step for understanding goals and methods of adversaries[1], analyzing security of systems and operations as well as for improving defense and security mechanisms. Static malware detection may fail due to obfuscation and encryption techniques used by the creators of malware. Because of this dynamic, or behavior-based detection methods are used. Moreover, malware samples are categorized into *types* and *families* by anti-virus vendors based on their behavior[4]. Hence, it is possible to assume that the use of features derived from *malware behavior* for malware classification can outperform static methods due to the nature of categories. Both static and dynamic methods need predefined sets of *features*: properties derived from a malicious file itself or its behavior.

We can divide features for dynamic analysis into two main groups: high-level (API and system calls, network activity, etc.) and low-level (memory access operation, opcodes, operations on hard-drive, etc). Generally speaking, we consider low-level features as those that directly emerge from the system's hardware[4] [13] [17]. Malware authors can try to conceal their malware and its behavior from anti-malware solutions and malware analysts by utilizing different techniques such as obfuscation, encryption, polymorphism or anti-debug. Despite their attempts, they can not avoid execution on the systems hardware[16][4]. That's why hardware-

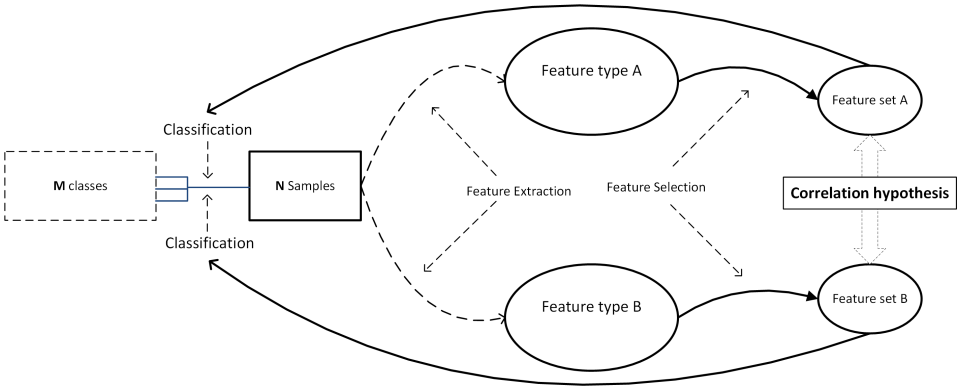


Figure 8.1: Generalized problem description

based, or low-level, features (since they are behavior features) are a reliable source of information for malware detection[5][17] and classification [4]. Different low-level features have been used for malware detection and classification: Hardware Performance Counters[3], frequencies of memory reads and writes[16],memory access patterns[5][4], architectural and micro-architectural events[21]. To the author's knowledge, there are no attempts to explain how particular low-level features correspond to high-level activity. Hardware-based features describe behavior of an executable on a very fine-grained level, so it is hard, by looking at the low-level feature itself, to explain which role in the behavior of an executable it has. Therefore, in this paper, we made an attempt to explain how memory access patterns correlate with the behavior of malware described by high-level features. This will make it easier for a human analyst to understand what exactly makes malware samples to be distinctive.

In order to describe our problem more generally we use an approach pictured on the Figure 8.1. Assume we have a dataset that contains N samples and the task is to classify them into M classes. From the dataset we can extract features of types A (e.g. low-level features) and B (e.g. high-level features). Different feature types are derived from different sources of information: different ways to describe properties of samples in the dataset. After feature selection, features of both types can be independently used for classification of samples from the dataset. Here we suggest a hypothesis that features from feature sets A and B can correlate with each other. In this paper, we focus on finding a correlation between n-grams of memory access operations and API calls. To address this problem we take paper [4] as baseline. In their paper authors used patterns of memory access operations to classify malware into 10 malware families and 10 malware types. The best 29 n-grams of memory access operations are selected, and we reuse them in our case since our datasets are identical. As the high-level, or human understandable, features we decided to use n-grams of API calls since they are shown to be a reliable

source of information for malware detection[2] and classification[12]. To get the most complete picture of possible correlations between memory access operations and API calls we need to search for *all-to-all* correlations. However, such an exhaustive search is computationally infeasible. In order to be able to carry out the search, we had to adjust the method, as described in Subsection 8.4.7. We record an execution flow of malware samples that contains memory access operations performed by single instructions as well as calls to the API functions (more details in Section 8.4). First, we perform classification and feature selection for n-grams of API calls. Our goal is not to study the performance of API calls for malware classification, but rather to find good and relatively short feature set of API calls n-grams as it will be more useful for research and analysis purposes. This feature set is later used in an attempt to find a possible relationship between memory access patterns and API calls, which existence or non-existence will help to reveal nature of memory access patterns that were successfully used for the same classification task.

The key findings of our paper are following. Our results show no significant correlation between information relevant for multinomial malware classification represented by best API-calls and best memory access patterns. This is important, as it shows that memory access patterns are not redundant to the higher level features such as API calls. As the result, feature set combined from memory access patterns and API calls show improved classification performance. This contributes to better malware detection and classification as well as to the potential hardware-based security solutions.

This paper is a proof of concept, and our main goal is to address challenges and possibility of a high-level explanation of low-level events as well as creation of a stepping stone towards an explanation of a performance of low-level features in malware classification context. The remainder of the paper is arranged as follows. In Section 8.2 we provide an overview of the related studies and focus on the baseline paper [4]. In Section 8.3 we describe our problem more specifically and describe an approach for validating of our hypothesis. In Section 8.4 we describe our experimental design, analysis environment, methods used for feature extraction and selection, explain how we search for correlation between features of different types as well as provide terms, definitions and assumptions important for our study. Finally, in Section 8.5 we present results, analyze them and provide conclusions in Section 8.6.

8.2 Background

In this section, we present a short overview of articles that are related to features and methods we use in this paper. There are many papers that use hardware-based features for malware detection or categorization. For example in [3] a real-

time dynamic malware detection with the use of special-purpose registers of modern CPUs as a source of features is proposed. Special-purpose registers, or hardware performance counters, are used for CPU scheduling, performance monitoring, integrity checking or workload pattern identification. In their paper, authors used four different events to construct features: retiring of a branch, load and store instructions as well as mispredicted branch instructions. With the use of various machine learning algorithms, they achieved up to 96% accuracy when classifying malicious and benign executables. Even though their dataset is small, consisting of only 20 benign and 11 malicious samples, their paper shows that hardware-based (or low-level) features can be used for malware detection.

In [16] Ozsoy et al. propose so-called malware-aware processors: processors that has a built-in hardware module that is capable of malware detection. In their work authors also mention hardware performance counters, but choose slightly different features to be used in malware detection: frequency of memory reads and writes, immediate and taken branches as well as unaligned memory accesses. They implemented a malware-aware processor in an FPGA emulator and state that their system is capable of malware detection with detection rates up to 94% and false positive rates of up to 7%. As they didn't achieve low-enough false positive rates, they propose to use malware-aware processor together with a software-based solution. They also emphasize the importance of malware-aware processor to be always on, so that it is hard to avoid detection from it.

Paper [5] is of particular interest for us, since it proposes a novel method for malware detection based on memory access pattern. In their work Banin et al. recorded sequences of memory access operations produced by malicious and benign executables. They didn't take into account addresses and values used by these operations but utilized only a type of operation: read or write. Each sample in their dataset was launched under surveillance of specially crafted Intel Pin [11] tool and was made to produce up to 10 millions of memory access operations. Later, larger sequences of memory access operations were split into a set of overlapping subsequences - n-grams of a size from 16 to 96. With the use of a feature selection and various machine learning algorithms they achieved a classification accuracy of up to 98%. Results showed, that 800 memory access n-grams are enough to achieve the highest accuracy on their dataset of 455 benign and 759 malicious executables. They claimed, that n-grams of memory access operations of a size 96 extracted from only the first million of memory access operations performed by executables are reliable features for malware against benign classification. Later, in [4] they evaluated performance of 96-grams derived from the first million of memory access operations for the malware classification task. They used two different datasets, one consisted of 952 malware samples and was label according to malware types while the other had 983 malicious executables that were labeled

according to malware families. With the use of feature selection, they compared results from feature sets of a size 50,000 and 29. Even though machine learning algorithms showed a decline in performance while given 29 features instead of 50000, this decline was only of a 5%. With only 29 features they achieved a classification accuracy of up to 78% for malware families and 66% for types. Even though it was far from the 98% from their previous paper they stated, that 78% can be considered good enough for 10-class classification problem. They also compared their results to the results from a paper [19], where authors used the same malware families and types but on the different dataset. In [19] Shalaginov et al. used static features, and achieved lower true positive and higher false positive rates. As we stated in the Section 8.1 we use paper [4] as a baseline: we use the same datasets, execution environment (Virtual Machine) and use their feature set as low-level features which origin we tend to explain. We will elaborate more on the similarities between our data collection processes in the Section 8.4.

Finally, we will look at some articles that make use of API-calls performed by malware during its execution for malware detection and classification. In their paper [12] Islam et al. used frequencies of occurrence of API calls during the execution of malware to detect malware and categorize into one of the 9 malware families. They also carve several static-based features such as lengths of functions or printable strings. Combining dynamic and static features they created so-called *integrated feature vector* and evaluated the classification performance of different features separately and together. They achieved a classification accuracy of up to 97% and showed that integrated feature vector can outperform other feature vectors. On its turn, Lim et al. in [15] proposed to use *k-grams* (special modification of n-grams derived from behavior automatons) of API-calls for malware detection. Even though authors didn't clearly picture the performance of their algorithm, they explained how small sequences of API calls can be used to measure the similarity between the behavior of different malware samples.

Shijo et al. [20] (similarly to [12]) utilized integrated feature vector constructed from dynamic and static features. As dynamic features, they used API calls n-grams of a size 3 and 4. With the use of only dynamic features they achieved a classification accuracy of up to 97% for malware against benign classification. Integrated feature vector allowed them to gain an increase in classification accuracy of up to 1%. The last paper we want to mention is [2] where Alazab et al. used API calls n-grams of a size 1 to 5 for malware detection. With the use of Support Vector Machines they achieved a classification accuracy of up to 96% and concluded, that for their dataset the best features were actually 1-grams or unigrams: n-grams of a size 1.

As we have seen, different high- and low-level features are used for malware detection and classification. Our goal in this paper is to find possible correlation

between memory access patterns (low-level features) and API calls (high-level features).

8.3 Problem description

From the literature overview, we can state that low-level features (despite difficulties with their extraction) can be a reliable source of information for malware detection and classification. However, system counters, opcodes and memory access patterns don't give much information about malware functionality to the security analyst. An n-gram of opcodes of a size 4, when given out of context, does not reveal what it was used for by itself. The same can be said about sequence of memory access operations: it is challenging to grasp which goals were achieved by malware when a certain sequence of memory access operations was performed. For example, a typical n-gram of a size 96 of memory access operations found in [4] looks like this: $WRWRRRRR...WWWRRRRRW$. It is obvious, that such features, even if they can be effectively used for malware classification, do not bring much useful information about malware's behavior. As different papers describe the use of low-level features for malware detection and classification, to the author's knowledge there have been no attempts to find a relationship between low-level activity and high-level events such as API-calls. Because of everything said above, first, we propose two following statements:

1. N-grams of memory access operations can be used for malware classification (shown in [4]).

2. N-grams of API calls can be used for malware classification (shown in e.g. [20]).

Based on statements 1 and 2 we propose the following hypothesis: if statements 1 and 2 are true, then it should be possible to find a correlation between some of the features from both feature spaces. An approach for validating this hypothesis is described in Subsection 8.4.7. For example, we assume that some memory access n-grams might originate in API call n-grams. If we are able to validate this hypothesis then we will find a way to correlate sequences of memory access operations to the events of higher level which are more human understandable. If our hypothesis is rejected, then API calls and memory access n-grams are independent features, thus combining them into an integrated feature vector should increase overall classification accuracy. Generally speaking, our goal is to check whether sequences of memory access operations that were successfully used for multinomial malware classification can be attributed to certain sequences of API calls, thus can be explained with high-level events and become more human understandable.

8.4 Experimental design

In this section, we present terms and definitions, provide the assumptions used and describe experimental setup and properties of datasets. Later on, we explain methods used for data collection and processing, list the machine learning and feature selection algorithms and describe the way we were searching for correlation between high- and low-level event.

8.4.1 Terms, definitions and assumptions

In this subsection, we provide terms and definitions and assumptions used during this study. We begin with the definitions:

- **N-gram**. An n-gram is a sub-sequence of length n of an original sequence of length L. For example if an original sequence of length L=6 [RRWRWW] is split into n-grams of length n=4(4-grams) then our n-grams set will be: RRWR,RWRW,WRWW[4]. In this example, similarly to baseline paper [4], and our paper we use overlapping n-grams: the next n-gram begins from the second element of the preceding one.

- **Memory access operations**: when an executable is *reading* from virtual memory, *read* (or *R*) memory operation is recorded. When *writing* to virtual memory performed by an executable, *write* (or *W*) memory operation is recorded.

- **API call**: or Application Programming Interface call is a call to a function provided by the operating system (Windows 7 in our case). API calls are usually used by malware and goodware to perform network, file, process and other kinds of activity.

- **Malware types and families**. Malware *types* and *families* are different ways to divide malware into categories. Malware *types* describe *general* functionality of malware: *what* it does, which *goals* it pursues. Malware *families* describe *particular* functionality of malware: which *methods* it use and *how* it pursues its goals [4]. For example, *virus, worm* and *backdoor* are malware types, while *hupigon, vundo* and *zlob* are malware families.

We continue with the following assumptions:

1. We assume that for the research and analytic purposes it is better to use smaller feature sets even if their performance in terms of classification accuracy is slightly lower [4]. For example, it is way easier to understand feature set of a size 33 that brings classification accuracy of around 70% than the one of a size 20000 with classification accuracy 73%.

2. We assume that if features from different sources (memory access operations and API calls) are related to each other, then this relationship can be found among small sets of the best features.

8.4.2 Experimental flow

In this subsection, we will describe our experimental flow. On the Figure 8.2 we picture a schematic view of our experiment. By running malware samples from two datasets (see Subsection 8.4.3), we collect data (memory access operations and API calls, Subsection 8.4.5) and perform feature construction (n-grams of API calls), later on, we use feature selection to reduce feature space and train machine learning algorithms in order to assess quality of a newly built feature vectors (Subsection 8.4.6). For the consistency (to the baseline paper) reasons, we run malware samples until they generate 1,000,000 of memory access operations. Some samples stop execution before they generate the desired amount of memory access operations, but we keep such data as is since this is a real-world scenario where one can't expect malware to produce as much traces as needed. While running malware, we record memory access operations and API calls (if present) for every executed instruction. From the literature review we understood, that API call n-grams of a size 3 and 4 are the most promising features. However, we also decided to use n-grams of length 8 in order to get a slightly more complete picture of API calls n-grams capabilities for malware classification. This also gives us more data to use in the search for correlation between memory access patterns and API calls. The number of n-grams is quite big, so in order to pursue one of our goals (shorter and more understandable feature set) we perform **feature selection** to reduce the dataset. As well as authors of [4] we used Correlation Based feature selection [9] from machine learning tool Weka [10] as it showed quite good performance while reducing the size of a feature set in several times of magnitude. After getting a reduced feature set, we store data in the format that can be used for training of machine learning models. In our case, similarly to the baseline paper, as feature values, we store only the fact of presence (1 or 0) of a certain feature in the behavior of a malicious sample. The logic here is similar to [4]: in contrast to other articles, where authors rely on frequencies of appearance of certain features, we want to find features that work regardless the time malicious executable has run. Our data looks like a **bitmap** of presence, where each row represents a single malicious sample, first column represents a category of a sample (family or type) and the rest of the columns represent features. Cells contain *1* if a certain feature is present in the behavior of malware and *0* if not. The bitmap of presence is later used for training the **machine learning** models (see Subsection 8.4.6), which classification performance (classification accuracy) is compared with the one from baseline paper. Having API call n-grams as features, we later search through the

entire records or behavior data from each malware sample in order to find whether these n-grams are related to the 29 memory access n-grams derived by authors of [4]. We elaborate on the search technique in the Subsection 8.4.7.

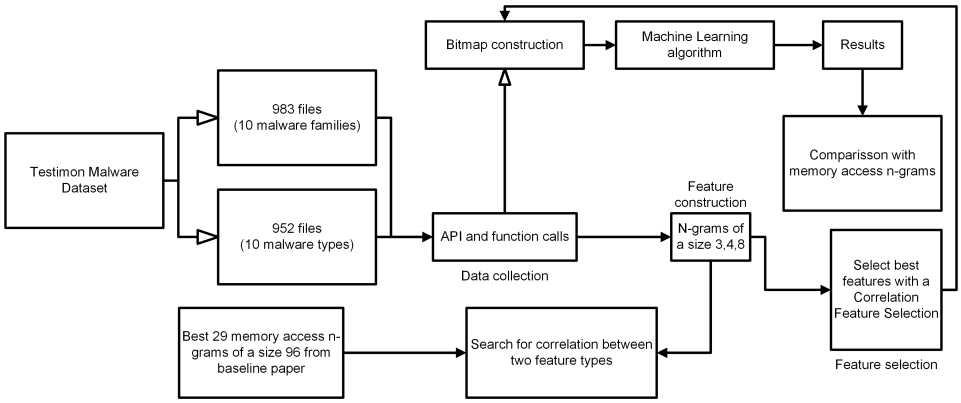


Figure 8.2: Detailed experimental flow

8.4.3 Dataset

Similarly to [4], our two datasets are derived from the original dataset collected under the initiative of Testimon [8] research group. It consists of 400k malware samples: malicious PE32 executables gathered from VirusShare[23]. Initial dataset was used for research purposes and is described in [19]. Both our datasets are the same as in baseline paper [4]. The authors of a baseline paper provide a detailed description of their datasets, while we focus only on the most important properties of these datasets. First of all, one dataset (952 files) has malware samples that are labeled according to ten types: *backdoor, pws, rogue, trojan, trojandownloader, trojandropper, trojanspy, virtool, virus, worm*. Secondly, another dataset (983 files) has its malware samples label according to ten families: *agent, hupigon, obfuscator, onlinegames, renos, small, vb, vbinject, vundo, zlob*. The choice of categories was made by the simple rule: 10 most prevalent categories in the original dataset were chosen. To simplify automated malware analysis (see Section 8.4.4) sample were chosen to be without anti-VM and anti-Debug features. As described in [4], dealing with anti-analysis functionality of malware is out of scope in such research, since their goal was to study a possibility of malware classification with memory access patterns as features. The distributions of categories within datasets are almost uniform, so we assume that datasets are nearly balanced, so there is no need to study the influence of categories distribution on the results of an assessment of machine learning models.

8.4.4 Analysis environment

Our analysis environment was almost identical to the one in [4], apart from different versions of host OS and VirtualBox. We assume that these changes will not influence the results of the experiments since hardware and guest OS are identical. We run our experiments on Virtual Dedicated Server (VDS) with Intel Core CPU running at 3.60GHz, 4 cores, SSD RAID storage and 32GB of virtual memory. As a main operating system, Ubuntu 18.04 64bit was used. Additionally, Intel Pin 3.6[11], Python 2.7 and VirtualBox 5.2.22 were used. Windows 7 32-bit was installed on the VirtualBox virtual machine as a guest OS. We used a virtual machine as an isolated environment to run malware together with a specially crafted Intel Pin tool. The virtual machine is reverted to the same snapshot before each run, so we avoid the influence of the environment on the results of data acquisition. To be consistent with a baseline paper, we choose the 32-bit version of Windows 7.

8.4.5 Data collection

We focus on "correlating" the n-grams of memory access operations with n-grams of API calls. We need to: a) record memory access operations produced by malware b) record calls to API functions. The first task is the easiest one. With the use of dynamic binary instrumentation framework Intel Pin, one can put instrumentation on each executed instruction and record memory access operations performed by it. For the consistency reasons, we chose the same amount of memory access operations to record as was used in [4]. A malicious executable run until it produces 1 million of memory access operations. As it was shown in the previously published papers, this is not only enough to reveal maliciousness of an executable [5] but also to perform multinomial classification of malware into categories and types[4]. The second task is more difficult. When a *call* instruction is performed it only contains an address of a function. In order to get its name from a library, one should find which one of the export symbols correspond to a certain address. Moreover, some native Windows libraries perform inter- and intra-modular calls not to the functions themselves (a call to a first instruction of a function) but to the subroutines within these functions. Most of the papers that use dynamic API call sequences do not describe how they treat such calls: it is not clear whether they record or just ignore them. In this paper, we treat a call to a first instruction of an API function and a call to a subroutine in an API function equally. Our reason for this is that if a logic of an executable requires such calls to be done and we can collect this information, it may improve the understanding of malware's current execution goals and context.

The call instruction can be used to invoke internal (to an executable itself) function. It is usually impossible to derive a name of an internal function of an executable (unless you have debug file, which is not the case in malware analysis),

so we store a name of a section where a function of interest is placed. We also keep this information and treat such calls equally to the API calls. Having raw data recorded, we split a sequence of API calls generated by each malicious sample into **n-grams**.

For better analysis capabilities as well as future work we record additional information for *each* instruction executed after launching a malware sample. A real example of raw data is present in the Listing 2 in Appendix A. In order to record this data, we created an Intel Pin based tool that is launched together with each sample from a dataset. A tool records all data into a file and stops if an executable generated 1 million of memory access operations. Some samples generate less memory activity than others, but we consider it a real-world scenario where one can't rely on malware to generate a particular amount of data.

From the raw data we extract names of the called functions, store them into the sequence according to their execution order and split the sequence into n-grams of a different size. For example, one of the API call n-grams of a size 4 derived from malware families dataset looks as following: *memset, GetModuleHandleW, ferror, _freea*. From the raw data, we extract API calls and memory access operations, that are later used in training the machine learning models and searching for mutual correlation.

8.4.6 Machine learning algorithms and feature selection

For the consistency reasons, we chose the same machine learning (ML) algorithms as in [4]: k-Nearest Neighbors (kNN), RandomForest (RF), Decision Trees (J48), Support Vector Machines (SVM), Naive Bayes (NB) and Artificial Neural Network (ANN). The following parameters (default for Weka[10] package) were used for ML algorithms: kNN used *k=1*; RF had 100 random trees; J48 used pruning confidence of 0.25 and a minimum split number of 2; SVM used radial basis as function of kernel; NB used 100 instances as the preferred batch size; ANN used 500 epochs, learning rate 0.3 and a number of hidden neurons equal to half of the sum of a number of classes and a number of attributes. In order to assess the quality of machine leraning models we used 5-fold cross validation, and chose accuracy (number of correclty classified instances) as the measure of evaluation. To reduce the feature set we used Correlation Based feature selection from Weka. Correlation-based feature selection [9] is an algorithm that chooses a subset of features that have the highest correlation with classes, lowest correlation with each other and give the best merit among other possible subsets. First reason to choose this feature selection method as it helped authors of a baseline paper to go from 50 thousands of features to just 29, so we wanted to get a number of features of nearly the same magnitude. Second reason is that one of our goals is to have relatively short feature set that can be easily analyzed by a human analyst.

8.4.7 Correlating features derived from different sources

In this section, we present a method to validate our hypothesis presented in Section 8.3. There are several approaches that can be used to validate our hypothesis. The first one is the most obvious: create the *entire* feature sets for memory access operations and API calls n-grams and find correlations between them (*all-to-all* approach). This approach will reveal the full picture of correlation between the two feature types. But it also has one major drawback, that makes its use almost impossible. The entire feature space of memory access n-grams in [4] consists of 15 millions distinctive features for malware families dataset. Finding their correlation with around 12 thousands of API calls 3-grams (see Subsection 8.5.1) can not be finished in feasible time. Slightly less time consuming variant is to search for correlation between the best memory access operations features and the entire feature space of API calls n-grams (*best-to-all* approach). This method would provide a less complete overview over the possible correlations, but would still be very time consuming, and is left for the future work. To some initial results we used a *best-to-best* approach: instead of taking the entire feature sets of memory access operations and API calls, we use only the best features out of both feature spaces. This approach allowed us to finish the experiments in feasible time, but also has some limitations that will be discussed in the following sections. As this paper is aiming to provide a proof of concept for searching for correlations, we believe that this approach properly fits our purposes.

One of the challenges we met during this research is how to correlate a certain n-gram of memory access operations to an n-gram (n-grams) of API calls. First of all, we need to locate a place in a raw data, where a certain n-gram of memory access operations is found. To do this, we iterate over the raw data, collect memory access operation into a buffer of a size 96 (see Section 8.2) and check if the pattern in the buffer is found among one of the features taken from the baseline paper. If match occurs - we save the position where memory n-grams starts and begin the search for API call n-gram. There can be various approaches to this and we selected the following one, as it brings wider coverage of execution flow. To state that a certain memory access n-gram is related to an API call n-gram we use the following criteria:

1. If the beginning of memory access n-gram lays after first call in API calls n-gram and before the call that follows current n-gram - these memory and API call n-grams correlate. In this case we assume that memory access n-gram is correlated to an API calls n-gram.

2. For any other case we state, that memory access and API call n-grams are not correlated.

The above mentioned criteria works as shown in Figure 8.3 where we present a simplified version of our data. On this Figure, memory access n-gram of a size 96 correlates with API calls 3-grams [APIcall_1, APIcall_2, APIcall_3], [APIcall_2, APIcall_3, APIcall_4] and [APIcall_3, APIcall_4, APIcall_5] but does not correlate with [APIcall_4, APIcall_5, APIcall_6]

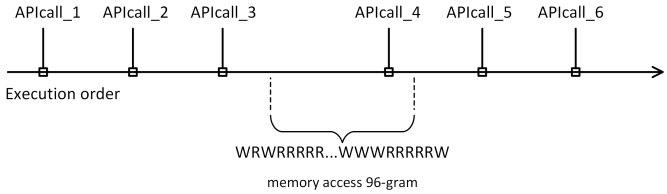


Figure 8.3: Correlation between API calls and memory access n-grams

8.5 Results and analysis

In this section we provide the results of feature selection and classification for API calls n-grams, compare them to the results achieved with memory access n-grams from [4] and evaluate our findings in correlating these two types of features.

8.5.1 API call n-grams for malware classification

From the raw captured data we extracted 12818 3-grams, 17407 4-grams and 33900 8-grams in the malware family dataset and 17252 3-grams, 24054 4-grams and 49513 8-grams in the malware types dataset. Using correlation based feature selection allowed us to reduce number of features to the following: 23 3-grams, 33 4-grams and 47 8-grams in the malware family dataset and 52 3-grams, 62 4-grams and 76 8-grams in the malware types dataset. The reduction of feature vectors worked similarly to the baseline paper: we went down from *tens of thousands* to *less than hundred* features. As assessment of classification performance of API call n-grams is not the main goal of this paper, we provide only the results for reduced feature sets. However, we performed classification on the full feature sets and their classification accuracy was only a few percents higher then in reduced feature sets. It is again similar to [4], so we assume that it is possible to compare newly acquired feature set with the one from [4]. In the Table 8.1 the classification accuracy achieved by different machine learning algorithms is presented. On the left and right sides of the table we present the results achieved on malware families and malware types datasets respectively. First row represent results achieved with n-grams of memory access operations of a size 96 from [4]. We name this feature type *Mem96*. Rows from 2 to 4 represent results achieved with API calls n-grams of sizes 3,4, and 8. We name them *API3,API4* and *API8* respectively. As we can see, most of the time API calls n-grams performed on the

same or even higher level then memory acess n-grams for the malware families dataset. In contrast, performance of API calls n-grams for malware types dataset most of the time was lower then the one by memory access n-grams. These results help us to prove Statement 2 from Section 8.3. In the Table 8.1 we use bold font in order to underline best classification accuracy for a certain type of features. It is also worth mentioning, that in general API calls n-grams of a size 4 performed better then other types of n-grams. We have to draw an important conclusion from the results we achieved with API calls n-grams. Classification performance of less then a hundred API calls n-grams are comparable to those achieved with tens of thousands of memory access n-grams in [4].

Table 8.1: Classification accuracy for baseline feature set, API call n-grams feature sets and combined feature sets.

		Feature set size		Families						Types					
#	Feature type	Fam.	Typ.	kNN	RF	J48	SVM	NB	ANN	kNN	RF	J48	SVM	NB	ANN
1	Mem96	29	29	**0.784**	0.781	0.769	0.740	0.724	**0.784**	**0.668**	**0.668**	0.626	0.584	0.498	0.617
2	API3	23	36	0.775	0.780	0.746	0.709	0.652	0.774	0.616	0.631	0.587	0.533	0.521	0.607
3	API4	33	46	**0.813**	0.810	0.792	0.765	0.677	0.805	0.636	0.636	0.604	0.541	0.566	0.616
4	API8	47	67	0.799	0.801	0.784	0.751	0.694	0.797	0.643	**0.660**	0.605	0.537	0.562	0.615
5	API3+Mem96	52	65	0.834	0.856	0.817	0.781	0.711	0.845	0.680	**0.700**	0.641	0.573	0.556	0.682
6	API4+Mem96	62	75	0.838	**0.859**	0.824	0.786	0.716	0.842	0.680	0.694	0.662	0.580	0.566	0.676
7	API8+Mem96	76	96	0.832	0.845	0.801	0.773	0.717	0.835	0.667	0.687	0.649	0.586	0.575	0.686

8.5.2 Correlating memory access and API call n-grams

The results we got were quite surprising. With the feature selection, we used and feature correlation search method we described in Subsection 8.4.7 we found no correlation between memory access n-grams and API call n-grams for malware *types* dataset. For malware *types* dataset our hypothesis about the correlation between features derived from different sources was rejected. Results for malware *families* dataset was not much different. One memory access n-gram was found to be related to a certain API calls 3-gram in different malicious samples, and the other was found to be related to two API calls 4-grams in different malicious samples as shown in Listing 8.1. So our initial hypothesis was mostly rejected for malware *families* dataset as well. Having this information we decided to create integrated feature sets by combining memory n-grams feature set with API call n-grams feature sets. We analyze the performance of an integrated feature set in the next subsection.

8.5.3 Performance of integrated feature sets

We found an idea about combining features of different types into an integrated feature vector from [15]. In the Table 8.1 we present classification accuracy achieved with integrated feature vectors. In the rows 5 to 7 results of combining memory n-grams feature vector with all API call n-gram feature vectors are

present. As we can see, with several exceptions, most of the time integrated feature vector outperform separate feature vectors. Moreover, with an integrated feature vector (which size didn't exceed 100) we achieved a classification accuracy of 85.9% for families and 70% for types, which are higher than respective 84.5% and 66.8% achieved in [4] with use of 50,000 memory access n-grams. This indicates that combining API call and memory access n-grams does not bring redundant information which often results in lower classification accuracy[14]. Even though our hypothesis was rejected, the increased classification accuracy of an integrated feature set show that our correlation search method (Subsection 8.4.7) was correct.

8.5.4 Discussion and analysis of correlation findings

As we already said, for the two entire datasets, we found only two memory access n-grams that we found to be related to the API call n-grams from a reduced feature set. In the Listing 8.1 we show found relationships of memory access n-grams and API call n-grams. As we can see, for our family dataset, a memory access n-gram is related to only one API call 3-gram. However, the relationship between memory access n-gram and API 4-grams can look a little bit more complicated. We found that the same memory access n-gram can originate from different API call n-grams. But this can be easily explained after analysis of the API 4-grams themselves. As we can see, these two API call 4-grams can easily overlap: last three API calls of the first 4-gram can be the first three API calls in the second 4-gram. And as we described in Subsection 8.4.7 if the beginning of memory access n-gram lays between first and last call in the API call n-gram - these memory and API call n-grams are related. So it is easy to understand now, that if selected API call n-grams are overlapping - the same memory access n-gram can originate from both of them.

```
Memory ngram WWWRRRRRRRWRRWRRWWWWRRWWWWRWRRRRRWRRWWRWWR
    ↪ RRWWWRRRWWRRRRWRWRRRRRRWRRRRRRRRRRRRWRWWRWWWWWRRRWRRRWW
is related to the following API call 3-gram: RtlTryEnterCriticalSection,
    ↪ RtlLeaveCriticalSection,memset

Memory ngram WRWRRRRRRWRRWRWRRRWRRRRRRWRRRWRWRRRRRRWRRR
    ↪ RRRRRWWRWRRRWWWRRWWRWRRRRRWRWWRWRWWRRWWRWWRWWWWWRRRRRW
is related to the following API call 4-grams: RtlEnterCriticalSection,
    ↪ RtlEnterCriticalSection,RtlEnterCriticalSection,
    ↪ RtlCompareUnicodeStrings
and RtlEnterCriticalSection,RtlEnterCriticalSection,
    ↪ RtlCompareUnicodeStrings,RtlCompareUnicodeStrings
```

Listing 8.1: Memory access n-grams and correlated API call n-grams from malware families dataset

As a way to improve our search technique it is possible to split a sequence of API calls into *non-overlapping* n-grams. However, in some rare cases, it might result in several memory trace n-grams to be related to a single API call n-gram. Another reason of small correlation findings can be a *best-to-best* approach that we chose

for correlation search. Utilizing a *best-to-all* approach together with an in-depth explanation of correlated API calls n-grams is one of the priority goals for the future work.

There is one thing that is important to look at after presenting relatively poor correlation findings. As we have written above, we trace the execution of malware samples until they generate 1 million of memory access operations. Some samples produce less than the expected number of memory access operations. It is important to understand, that the execution of PE file does not start from the main module of a file. Instead, different API calls are invoked by an operating system (they still executed under the process of malware, so we trace them anyway) in order to prepare an execution environment. The amount and type of calls performed before execution of main logic (main module) of a malware depends on the way an executable was compiled and the resources it needs for execution. It is important to notice, that even if an API call is made from the main module of an executable, its instructions will be corresponded to the external module(e.g. ntdll.dll). To go deeper into this problem first we counted the number of instructions executed by malware from its main module and divided it by the total number of instructions in the trace. Amount of instructions performed from the main module (defined by the malware directly) ranges from 0% to 99.9% with an average of around 20%. It means that some samples didn't even reach to the execution of their main module. From first glance, it should have led to the sample being indistinguishable from each other. Nevertheless, as we already said, this platform-specific (PE is an executable format used in Windows) preamble depends on the properties of the file. Another thing that we checked was the percentage of *call* instructions executed from the main module. These numbers range from 0% to 8% with an average of up to 1.5%. From what was said above, and from additional data analysis, it is possible to draw the following conclusion: most of the API calls in our experiments didn't originate from the main modules of executables. Moreover, as the number of instructions performed from the main modules is relatively low, the memory access n-grams from [4] also did not originate from main modules either. The first conclusion that can be drawn from this is that some malicious executables can be categorized into families and types (with an accuracy we achieved) based on the activity they produce before executing their main logic. On the first hand, these are very promising results since detection mechanisms based on the features used in this paper can potentially detect malware before anything malicious is done. However, we didn't study what changes to our victim system our malicious samples did. So this is clearly a question for future research. On the other hand we might have actually detected malicious behavior by itself: there are known malware samples that achieve its goals from TLS callbacks or by inserting malicious code into legitimate DLLs or executables (other than malware's main modules) and performing

direct jumps or calls to the infected parts of legitimate DLL's or executables.

As a final remark to this subsection we suggest the following solution to the questions we outlined in the beginning. To understand if API calls that actually produce memory access patterns from [4] can be useful for malware classification we have to use only a certain amount of API calls made around a place from where memory access n-gram is originated from. Based on these API call sequences we may try to find features that are relevant for malware classification. This is planned to be done in the future work, as the amount of "API calls made around a place from where memory access n-gram is originated from" has to be found after a number of experiments. Also, the type of features in this future case has to be discussed as well.

8.6 Conclusions

In this paper, we examined the nature of memory access n-grams that were successfully used for malware classification by authors of [4]. We also attempted to understand the relationship between those low-level features and high-level activity patterns such as API call n-grams. Our findings showed no significant correlation between the best n-grams of memory access operations and the best n-grams of API calls (at least under our experimental design). We also showed that API calls n-grams can be used for malware classification on the dataset from [4] and found that combining features derived from different sources (low- and high-level activity) can bring improvement in classification accuracy. While analyzing our data we concluded, that both low- and high-level features used in our experiments often have their origin outside of the main module of an executable. This paper brings important findings and outlines the direction of future research about the use of low-level features in malware analysis.

8.7 Bibliography

[1] Types of malware. https://usa.kaspersky.com/resource-center/threats/types-of-malware. accessed: 17.03.2019.

[2] Manoun Alazab, Robert Layton, Sitalakshmi Venkataraman, and Paul Watters. Malware detection based on structural and behavioural features of api calls. 2010.

[3] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 703–708. IEEE, 2014.

[4] Sergii Banin and Geir Olav Dyrkolbotn. Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107–S117, 2018.

[5] Sergii Banin, Andrii Shalaginov, and Katrin Franke. Memory access patterns for malware detection. *Norsk informasjonssikkerhetskonferanse (NISK)*, pages 96–107, 2016.

[6] Eric Cole. *Advanced persistent threat: understanding the danger and how to protect your organization*. Newnes, 2012.

[7] What APT Means To Your Enterprise and Greg Hoglund. Advanced persistent threat.

[8] Testimon Research Group. Testimon research group. https://testimon.ccis.no/, 2017.

[9] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[11] IntelPin. A dynamic binary instrumentation tool, 2017.

[12] Rafiqul Islam, Ronghua Tian, Lynn M Batten, and Steve Versteeg. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2):646–656, 2013.

[13] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2015.

[14] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[15] Hyun-il Lim. Detecting malicious behaviors of software through analysis of api sequence k-grams i. 2016.

[16] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661. IEEE, 2015.

[17] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[18] Reuters. Ukraine's power outage was a cyber attack: Ukrenergo. https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA, 2017.

[19] Andrii Shalaginov, Lars Strande Grini, and Katrin Franke. Understanding neuro-fuzzy on a class of multinomial malware detection problems. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 684–691. IEEE, 2016.

[20] PV Shijo and A Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015.

[21] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109–129. Springer, 2014.

[22] The Verge. The petya ransomware is starting to look like a cyberattack in disguise. https://www.theverge.com/2017/6/28/15888632/petya-goldeneye-ransomware-cyberattack-ukraine-russia, 2017.

[23] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

Appendix A Raw data sample

In this Appendix we present a sample of a raw data gather during our experiments. We also explain each field included in the data.

1. Opcode id: each opcode is given a unique identifier. If this opcode is executed again (e.g. in a loop), it will receive the same id.

2. Module name: a name of a module where current instruction is executed, It can be a name of a library or a name of an executable itself.

3. Section name: a name of a section in executable file or library where current instruction is executed. Often it will be *.text* or *CODE*, however it some cases (especially with malware) a name of an executable section can be different from standard.

4. Current function name: if a function name of a current instruction can be found we record it to understand *which* function performed a certain part of logic.

5. Opcode: text representation of an assembly instruction together with arguments but without arguments values.

6. Type of module: whether an instruction is executed from the main module of executable under analysis or from the external library.

7. Memory operations: memory operations performed by an instruction. Only *read* or *write* without addresses and values.

8. Name of a function being called: if a current instruction is *call* - a name of a function is being stored.

A real example of raw data is present in the Listing 2. The first line represents header: names of fields are in the same order as in the list above.

```
OPID;MODULE;SECTION;ROUTINE;OPCODE;MODULETYPE;MEMOPS;ROUTINETOCALL
6712;C:\Windows\SYSTEM32\ntdll.dll;.text;RtlInitializeExceptionChain;xor
    ↪ ecx, ecx;isNotMainModule;;
6713;C:\Windows\SYSTEM32\ntdll.dll;.text;RtlInitializeExceptionChain;call
    ↪ eax;isNotMainModule;W;BaseThreadInitThunk
6369;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;mov edi,
    ↪ edi;isNotMainModule;;
6370;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;push ebp;
    ↪ isNotMainModule;W;
6371;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;mov ebp,
    ↪ esp;isNotMainModule;;
6372;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;test ecx,
    ↪ ecx;isNotMainModule;;
6373;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;jnz 0
    ↪ x76f4853d;isNotMainModule;;
6374;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;push dword
    ↪  ptr [ebp+0x8];isNotMainModule;RW;
6375;C:\Windows\system32\kernel32.dll;.text;BaseThreadInitThunk;call edx;
    ↪ isNotMainModule;W;unnamedImageEntryPoint
6714;C:\Users\win7\Documents\malware_PE32\1b6142e3c80362a3f49666856f330510
    ↪ ;.duciuni;unnamedImageEntryPoint;inc ebx;isMainModule;;
6715;C:\Users\win7\Documents\malware_PE32\1b6142e3c80362a3f49666856f330510
    ↪ ;.duciuni;unnamedImageEntryPoint;pushad ;isMainModule;W;
```

Listing 2: Raw data sample

Chapter 9

P4: Detection of running malware before it becomes malicious

Sergii Banin, Geir Olav Dyrkolbotn

Abstract

As more vulnerabilities are being discovered every year[16], malware constantly evolves forcing improvements and updates of security and malware detection mechanisms. Malware is used directly on the attacked systems, thus anti-virus solutions tend to neutralize malware by not letting it launch or even being stored in the system. However, if malware is launched it is important to stop it as soon as the maliciousness of a new process has been detected. Following the results from [6] in this paper we show, that it is possible to detect running malware before it becomes malicious. We propose a novel malware detection approach that is capable of detecting Windows malware on the earliest stage of execution. The accuracy of more than 99% has been achieved by finding distinctive low-level behavior patterns generated before malware reaches it's entry point. We also study the ability of our approach to detect malware after it reaches it's entry point and to distinguish between benign executables and 10 malware families. **Keywords:** Malware detection, Low-level features, Hardware-based features, Information security, Malware analysis, Malware classification

9.1 Introduction

Every year our society becomes more dependent on computers and computer systems, thus attacks on the personal, industry and infrastructure computers start

having more severe consequences [22][25]. According to NIST the amount of vulnerabilities discovered every year has grown almost 3 times during the years 2015-2019 [16]. At the same time a number of vulnerabilities found on Windows platforms has shown 10% growth[17]. Furthermore, the amount of newly discovered Windows malware has grown 30% during the same period[2]. Such security landscape outlines the need for updates in existing and invention of new malware detection mechanisms.

Malware detection methods can be divided based on which features of malware they use for detection: static and dynamic. Static features emerge from the properties of an executable files themselves: file header, opcode and byte n-grams or hashes are known to be used for malware detection[23]. Dynamic features represent the behavior of malware when it runs and can be roughly divided into high- and low-level features[6]. API and system calls, network and file activity are some of the high-level features, while memory access operations, opcodes or hardware performance counters are the low-level features. Basically we perceive behavioral features that emerge from the system's hardware as the low-level ones[5][12][19]. Static features are easier to change for an attacker utilizing techniques such as obfuscation or encryption. However, malware becomes malicious only when it is executed and it is impossible to avoid a behavioral footprint[8]. Even though different techniques such as polymorphism, anti-VM or anti-debug might be used to change high-level behavioral patterns, the functionality of malware remains similar. Moreover, as soon as malware is launched - it is impossible to avoid execution on the system's hardware. That's why in this paper we use low-level features such as memory access patterns for malware detection[7] and classification[5].

Memory access patterns previously were proven to be effective features for malware detection[7] and classification[5]. A memory access pattern is a sequence of read and write operations performed by an executable and will be described in details in Section 9.3. The problem with low-level features is that it is hard for a human analyst to understand the context under which a certain pattern has occurred. A previous work [6] presented an attempt to fill the gap between low-level activity (memory access patterns) and its high-level (more human understandable API calls) representation. During the study it was also found, that under the experimental design used in [6] and [5] most of the recorded behavioral activity emerged not from the main module of an executable (after the Entry Point[1] - AEP) but prior to the moment when instruction pointer (IP) is set to the Entry Point (before the Entry Point - BEP). Without going into much details (see Section 9.2 for details) these findings showed, that it is potentially possible to detect running malicious executable before it starts executing the logic that was put into it by the creator.

[1] In this paper, by Entry Point, we mean the first executed instruction from the main module of executable.

To study these findings, in this paper we use a novel approach in behavioral malware analysis. This approach involves analysis of behavioral traces divided into those generated BEP and those generated AEP: *BEP-AEP approach*. More specifically, we show how memory access patterns can be used for malware detection based on the activity produced BEP. To be consistent in our studies we also compare these results to those achieved based on the activity produced AEP: by the malicious code itself. As paper [5] showed a possibility to classify malware into categories (families or types) using memory access patterns, further we investigate the usefulness of *BEP-AEP approach* for distinguishing between benign executables and different malware families. In order to formalize our future findings we propose the following hypotheses:

Hypothesis 1. *It is possible to detect (distinguish from benign) running malicious executable based on the memory access patterns it produces before it begins to execute malicious code (BEP).*

Hypothesis 2. *It is possible to detect (distinguish from benign) running malicious executable based on the memory access patterns it produces after its Entry Point (AEP).*

And as the logic put into the executable (and makes malware malicious) normally runs AEP we had another hypothesis:

Hypothesis 3. *If Hypotheses 1, 2 are true, then it should be easier (higher classification performance) to detect running malicious executable AEP than BEP.*

To test whether a *BEP-AEP approach* can be used to distinguish between benign and several different categories of malicious executables we had another three hypotheses (directly derived from Hypotheses 1, 2 and 3)

Hypothesis 4. *It is possible to distinguish between several malware categories and benign executables based on the memory access patterns they produce BEP.*

Hypothesis 5. *It is possible to distinguish between several malware categories and benign executables based on the memory access patterns they produce AEP.*

Hypothesis 6. *If Hypotheses 4, 5 are true then it should be easier (higher classification performance) to distinguish between several malware categories and benign executables based on the memory access patterns they produce AEP.*

In order to check the above mentioned hypotheses we decided to perform a series of experiments that consist of several parts. First, we record memory access patterns produced by executables before and after entry point with help of dynamic

binary instrumentation framework Intel Pin[11]. Second, we perform feature construction and selection to create different feature vectors. Last, we train several machine learning (ML) algorithms to check our hypotheses by looking at classification performance of machine learning models.

The remainder of the paper is arranged as following: Section 9.2 provides a literature overview, Section 9.3 describes our choice of methods, Section 9.4 explains our experimental setup, in Section 9.5 we provide results and analyze them, in Section 9.6 we discuss our findings and in the Section 9.7 we provide conclusions.

9.2 Related works

In this section we provide an overview of papers that are related to this article in terms of features used for malware detection as well as methods to extract those features. The first paper we would like to mention is [1] where authors suggested to use Intel Pin based tool to detect malicious behavior by matching it against predefined security policies. Authors record execution flow of executables and describe it by splitting into basic blocks with additional information about each basic block. Among the different sources of information of the basic blocks they used: file modification system calls, fact of presence of *exec* function call and the fact of presence of memory read and write operations. During the testing phase they managed to achieve average path coverage of more than 93% which later helped them to get as much as 100% detection rate on Windows and Linux systems. Even though their datasets were relatively small this work showed promising capabilities of Intel Pin in the malware research.

The next paper [3] focuses more on the low-level features and their use in malware detection. As the features they used retired and mispredicted branch instructions as well as retired load and stored instructions derived from hardware performance counters. Authors achieved classification precision of more then 90%. Their dataset was also relatively small, but they pointed to the effectiveness of low-level feature in malware detection. Later, the same authors expanded their approach by using additional low-level features (near calls, near branches, cache misses etc.) in the paper [4]. They have also expanded their task to multinomial classification of benign and malicious samples divided into several families. This time they achieved 95% precision on a bigger dataset, what, once again, showed capabilities of low-level features use in malware detection and classification.

In [13] another example of application of hardware-based features is proposed. Extending their work from [18], authors propose hardware malware detector that uses several low-level features such as: frequencies and presence of opcodes from different categories, memory reference distance, presence of a load and store operations, amount of memory reads and writes, unaligned memory accesses as well

as taken and immediate branches. Using ensemble specialized and ensemble classifiers authors achieved classification and detection accuracy of around 90% and 96% respectively.

Papers [1][3][13] used information about memory access operations but they didn't use sequences, or patterns, of memory access operations. The first paper where memory access patterns were used for malware detection was [7]. There authors explored a possibility of malware detection based on n-grams of memory access operations. They recorded sequences of memory access operations from malicious and benign executables. After the experiments authors found, that with n-gram size of 96 it is possible to achieve malicious against benign classification accuracy of up to 98%. Later, the same authors explored possibility of a use of memory access n-grams for malware classification [5]. They tested their approach on two datasets label into malware *types* and *families* respectively. After the feature selection they went down to as low as 29 features which allowed them to classify malware types with accuracy of 66% and families with accuracy of 78%. This performance was not as good as pure malicious against benign classification. However, for 10-class classification problem such accuracy showed that this methods (with certain limitations) can be used for malware classification as well. During their studies authors discovered a following problem: memory access patterns provide little context to a human analyst as it is almost impossible to understand which part of the execution flow created a distinctive memory access pattern. To eliminate this knowledge gap, in their next paper [6] they performed an attempt to "correlate" memory access patterns (as low-level features) with API calls (as high-level features). Together with memory access operations they recorded API calls performed by malicious executables. In the end their attempt was not successful: with their methodology they were not able to find any significant "correlation" between memory access patterns and API calls. However, as those events were proven to be independent they showed, that combining API calls and memory access patterns into integrated feature vector results into increased classification performance. On the dataset from [5] they managed to show increased classification accuracy of 70% and of 86% for malware types and families respectively. It was in this paper where they discovered, that most of the behavioral activity they recorded originated from BEP and outlined a need for additional study of such finding.

To the best of our knowledge no one has analyzed the possibility of malware detection and classification based on activity generated BEP. Therefore we think that our paper provides a novel contribution and grounds for further research.

9.3 Methodology

This section describes the methods used in our work. We begin with a description of the process creation flow on Windows. It has multiple stages and it is important to show where we begin to record a behavioral trace: a set of opcodes with their memory access operations, current function and module name. Second, we explain the way we transform a behavioral trace into the memory access patterns that are later used as features for training the ML models. We also describe how we perform a feature selection. Last, we provide a description of ML methods and evaluation metrics.

9.3.1 General overview

As we present BEP-AEP approach in this paper, we have to provide a brief description of a process creation flow the way it is implemented in Windows. The flow of process creation consists of several stages (as described in Windows Internals [28]) and is depicted on the Fig. 9.1. First, the process and thread objects are created. Then a Windows Subsystem Specific process initialization is performed. Lastly, the execution of the new process begins from the Final Process Initialization (Stage 7 on Fig. 9.1). During these stages OS initializes a virtual address space that is later used by a process. Virtual address space is divided into private process memory and protected OS memory. The size of virtual address space depends on the OS type. Normally 32-bit Windows will have up to 4GB while 64-bit - up to 512GB of virtual address space. The virtual address space contains heap, stack, loaded DLLs, kernel and code of the executable (main module). CPU executes instructions (opcodes) from main module or one of the loaded libraries. Each opcode can be divided into several microoperations. Some microoperations are used for arithmetical-logical operations while some are responsible for memory read and write operations. Whenever execution of an opcode requires a memory related microoperation to be executed, Intel Pin tool will record this into the behavioral trace. Intel Pin tool begins to record the behavioral trace at Stage 7, when a new process is started. In the context of a newly created process, Stage 7 generates a BEP activity and includes (but is not limited to) the following actions: installing of exception chains; checking if the process is debugee and whether prefetching is enabled; initialization of image loader, heap manager; loading of all the necessary DLLs. When it is finished, AEP activity begins from execution of Entry Point in the main module. Some malware samples might use packing, thus will unpack itself in the beginning of its execution. However, it is important to understand, that unpacking will be done with instructions from the main module of executable. Thus, with our approach, unpacking will happen AEP.

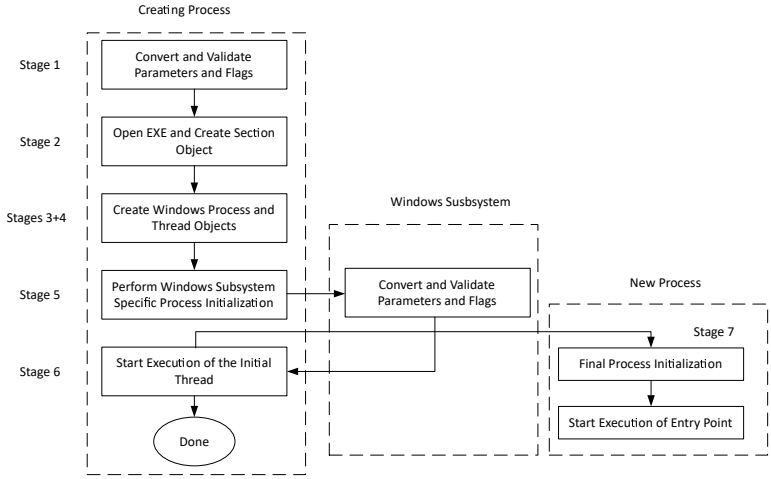


Figure 9.1: Process creation flow [28]

9.3.2 Data collection

In this subsection we describe the way we record behavioral traces. In order to record memory access traces we wrote a custom tool that was based on the Intel Pin framework[11]. Intel Pin is a binary instrumentation framework that allows to intercept execution flow of a process and extract much of the information related to this process such as: memory access operations, opcode, name of a module from which an opcode is being executed and name of a current routine (if possible to derive). Every executable from our dataset (Section 9.4.1) was launched together with the Intel Pin tool. The tool records all the data mentioned above into the behavioral trace. The process of each executable was observed from the beginning of its execution. We recorded the behavioral trace until we gathered 1,000,000 (1M) of memory access operations (similar to [7] and [5]) BEP, and then we continued recording AEP - again until we reached 1M of memory access operations. As we worked with real-life malicious and benign executables, we were not always able to record the desired amount of memory access operations. Some samples reached main module before producing the desired 1M of memory access operations BEP, while some finished their work before producing 1M of memory access operations AEP. It is worth mentioning, that some samples didn't produce any traces AEP. All data collection was done in the Virtual Box virtual machine (VM) in order to protect the host system, allow automation and ensure equal launch conditions for all executables.

9.3.3 Feature construction and selection

Before using our data for training the ML models we have to construct and select features. Memory access operations (BEP or AEP respectively) are concat-

enated into memory access sequence. Based on the methods used in [6] we split memory access sequence produced by an executable into a set of subsequences: n-grams of the length 96. These n-grams are overlapping, so every next n-gram begins from the second element of the previous one. A typical memory access n-gram looks the following way: *RRRWWWWR...WRRRRRRRW*. If we treat *R* as *0* and *W* as *1* n-gram of a size *n=96* becomes binary sequence with potential feature space of 2^{96}. Even though we do not get this amount of distinctive features, our samples still produce millions of features (see Section 9.5). So we need to perform feature selection in order to reduce feature space, reject uninformative features and be able to train ML models in a feasible time. Smaller feature set also contributes for better understanding of the findings and allows "manual" analysis if necessary[5][6].

The feature selection is performed in two steps. On the first step we go down from millions of features to 50,000 by using Information Gain feature selection method. Information Gain (IG) is an attribute quality measure that reflects "the amount of information, obtained from the attribute *A*, for determining the class *C*"[14] and is calculated as following:

$$Gain(A) = -\sum_{k} p_k \log p_k + \sum_{j} p_j \sum_{k} p_{k|j} \log p_{k|j}$$

where p_k is the probability of the class *k*, p_j is the probability of an attribute to take j_{th} value and $p_{k|j}$ is the conditional probability of class *k* given j_{th} value of an attribute. On the second step we use Correlation-based feature selection (CFS)[9] from Weka[10] package (CfsSubsetEval). This method selects a subset of features based on the maximum-Relevance-Minimum-Redundancy principle by selecting features that have maximal relevance for representing the target class and minimal mutual correlation[20]. The reason we did not apply this method to the full feature set is computational complexity. In order to perform CFS feature selection one needs to calculate correlation matrix between all features which would require infeasible amount of computational resources and time. We also select 5,10,15 and 30 thousands of features with IG. It is important to know, that CFS adds features to the feature set until further increase of its merit is no longer possible. Thereby, in the end we use IG to select the same amount of features as was selected by CFS. By doing so we can directly compare performance of two feature selection methods. After the feature selection process we create data that is later used to train ML models. Basically we generate a table, where each row represents values that features from the feature set take for a certain sample. In our paper similarly to [7] we use *1* if feature (memory access n-gram) is generated by sample and *0* if not.

9.3.4 Machine Learning methods and evaluation metrics

We use Weka[10] machine learning toolkit to build and evaluate our models. Similarly to [6] we choose the following ML methods to build our models: k-Nearest Neighbors (kNN), RandomForest (RF), Decision Trees (J48), Support Vector Machines (SVM), Naive Bayes (NB) and Artificial Neural Network (ANN) with the default for Weka[10] package parameters. To evaluate quality of models we use 5-fold cross validation[14] and choose the following evaluation metrics for models assessment: accuracy (ACC) as number of correctly identified samples and F1-measure (F1M) which takes into account precision and recall. We omit using False Positives measure as it is not representative for multinomial classification. The F1M values presented in Section 9.5 are average weighted. For the benign against malicious classification our dataset is nearly balanced (see Subsection 9.4.1), however while doing multinomial classification we had to deal with imbalanced classes. The problem with imbalanced classes is that evaluation metrics does not reflect real quality of models, since simple guessing on the majority class will give high accuracy. To deal with this problem we apply weights to the samples, so that sums of the weights of samples within each class would be equal.

9.4 Experimental setup

In this section we describe our dataset, experimental environment and experimental flow.

9.4.1 Dataset

As Windows is the most popular desktop platform [15] we focused on analyzing Windows malware. Our dataset consists of two parts: malware samples and benign samples. Benign samples were collected from Portable Apps [21] in September 2019. It is a collection of free Portable software that includes various types of software such as graphical, text and database editors; games; browsers; office, music, audio and other types of Windows software. In total we obtained 2669 PE executables. Malicious samples were taken from *VirusShare_00360* pack downloaded from VirusShare[27]. *VirusShare_00360* contained 65518 samples, out of which 2973 were PE executables. For each sample we downloaded a report from VirusTotal[26] and left samples that belonged to the 10 most common families. Those families are: Fareit, Occamy, Emotet, VBInject, Ursnif, Prepscram, CeeInject, Tiggre, Skeeyah, GandCrab. According to the VirusTotal reports, resulted samples were first seen (first submission date) between March 2018 and March 2019. Not all the samples were launched successfully, and from those that launched not all the samples produced traces AEP (most likely executables lacked some resources, e.g. certain libraries). So the amounts of samples that generated

Table 9.1: Amount of samples that generated traces BEP and AEP.

	Benign	Malicious	Fareit	Occamy	Emotet	VBInject	Ursnif	Prepscram	CeeInject	Tiggre	Skeeyah	GandCrab
BEP	2098	2005	573	307	196	164	162	143	127	117	115	101
AEP	1717	1755	573	174	188	162	161	143	115	69	73	97

traces BEP and AEP are different. In the Table 9.1 we present amount of samples of each category that produced traces BEP and AEP.

9.4.2 Experimental environment

For our experiments we used Virtual Dedicated Server with 4-cores Intel Xeon CPU E5-2630 CPU running at 2.4GHz and 32GB of RAM with Ubuntu 18.04 as a main operating system. As a virtualization software we used VirtualBox 6.0.14. We created a Windows 10 VM and disconnected it from the Internet. We have uploaded Intel Pin togther with our custom tool into the VM. We also disabled all built-in anti-virus features to make malware run properly and also because they kept interrupting the work of Intel Pin and created a base snapshot which was used for all experiments. We controlled the VM and data collection process with Python 3.7 scripts.

9.4.3 Experimental flow

During the data collection phase we begin with starting up a VM. Then we upload an executable to the VM and launch it together with Intel Pin tool. When a behavioral trace is ready we download it from the VM and begin a new experiment with reverting a VM to the base snapshot. It is important to notice that benign executables were uploaded together with their folder. This allowed more of the benign applications to run properly and helped to emulate a more real-life scenario, where benign applications often come with various additional resources they need for normal operations.

9.5 Results and Analysis

In this section we provide the classification performance of ML models performed under different conditions. We also analyze the results and show how they align with the Hypotheses from Section 9.1.

9.5.1 Classification performance

Each table contains performance metrics of ML methods (Subsection 9.3.4) achieved with a feature sets (Subsection 9.3.3) of a different length (*FSL* stands for feature set length). Some of the cells contain missing values: due to processing limitations of Weka we were not able to obtain all of the results.

In the Tables 9.2 and 9.3 the results of malicious against benign classification

Table 9.2: Malicious vs Benign BEP classification performance.

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	0.996	0.996	0.996	0.996	0.997	0.997	0.983	0.983	0.693	0.671	-	-
	30K	0.996	0.996	0.997	0.997	0.998	0.998	0.986	0.986	0.983	0.983	-	-
	15K	0.996	0.996	0.998	0.998	0.998	0.998	0.991	0.990	0.983	0.983	-	-
	10K	0.998	0.998	**0.999**	**0.999**	0.998	0.998	0.992	0.991	0.983	0.983	-	-
	5K	0.995	0.995	0.997	0.997	0.997	0.997	0.988	0.988	0.983	0.983	-	-
	9	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988
CFS	9	0.997	0.997	0.997	0.997	0.996	0.996	0.997	0.997	0.988	0.988	0.997	0.997

Table 9.3: Malicious vs Benign AEP classification performance.

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	0.974	0.974	0.985	0.985	0.991	0.991	0.948	0.948	0.735	0.720	-	-
	30K	0.982	0.982	0.990	0.990	0.989	0.989	0.949	0.949	0.795	0.787	-	-
	15K	0.990	0.990	**0.992**	**0.992**	0.988	0.988	0.947	0.947	0.795	0.787	-	-
	10K	0.990	0.990	**0.992**	**0.992**	0.989	0.989	0.955	0.955	0.795	0.787	-	-
	5K	0.989	0.989	0.991	0.991	0.988	0.988	0.960	0.960	0.795	0.787	-	-
	39	0.910	0.909	0.910	0.909	0.908	0.908	0.907	0.906	0.844	0.840	0.910	0.909
CFS	39	0.990	0.990	0.990	0.990	0.989	0.989	0.987	0.987	0.982	0.982	0.991	0.991

BEP and AEP are presented. As we can see, under our experimental design it is possible to achieve classification accuracy of 0.999 for BEP and 0.992 for AEP with 10000 features. CFS selected 9 features for BEP and 39 for AEP. Classification performance with use of CFS-selected features is slightly lower than the best result achieved with those selected by IG. At the same time, it is often higher for the same amount of features selected by IG.

In the Tables 9.4 and 9.5 we present performance of ML models in classifying benign and 10 malicious families using features generated BEP and AEP. In these tables we show classification performance for the imbalanced (*Imb*) and balanced datasets (*Bal*) (Subsection 9.3.4). As we can see, performance of multinomial classification is lower than the benign against malicious classification. By using BEP and AEP features we achieved 0.605 and 0.749 classification accuracy respectively. The main observation that can be derived from these tables is that it is easier to distinguish between benign executables and 10 malware families using features generated AEP than BEP. As the number of samples that produced traces BEP and AEP is different we have also tested the performance of features from BEP on the normalized dataset, when we only take into account samples that produced traces AEP. These results are present in the Appendix Appendix A. We also combined features produced BEP and AEP and tested classification performance of the combined feature set. These results presented in the Appendix Appendix B.

Table 9.4: 10 Malicious families vs Benign BEP classification performance.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.812	0.776	0.812	0.775	0.811	0.771	-	-	0.429	0.462	-	-
		Bal	0.601	0.546	**0.605**	0.549	0.596	0.541	-	-	0.403	0.326	-	-
	30K	Imb	0.812	0.777	0.813	0.776	0.808	0.769	0.789	0.740	0.687	0.667	-	-
		Bal	0.598	0.542	0.600	0.543	0.594	0.538	0.549	0.477	0.433	0.355	-	-
	15K	Imb	0.813	0.777	0.815	0.777	0.811	0.772	0.792	0.745	0.689	0.668	-	-
		Bal	0.594	0.538	0.596	0.540	0.594	0.539	0.565	0.501	0.435	0.355	-	-
	10K	Imb	0.813	0.775	0.814	0.776	0.809	0.770	0.798	0.753	0.689	0.668	-	-
		Bal	0.589	0.531	0.593	0.535	0.590	0.531	0.569	0.502	0.435	0.356	-	-
	5K	Imb	0.789	0.745	0.790	0.745	0.789	0.743	0.782	0.728	0.633	0.591	-	-
		Bal	0.508	0.446	0.513	0.452	0.512	0.446	0.492	0.413	0.382	0.301	-	-
	92	Imb	0.653	0.575	0.653	0.575	0.653	0.575	0.652	0.571	0.651	0.567	0.652	0.573
		Bal	0.184	0.140	0.185	0.140	0.183	0.137	0.182	0.135	0.180	0.128	0.170	0.136
CFS	92	Imb	0.813	0.775	0.813	0.775	0.810	0.769	0.805	0.760	0.740	0.725	0.810	0.771
		Bal	0.585	0.526	0.585	0.527	0.578	0.529	0.572	0.512	0.521	0.467	0.576	0.540

Table 9.5: 10 Malicious families vs Benign AEP classification performance.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.890	0.883	0.902	0.890	0.897	0.888	-	-	0.433	0.420	-	-
		Bal	0.715	0.694	0.724	0.714	**0.749**	0.737	-	-	0.503	0.418	-	-
	30K	Imb	0.891	0.883	0.891	0.883	0.898	0.888	0.727	0.635	0.505	0.527	-	-
		Bal	0.725	0.705	0.732	0.714	0.729	0.708	0.539	0.498	0.493	0.414	-	-
	15K	Imb	0.889	0.881	0.900	0.891	0.898	0.887	0.780	0.712	0.508	0.530	-	-
		Bal	0.724	0.704	0.731	0.714	0.723	0.702	0.625	0.589	0.499	0.420	-	-
	10K	Imb	0.887	0.878	0.900	0.891	0.897	0.886	0.805	0.756	0.509	0.530	-	-
		Bal	0.717	0.695	0.725	0.706	0.729	0.711	0.645	0.606	0.497	0.418	-	-
	5K	Imb	0.866	0.851	0.872	0.854	0.865	0.848	0.747	0.669	0.384	0.384	-	-
		Bal	0.660	0.618	0.661	0.619	0.653	0.605	0.504	0.442	0.433	0.342	-	-
	40	Imb	0.694	0.615	0.694	0.616	0.693	0.613	0.688	0.604	0.670	0.597	0.693	0.617
		Bal	0.306	0.212	0.307	0.206	0.302	0.204	0.299	0.217	0.298	0.193	0.283	0.225
CFS	40	Imb	0.902	0.896	0.903	0.892	0.891	0.880	0.889	0.873	0.872	0.864	0.900	0.890
		Bal	0.725	0.701	0.726	0.704	0.717	0.695	0.706	0.667	0.695	0.653	0.722	0.692

9.5.2 Analysis

From the results presented in Tables 9.2 and 9.3 we can conclude that both Hypotheses 1 and 2 are supported: we can distinguish between malicious and benign behavior BEP and AEP. However, even if there is a visible decline in accuracy when switching from AEP behavior to BEP it is relatively low. Thus, we are not able to conclude that our approach allows to detect malware BEP better then AEP or vise versa. Thereby, we were not able to support or reject Hypothesis 3. This may be a reflection of property of our dataset or a limitation of our approach, and therefore needs further investigation in the future work.

By looking at the numbers of features selected by CFS we can see, that it selects more features for BEP data than for AEP data. And it's not surprising, since the behavior of executables become more diverse AEP: this is where their internal logic starts being executed. It is also confirmed by the amount of unique features

Table 9.6: Evaluation of Hypotheses after analyzing the results

	H 1	H 2	H 3	H 4	H 5	H 6
Supported	**Yes**	**Yes**	-	**Yes**	**Yes**	**Yes**

produced by the samples BEP and AEP. Malicious samples produced more than 1M features BEP, and more than 7M features AEP. On the other hand, benign applications produced more than 4.5M of features BEP and almost 20.5M AEP. This resulted in more then 5M unique features to choose from for BEP classification, and 25M for AEP classification. This also shows, that benign applications are more diverse and produce more distinctive memory access patterns as a result of a more distinctive behavior. And it makes sense, since malware samples belong to 10 malware families, thus should share more common properties according to the definition of malware family from [5].

The results of multinomial classification (Tables 9.4 and 9.5) are more diverse then those for malicious against benign classification. This time, it is clearly easier to distinguish between 11 classes AEP than BEP. Even though multinomial classification accuracy BEP is not that impressive it is still significantly better than potential accuracy of 0.09(09) that can be achieved by random guessing. Thus we can conclude, that Hypothesis 4 is supported. Multinomial classification accuracy AEP was significantly better. So we can conclude that Hypothesis 5 is also supported, thereby Hypothesis 6 as well.

This time CFS has chosen less features for the AEP classification than for the BEP classification. As we mentioned above, malware assigned to one of the families based on its particular functionality. And this functionality becomes revealed AEP. Thereby it is logical to say, that classification of 11 classes is more accurate based on the behavior generated AEP. Table 9.6 present combined results of the Hypotheses evaluation.

9.6 Discussion

In this section we present an attempt to interpret our findings. Earlier, we showed the possibility of malware detection based on the memory access patterns generated BEP. So, we wanted to find an explanation of why the BEP activity of malicious and benign executables is so different. More specifically we wanted to see which high-level activity is responsible for generating specific memory access patterns. As it was written in Subsection 9.3.2, we recorded not only memory access operations, but also routine names for each executed opcode. Since BEP activity happens in the Windows libraries (Subsection 9.3.1) we are always able to derive a name of a current routine. Thereby, a memory access pattern can be represented as a sequence of routine names. However, our memory access

patterns are of a length 96, so having 96 routine names (many of which are repetitive) makes analysis harder and adds redundant information. Thus, we decided to represent each memory access pattern as a sequence of unique routine names. For example, if memory access pattern begins in a routine RTN_1, proceeds into the RTN_2 and finishes in the RTN_1 we store the following sequence: *{RTN_1,RTN_2,RTN_1}*. After performing this search on the 9 features selected by CFS (Subsection 9.5.1) we made a surprising discovery: most of these features originated in *RtlAllocateHeap* routine from the *ntdll.dll* Windows library. Some memory access patterns were completely generated by *RtlAllocateHeap*, while others involved other routines as well. The same memory access pattern can be found in different routine sequences. However, similar to [6], this is the result of our patterns structure and feature construction method (e.g. they can start and end with a sequence of repetetive *W's* or *R's*) that allow similar pattern to appear multiple times in a row. For example, one feature can be found in the following sequences: *{RtlAllocateHeap}, {bsearch,RtlAllocateHeap}, {LdrGetProcedureAddressForCaller,RtlAllocateHeap}, {RtlEqualUnicodeString, RtlAllocateHeap}*. The *RtlAllocateHeap* routine is responsible for allocating a memory block of a certain size from a heap. Thus, when the Final Process Initialization phase of process creating flow needs to allocate a memory block it produces a distinctive activity that allows to distinguish between malicious and benign processes on the stage of initialization. Unfortunately, we were not able to explain why this memory allocation activity can be so distinctive. Neither the official Microsoft documentation on *RtlAllocateHeap*, nor the Windows Internals book[28] gives enough details about memory allocation routines. To answer this question, one may need to reverse engineer *ntdll.dll* library and perform a Kernel-level[24] debugging. And we leave it for the future work, as this is out of scope of this paper.

9.7 Conclusions

In this paper we presented a novel dynamical malware analysis approach, where we distinguish between activity produced before and after Entry Point. As we were able to show, it is possible to distinguish between malicious and benign executables BEP with accuracy of up to *0.999* with 10000 features, and up to *0.997* with just 9 features. It means, that it is possible to detect malicious executables on the stage of their launch: before they become malicious. We also found, that distinguishing between benign samples and samples from 10 malware families is also possible using BEP activity. We have also made an interesting discovery: many of the memory access patterns used for malware detection BEP are generated by the *RtlAllocateHeap* routine. This paper shows a need for further research of the low-level activity use in malware analysis. First of all, we need to make a complete explanation of why the BEP activity of malicious and benign executables are that

different. Second, we have to check the robustness of this approach against the previously unknown malware. Lastly, to fully utilize the capabilities of BEP-AEP approach we need to study the possibility of building the real-time system that uses our approach. This will involve assessment of computational overhead and potential impact on the user experience.

9.8 Bibliography

[1] Najwa Aaraj, Anand Raghunathan, and Niraj K Jha. Dynamic binary instrumentation-based framework for malware defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[2] AVTEST. The independent IT-Security Institute. Malware. https://www.av-test.org/en/statistics/malware/, 2020.

[3] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 703–708. IEEE, 2014.

[4] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. Hlmd: a signature-based approach to hardware-level behavioral malware detection and classification. *The Journal of Supercomputing*, 75(8):5551–5582, 2019.

[5] Sergii Banin and Geir Olav Dyrkolbotn. Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107–S117, 2018.

[6] Sergii Banin and Geir Olav Dyrkolbotn. Correlating high-and low-level features. In *International Workshop on Security*, pages 149–167. Springer, 2019.

[7] Sergii Banin, Andrii Shalaginov, and Katrin Franke. Memory access patterns for malware detection. *Norsk informasjonssikkerhetskonferanse (NISK)*, pages 96–107, 2016.

[8] Pete Burnap, Richard French, Frederick Turner, and Kevin Jones. Malware classification using self organising feature maps and machine activity data. *computers & security*, 73:399–410, 2018.

[9] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[11] IntelPin. A dynamic binary instrumentation tool, 2017.

[12] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2015.

[13] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu Ghazaleh, and Dmitry V Ponomarev. Ensemblehmd: Accurate hardware malware detectors with specialized ensemble classifiers. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[14] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[15] NetMarkeshare. Operating system market share. https://netmarketshare.com/operating-system-market-share.aspx, 2020.

[16] NIST. National vulnerability database. https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all, 2020.

[17] NIST. National vulnerability database: Windows. https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&query=Windows&search_type=all, 2020.

[18] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661. IEEE, 2015.

[19] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[20] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8):1226–1238, 2005.

[21] PortableApps.com. Portableapps.com. https://portableapps.com/apps, 2020.

[22] Reuters. Ukraine's power outage was a cyber attack: Ukrenergo. https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA, 2017.

[23] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. Machine learning aided static malware analysis: A survey and tutorial. In *Cyber Threat Intelligence*, pages 7–45. Springer, 2018.

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

[25] The Verge. The petya ransomware is starting to look like a cyberattack in disguise. https://www.theverge.com/2017/6/28/15888632/petya-goldeneye-ransomware-cyberattack-ukraine-russia, 2017.

[26] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[27] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

[28] Pavel Yosifovich. *Windows Internals, Part 1 (Developer Reference)*. Microsoft Press, may 2017.

Appendix A Classification results: normalized dataset

Here we present classification results for the normalized dataset using features from BEP.

Table 7: Malicious vs Benign BEP classification performance on the normalized dataset.

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50000	0.997	0.997	0.996	0.996	0.998	0.998	0.981	0.980	0.750	0.738	-	-
	30K	0.997	0.997	0.997	0.997	0.998	0.998	0.983	0.983	0.981	0.980	-	-
	15K	0.997	0.997	**0.999**	**0.999**	0.998	0.998	0.990	0.990	0.981	0.980	-	-
	10K	0.997	0.997	**0.999**	**0.999**	0.998	0.998	0.990	0.990	0.981	0.980	-	-
	5K	0.995	0.994	0.996	0.996	0.997	0.997	0.988	0.988	0.981	0.981	-	-
	10	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988
CFS	10	0.998	0.998	0.998	0.998	0.997	0.997	0.998	0.998	0.988	0.988	0.997	0.997

Table 8: 10 Malicious families vs Benign BEP classification performance on the normalized dataset.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.819	0.779	0.818	0.776	0.817	0.774	-	-	0.478	0.500	-	-
		Bal	0.586	0.528	0.588	0.528	0.590	0.532	-	-	0.386	0.292	-	-
	30K	Imb	0.817	0.776	0.819	0.777	0.816	0.774	0.798	0.744	0.685	0.659	-	-
		Bal	0.579	0.517	0.587	0.525	**0.591**	**0.536**	0.555	0.491	0.423	0.337	-	-
	15K	Imb	0.815	0.774	0.821	0.779	0.815	0.770	0.799	0.747	0.686	0.662	-	-
		Bal	0.574	0.511	0.587	0.525	0.584	0.522	0.587	0.525	0.428	0.345	-	-
	10K	Imb	0.817	0.776	0.819	0.777	0.817	0.772	0.800	0.749	0.685	0.660	-	-
		Bal	0.576	0.513	0.580	0.517	0.578	0.518	0.569	0.505	0.422	0.335	-	-
	5K	Imb	0.812	0.769	0.815	0.771	0.814	0.770	0.803	0.750	0.631	0.572	-	-
		Bal	0.571	0.505	0.570	0.505	0.570	0.509	0.564	0.502	0.419	0.313	-	-
	52	Imb	0.663	0.579	0.663	0.579	0.663	0.579	0.661	0.575	0.661	0.575	0.661	0.575
		Bal	0.190	0.135	0.189	0.155	0.189	0.133	0.189	0.144	0.188	0.131	0.190	0.146
CFS	52	Imb	0.823	0.782	0.822	0.781	0.817	0.772	0.809	0.760	0.739	0.721	0.823	0.781
		Bal	0.588	0.531	0.584	0.525	0.584	0.525	0.571	0.510	0.507	0.444	0.567	0.529

Appendix B Classification results: combined feature set

Here we present classification results achieved with combined feature set.

Table 9: Malicious vs Benign classification performance on the normalized dataset using combined feature set

		kNN		RF		J48		SVM		NB		ANN	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50000	0.996	0.996	0.998	0.998	0.999	0.999	0.981	0.981	0.981	0.980	-	-
	30K	0.996	0.996	**0.999**	**0.999**	0.998	0.998	0.982	0.982	0.981	0.980	-	-
	15K	0.997	0.997	**0.999**	**0.999**	0.998	0.998	0.987	0.987	0.981	0.980	-	-
	10K	0.998	0.998	**0.999**	**0.999**	0.998	0.998	0.989	0.989	0.981	0.980	-	-
	5K	0.999	0.999	**0.999**	**0.999**	0.998	0.998	0.995	0.995	0.981	0.980	-	-
	13	0.988	0.987	0.988	0.987	0.998	0.998	0.988	0.987	0.988	0.987	0.988	0.987
CFS	13	0.997	0.997	0.998	0.998	0.996	0.996	0.996	0.996	0.988	0.988	0.997	0.997

Table 10: 10 Malicious families vs Benign classification performance on the normalized dataset using combined feature set.

			kNN		RF		J48		SVM		NB		ANN	
Method	FSL		ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	Imb	0.910	0.905	0.917	0.910	0.910	0.906	-	-	0.802	0.771	-	-
		Bal	0.740	0.718	**0.749**	0.726	0.746	**0.736**	-	-	0.508	0.419	-	-
	30K	Imb	0.906	0.900	0.918	0.910	0.910	0.904	0.787	0.750	0.795	0.761	-	-
		Bal	0.744	0.744	0.743	0.722	0.734	0.718	0.518	0.465	0.495	0.403	-	-
	15K	Imb	0.904	0.898	0.917	0.909	0.908	0.902	0.806	0.771	0.908	0.902	-	-
		Bal	0.744	0.723	0.740	0.720	0.737	0.723	0.608	0.570	0.493	0.400	-	-
	10K	Imb	0.903	0.896	0.909	0.901	0.908	0.898	0.799	0.765	0.790	0.753	-	-
		Bal	0.735	0.708	0.729	0.705	0.728	0.707	0.593	0.550	0.486	0.388	-	-
	5K	Imb	0.792	0.763	0.789	0.759	0.790	0.757	0.754	0.710	0.679	0.647	-	-
		Bal	0.535	0.499	0.534	0.498	0.535	0.499	0.440	0.382	0.408	0.311	-	-
	62	Imb	0.663	0.579	0.662	0.577	0.662	0.577	0.662	0.577	0.660	0.569	0.663	0.579
		Bal	0.190	0.134	0.191	0.156	0.190	0.134	0.189	0.133	0.181	0.418	0.185	0.140
CFS	62	Imb	0.915	0.909	0.916	0.909	0.909	0.903	0.896	0.879	0.876	0.862	0.908	0.903
		Bal	0.744	0.722	0.745	0.724	0.739	0.721	0.723	0.691	0.669	0.625	0.743	0.729

Chapter 10

P5: Fast and straightforward feature selection method: A case of high dimensional low sample size dataset in malware analysis

Sergii Banin

Abstract

Malware analysis and detection is currently one of the major topics in the information security landscape. Two main approaches to analyze and detect malware are *static* and *dynamic* analysis. In order to detect a running malware, one needs to perform dynamic analysis. Different methods of dynamic malware analysis produce different amounts of data. The methods that rely on low-level features produce very high amounts of data. Thus, machine learning methods are used to speed up and automate the analysis. The data that fed into machine learning algorithms often requires preprocessing. Feature selection is one of the important steps of data preprocessing and often takes significant amount of time. In this paper we analyze the Intersection Subtraction (IS) feature selection method that was first proposed and used on a high dimensional dataset derived from the behavioral malware analysis. In our work, we assess its computational complexity and analyze potential strengths and weaknesses. In the end, we compare Intersection Subtraction and Information Gain (IG) feature selection methods in term of potential classification performance and time complexity. We apply them to the dataset of memory access patterns produced by malicious and benign executables. As the result we found, that the features selected by IS and IG are very different. Nev-

ertheless, machine learning models trained with IS-selected features performed almost as good as those trained with IG-selected features. IS allowed to achieve the classification accuracy of more than 99%. We also show, the IS feature selection method is faster than IG what makes it attractive to those who need to analyze high dimensional datasets.

10.1 Introduction

Today many researchers from different research areas have to deal with big amounts of data. Various statistical methods are used to process and understand data that is too big or complex for human analysis. Part of these methods are called *machine learning*: "the automatic modeling of underlying processes that have generated the collected data" [22]. Currently, machine learning is one of the most used approaches when there is a need to predict certain qualities of objects or events. Machine learning algorithms can be divided into supervised (classification and regression) and unsupervised (clustering). In this paper we focus on the classification: prediction of a *class* (type) of a sample based on its *features* (properties). Machine learning is widely used in different fields such as medicine, biology, manufacturing [24] or information security [31] [4]. In information security, machine learning is extensively used in production and research, as the amounts of data that need to be processed are enormous. Especially, machine learning is actively used for malware analysis and detection. According to AV-TEST Institute, there are more than 350,000 new malware samples detected every day [3]. The developers of the anti-virus solutions and researchers work on finding a way to detect malware without having to search through the entire database of already known malware. Moreover, they try to find methods that allow detecting previously unknown malware. The common practice is to find certain characteristics that are common to many malware samples. As the number of malware is very big and growing [3] the machine learning methods are used to deal with the emerging amount of data. Machine learning methods rely on *features*: properties of objects that are being studied. There are two main types of features that can be extracted from malware: static and dynamic. Static features are extracted directly from the malicious file without a need to launch it. Static features are relatively easy to extract, but at the same time it is easier to change them with a use of obfuscation or encryption [1]. However, malware becomes malicious only after it has been launched. The features that occur after the launch of malware are called *dynamic*, or *behavioral* features. We can divide dynamic features into high- and low-level features [6]. File and network activity, API [2] and system calls are some of the high-level features, while opcodes, memory access operations [38] or hardware performance counters are considered to be low-level ones. We name dynamic features that emerge from the system's hardware as the low-level features [5] [21] [25]. To represent a certain

behavioral event with low-level features we need to record and process a significantly bigger amount of data. For example, to describe an API call on the high level we only need its name and arguments passed to it on the call. However, if we decide to record a sequence of opcodes or memory access operations invoked by the API call we'll end up with hundreds if not thousands of events. In this paper, we address a problem that arises from the number of low-level features one needs to record and process while doing dynamic malware analysis.

While machine learning provides good opportunities for automation and analysis, the data that is used by machine learning algorithms has to be preprocessed. Various methods of data preprocessing are described in the literature: discretization of continuous features, attribute binarization, the transformation of discrete features into continuous, dimensionality reduction and so on [22]. The first three of the aforementioned methods are mostly used when the chosen machine learning algorithm works only with a certain type of data. For example, the Naive Bayes classifier needs discrete data to provide a useful outcome. On its turn, dimensionality reduction is often needed, when the amount of features in the dataset is too big. Having too many features can result in increased model training times and model overfitting. There are several ways to reduce dimensionality: feature subset selection, feature extraction and principal components analysis (PCA) [22]. Feature extraction is aimed at finding a set of new features that are constructed as a function of original features. On its turn, PCA finds a new coordinate system with a focus on making the axes aligned with the highest variance of the data. These methods, however, make it harder to analyze the results achieved by the machine learning model: it is sometimes important to understand which features contribute the most towards the classification performance of a model. In such cases, in order to reduce the dimensionality, one may apply feature (subset) selection. With feature selection it is possible to select a certain amount of *best* features based on a certain feature quality measure while keeping the original features intact.

Feature selection is aimed at the dimensionality reduction. Ironically, when the amount of features becomes too big (for example millions as in [8] or [5]) the feature selection becomes a very computationally intense task. The datasets where the number of features is much bigger than the number of learning samples are called High-dimensional low (small) sample size (HDLSS/HDSSS) datasets. Sometimes there are so many features [8], that commonly used machine learning packages simply can not handle such datasets. Storing such a dataset in the single file or database table becomes a problem as well. Thus, the use of the common machine learning packages becomes impossible since they require data to be stored in one piece. On its turn, developing and implementation of a custom machine learning package can take more time than actual data collection and be a hard task for the researchers that don't have enough expertise in software development.

In this paper we focus on the feature selection method that was developed and used in [8] to detect malware based on the memory access patterns. In [8] the dataset contained almost six millions of binary features and 1204 samples divided into two classes. The features represented sequences of memory access operations generated by malicious and benign software. The feature took value *1* if it was generated by a sample, and *0* if not. Utilized feature selection method was aimed at removing those features, that are *present* (take value 1) in the samples of both classes. Thus, it is named Intersection Subtraction (IS) feature selection method. This method helped authors of [8] to reduce feature space from 6M of features to 800. With the use of selected features, it became possible to train a classification model that achieved 98% classification accuracy for the two-class dataset. In this paper, we provide an additional analysis of the IS feature selection method and discuss its advantages and disadvantages. We also compare its performance with an Information Gain [22] feature selection method in a similar malware detection problem. We run our tests on the newer and larger dataset of malicious and benign executables. We show how machine learning models trained with features selected by IS feature selection perform compared to those selected by IG.

The remainder of the paper is arranged as follows. In Section 10.2 we describe the problem and provide an overview of related articles. In Section 10.3 we describe the IS feature selection method, theoretically assess its strengths and weaknesses and explain the context in which IS might be used. In the Section 10.4 we describe our experimental setup, compare feature sets selected by IS and IG, and train machine learning algorithms with the use of selected features. In Section 10.5 we discuss our findings and outline the future work. In the last Section 10.6 we summarize our findings and provide conclusions.

10.2 Background

In this section, we describe the problem area and provide an overview of the papers related to HDLSS datasets and feature selection.

10.2.1 Problem description

While talking about the optimal size of the dataset to be used in machine learning model training, different authors consider different dataset sizes to be optimal. The size of the dataset consists of a number of samples and features. In various sources [26] [15] one can find suggestions, that a minimal amount of samples for training should be between 50 and 80, while 200 and more samples are expected to bring increased accuracy and significantly smaller error rates. Other authors have shown that it is important to have at least 20 to 30 samples per class [11]. When talking about the number of features it is generally considered, that the fewer features there are in the dataset - the better it is for machine learning algorithm [8] [5]

[7] [22]. Some authors advise utilizing *the rule of 10*: in order to train a model with a good performance, one needs to have ten times more samples than the number of features [23]. However, in some cases, the number of features can be significantly higher than the number of learning samples. This may happen due to the context of the research and the nature of data. For example, in [8] the authors describe a novel malware detection approach. They record memory access operations performed by malicious and benign executables, split them into n-grams of various sizes and use those n-grams as features for training the machine learning models. Each feature could take value *1* or *0* if the n-gram represented by the feature was or was not generated by the sample respectively. The sequence of memory access operations is a sequence of *Reads* (R) and *Writes* (W). In their work, authors record a first million of memory access operations performed by each executable after it was launched. Afterwards, the sequence of memory access operations is being split into the set of overlapping n-grams of a size 96. Since memory access operations take only two possible values (R and W), the potential feature space of the above-mentioned approach is 2^{96} if a sequence of memory access operations would be completely random. However, as the same authors mention in their next paper [5], the memory access operations are not random. Thus in [8] their initial feature space is "only" about 6M of features. They had 1204 samples divided into two classes. This can be considered a good sample size based on what was suggested in [11] [15]. However, the amount of features generated under such experimental design makes it impossible to follow "the rule of 10". A straightforward approach in such conditions could be to simply use all the data for training the machine learning model. However, just the storage of a complete dataset from [8] would take more than 6GB of space. Popular machine learning frameworks such as Weka [19] or Scikit-learn [26] are not suited to load and handle so much data. This shows a need for dimensionality reduction. In the works similar to [8] or [5] it is important to keep the original features in order to be able to interpret results. For example, having the results from [8] it might be possible to understand which memory access patterns make malicious behavior distinctive from the benign behavior. Thereby, dimensionality reduction methods such as feature extraction or PCA are not applicable in such cases. On its turn, feature selection can help to select a subset features without hindering their original state.

Feature selection methods can be divided into several categories: filter, wrapper and embedded methods. Filter methods choose features based on a certain quality measure such as Pearsons correlation, Chi-square, mutual information and so on. Wrapper methods choose features based on the classification performance of the target machine learning model trained with the use of those features [33]. Wrapper methods are very computationally intense since for every possible feature subset there is a need to train and test the machine learning model. Embedded

methods, as the name states, are embedded in the machine learning algorithms. Algorithms such as Decision Trees [22] perform feature selection simultaneously with model training. However, the computational overhead is higher than one of the filter methods and such algorithms are susceptible to overfitting [9] and are not suitable for high dimensional data [33]. So for the research similar to [8] the most suitable approach for dimensionality reduction will be a filter-based method. In the case of (very) high dimensional data, it is crucial to have a feature selection method with the lowest possible computational overhead. The perfect feature selection method will have a computational complexity of $\mathcal{O}(n)$ that is linear to a number of features *n*. But such a method does not exist, since filter methods are aimed to select features that *represent* classes (and consequently samples) in the best possible way [22]. Thereby, while choosing the feature selection method to work on the high dimensional dataset it is desirable to choose a method with the computational complexity of $\mathcal{O}(mn)$ where *m* is the number of samples in the dataset.

The use of different filter-based feature selection methods are described in various papers. Information Gain [7] [24], Correlation-based feature selection [5] [17] and ReliefF [17] are some of the common feature selection methods. *Information Gain* (IG) ranks features based on entropy in respect to the classes and can be described as "the amount of information, obtained from the attribute A, for determining the class C" [22]. Basically, in order to perform a feature selection based on IG one have to calculate probabilities of an attribute to take certain values and relevant class-conditional probabilities. This results in a computational complexity around $\mathcal{O}(mn)$, where *n* is the amount of features and *m* is the amount of samples. *Correlation-based Feature selection method* (CFS) was proposed in [18] and is aimed at selecting the subset of features that have a high correlation to the class but low correlation between each other. By doing so it is possible to find a subset of features with minimal redundancy. The problem with this method, is that it requires to calculate a pairwise correlation matrix between all of the *n* features and *m* classes which requires $m((n^2 - n)/2)$ operations. The feature selection search could require an additional $(n^2 - n)/2$ operations in a worst-case scenario. With a potential computational complexity of $\mathcal{O}(m((n^2 - n)/2) + (n^2 - n)/2)$ the use of CFS for high dimensional data becomes very problematic. For example, just storing of correlation matrix needed for 6M of features in [8] would require at least 18 TB of space. Thus, in order to apply CFS on high dimensional datasets it might be useful to first reduce a feature space with another, less computationally intense, feature selection method and only after apply the CFS [5]. *ReliefF* ranks features based on their ability to separate close samples from the different classes [22]. In order to perform feature selection with ReliefF, it is first important to calculate a distance matrix between all samples. The resulting computational complexity of

the method can be roughly estimated as $\mathcal{O}(n((m^2 - m)/2))$ that is almost *m/2* times more than the one of the IG. Having a large *n* makes the use of ReliefF less favorable than IG.

Based on the assumptions about the computational complexity of the above-mentioned feature selection methods one can make a conclusion, that IG might be one of the best choices when it comes to the high dimensional datasets. The problem is that even the feature selection methods with $\mathcal{O}(mn)$ complexity become slow with the large numbers of *n*. And as we mentioned above, common machine learning packages are not suitable to work with big datasets. Thus, a researcher that needs to perform feature selection on such datasets is forced to develop a custom implementation of feature selection algorithm with regards to the data in interest. In this case, inefficient implementation of the common feature selection algorithm may result in significant use of time and even inability to obtain results (e.g. due to the lack of virtual memory). For example, the Information Gain of a feature is calculated with the following formula:

$$Gain(A) = -\sum_k p_k \log p_k + \sum_j p_j \sum_k p_{k|j} \log p_{k|j}$$

where p_k is the probability of the class *k*, p_j is the probability of an attribute to take j_{th} value and $p_{k|j}$ is the conditional probability of class *k* given j_{th} value of an attribute [22]. This shows, that it is necessary to "count" how many times each attribute takes a certain value in total and when a certain class is given. Lets rewrite previously mentioned computational complexity of IG as $\mathcal{O}(nT_{qmeaureIG})$ where $T_{qmeaureIG} = f(m)$ is the computational time needed to calculate the quality measure (Information Gain in this case) of a feature. We will need $T_{qmeaureIG}$ later, to show that the IS feature selection method works faster than IG, which is important when working with high dimensional datasets. Thus, it is easy to see that the inefficient implementation of IG can significantly increase the time needed to obtain the results. As we will later show, it is possible to overcome this problem with a Intersection Subtraction feature selection method.

10.2.2 Literature overview

In this subsection, we refer to papers where authors addressed the problems related to HDLSS datasets and feature selection on them. In the [12] authors outline both *curses* and *blessings* of high dimensionality. By blessings of dimensionality, they mention the phenomenon of measure concentration and the success of asymptotic methods. While talking about curses of dimensionality authors outline several areas where they can occur: optimization, function approximation and numerical integration. They also stress attention to the fact, that many "classical" statistical methods are based on the assumption, that the amount of features *n* is less than the

amount of samples *m*, while $m \to \infty$. However, these methods may fail if $n > m$, especially when $n \to \infty$. Other authors in [14] outline the following challenges of high dimensionality: "(i) high dimensionality brings noise accumulation, spurious correlations and incidental homogeneity; (ii) high dimensionality combined with large sample size creates issues such as heavy computational cost and algorithmic instability" [14]. As well as authors of [12] they outline, that traditional statistical methods may fail when used on high dimensional data. The authors of [40] review the performance and limitations of several common classifiers such as Naive Bayes, Linear Discriminant Analysis, Logistic regression, Support Vector Machines and Distance Weighted Discrimination in the case of two-class classification problem on HDLSS datasets. They also say, that if the number of features $n \to \infty$ and both classes are from the same distribution "the probability that these two groups are "perfectly" separable converges to 1" [40]. In simple words, it means, that with a large enough amount of features it should be possible to construct a set of rules (build a classifier) that will perfectly fit (overfit) the training data. This fact outlines the importance of thorough feature selection. It will improve the capability of machine learning algorithms to create models with good *generality* and *interpretability*. The model with good generality is the model that is capable of generalizing over the dataset; such model would not be significantly changed if the number of samples in the dataset is slightly increased/decreased [40]. A model with good interpretability makes the analysis of the model itself easier. The fewer features are involved during the training the easier it is to analyze the obtained model. For example, authors of [5] underline the importance of the fact, that having 29 features instead of 6M or 15M helps in the understanding of the underlying processes. They performed multinomial (10 class) malware classification with the use of features constructed from memory access patterns. Similarly to [8], they used memory access 96-grams as features. Such feature, if found to be important in the classification, can not be directly understood by a human analyst. Thus, in [6] they made an attempt to interpret memory access sequences with more high-level system events (API calls). Such analysis would be much harder if they had millions of features instead of 29.

Various authors addressed the problem of feature selection on HDLSS datasets more specifically. For example same authors in [36] and [37] present possible improvements to the PCA in HDLSS cases. In [36] they propose a way to estimate singular value decomposition of the cross data matrix. Later, in [37] authors explore the impact of the geometric representation of HDLSS data on a possibility to converge the dataset to an n-dimensional surface. The authors of [13] propose a nonlinear transformation of HDLSS data. They showed, how transformation based on inter-point distances helps to increase final classification accuracy. In the [39] the authors propose a hybrid feature selection method that is based on ant-

lion optimization and grey wolf optimization methods (ALO-GWO). They evaluate the performance of the proposed method on several HDLSS datasets. The authors show that the ALO-GWO feature selection method provides a good balance between the performance of models and the ability to reduce a feature space. The above-mentioned papers addressed the problem of feature selection on HDLSS. However, the number of features in the dataset used in those papers rarely exceeded several tens of thousands (e.g. in [39]). On their turn, authors of [16] during the test of their feature selection method used a dataset with more than 3M of features. In their work, they proposed a feature selection method based on bijective soft sets (BSSReduce). They claim, that the computational complexity of the method is $\mathcal{O}(m)$ where m is the number of samples. This might have been a perfect feature selection method for the HDLSS datasets. However, after reviewing the provided algorithms, it looks like their approach relies on the precomputed bijective soft sets that have to contribute to the computational complexity as well. Nevertheless, the results of testing the BSSReduce on the several HDLSS datasets showed, that it is capable of significant dimensionality reduction while keeping a competitive level of the trained models performance. It could be useful to compare BSSReduce with our method, unfortunately, authors of BSSReduce did not provide the source code of their tool. An approach different from the previously mentioned papers is present in the [5]. The authors of the paper did not focus on feature selection. However, they needed to reduce feature space in two HDLSS datasets from 6M and 15M of features. Authors said that "models should be simple enough" [5] to make their analysis easier. In order to reduce a large feature space, they performed feature selection in two steps. On the first step, they used custom implementation of Information Gain feature selection to reduce feature space to 50K and fewer features. On the second step, they took the best 5K feature selected by IG and used them in CFS implementation from Weka. This resulted in 29 features selected by CFS. The models trained with just 29 features performed almost as good as a model trained on 5K and more features. For Naive Bayes and Support Vector Machine algorithms, smaller feature set even allowed to increase the performance of trained models. Such approach has its own limitations. CFS is aimed at selecting features that are not correlated with each other. However, since the first feature selection step utilizes IG, there is no guarantee that features passed to the CFS does not have a strong mutual correlation. But as we mentioned above, running CFS on the HDLSS dataset with millions of features requires enormous computational resources and sometimes impossible.

10.3 Intersection Subtraction selection method

In this section, we describe the IS feature selection method and evaluate its strengths and weaknesses.

10.3.1 The context

Before describing the Intersection Subtraction feature selection method we need to describe a context under which its use becomes meaningful. This method was developed during the research described in [8]. The task was to detect malware based on the memory access traces. To do this, malicious and benign executables were launched together with custom-built Intel Pin [20] tool. The raw data consisted of the first 1M of memory access operations performed by each executable. The sequences contained *W* for each write operation and *R* for each read operation performed by an executable. These sequences were later divided into a set of overlapping n-grams of various sizes. For example, a sequence *[WWRWRR]* of a length 6 can be divided into the set of 4-grams in the following way: *[WWRW,WRWR,RWRR]*. The n-grams were directly used as features for machine learning models training. Each feature got value *1* if the corresponding n-gram was generated by the sample regardless of the number of times it was encountered in the trace of a certain sample. In other cases, the feature got value *0*. As the goal of the [8] was to be able to detect malware, it is possible to state, that features that obtain *1* (are present within a certain class) pose greater interest. Such approach allows to state, that *presence* of certain memory access n-grams is the sign of malicious behavior. The dataset from [8] was nearly balanced and samples were divided into two classes. So the context of the use of the proposed feature selection method is the following: two-class classification problem on a balanced dataset with binary features.

10.3.2 Feature selection algorithm

The feature space in [8] was around 6M of unique memory access n-grams of a size 96. By the time of writing, authors were not able to implement any common feature selection method (for example IG) to operate on such dataset. Thus, they implemented the following feature selection method. It includes the following steps:

1. Construct two vectors of features for each class. The feature is included in the vector of the class if the corresponding memory access n-gram was generated by a sample from this class.

2. Having two vectors constructed, remove from them features that are present in both vectors. Having this done we obtain two vectors of class-unique features. In other words, we *subtracted an intersection* of two feature sets from both of them.

3. Decide on the size of the final feature set k.

4. From each of the class-unique features vectors select *k/2* features with the highest class-wise frequency. A class-wise frequency is the proportion of samples within the class that generate a corresponding memory access n-gram.

5. Use the *k* selected features to construct the final dataset with reduced dimensionality.

The resulting dataset is later used to build machine learning models. The operation performed in Step 2 is quite similar to the symmetric difference of two sets. However, we prefer to say that we subtract intersection from both sets, as we need those sets to be separated until the last step. It is also worth mentioning, that having an intersection of two feature sets allows to explore features that fell into it. It might be useful for additional analysis of the results [8].

10.3.3 Computational complexity

Lets discuss the potential computational complexity of Intersection Subtraction (IS) feature selection. As data is already labeled (samples divided into two classes) the feature vectors from the *Step 1* are ready from the beginning. *Step 2* requires finding an intersection of two sets. Imagine we have two sets A and B with cardinality of *a* and *b* respectively. In order to find the intersection of A and B we need to compare all elements of set A with all elements of set B. Such operation will have a computational complexity of $\mathcal{O}(ab)$. Let's denote the intersection of A and B as $C = A \cap B$ with cardinality *c*. Subtracting the elements of C from A and B, similarly to the previous operation, will have the computational complexity of $\mathcal{O}(ac + bc)$. The resulting computational complexity of $\mathcal{O}(ab + ac + bc)$ may look quite high already, since both *a* and *b* are large in case of HDLSS datasets. However, the real implementation of IS feature selection with the use of Python programming language shows, that execution of the *Step 2* does not take significant time (see Section 10.4). First of all, according to [28], subtraction *A-C* (set difference) will have computational complexity of $\mathcal{O}(a)$. So we can already rewrite previously mentioned computational complexity of Step 2 with $\mathcal{O}(ab + a + b)$. Moreover, if we are not interested in the intersection *C* itself, we can utilize two operations *A-B* and *B-A* in order to obtain sets of class-unique features. Complexity of such approach will be $\mathcal{O}(a + b)$. The *Step 4* requires the calculation of class-wise frequencies of the features. In our particular case, when features are binary, we only need to count how many samples from each class has value *1* of a certain feature. The Step 4 will then have $\mathcal{O}((a - c)m + (b - c)m)$ computational complexity. Here, *m* is the number of samples in the dataset, *a-c* is the amount of class-unique features from set A and *b-c* - from set B. It is also worth mentioning, that Step 4 can be optimized. Let's assume that the dataset is perfectly balanced, so

we have two classes with *m/2* samples. Since our IS feature selection is aimed on finding class-unique features, we can only search for *1s* among *a-c* and *b-c* features of *m/2* samples of each class. So the Step 4 can be optimized to have a complexity of $\mathcal{O}((a-c)m/2+(b-c)m/2)$. Lets now try to assess the overall computational complexity of the IS feature selection. Let us have the initial amount of features *a+b=n* and *m* samples. The amount of features from intersection *c* is normally smaller than both *a* and *b* (here we assume, that $A \not\subset B$ and $B \not\subset A$). Having this we can conclude, that the complexity of *Step 2* $\mathcal{O}(ab+a+b)$ after substitution will be smaller than $\mathcal{O}(n^2)$ for all $a > 1$. On its turn, the complexity of *Step 4* $\mathcal{O}((a-c)m/2+(b-c)m/2)$ should be smaller than $\mathcal{O}(mn)$. The resulting complexity of $\mathcal{O}(ab+a+b+(a-c)m/2+(b-c)m/2)$ should be smaller than $\mathcal{O}(n^2+mn)$. The feature selection method where the upper boundary of computational complexity is described with n^2 is not what we outlined in Section 10.2 as a good feature selection method for HDLSS dataset. Lets now make a substitution similar to the one we made in Section 10.2. First, lets substitute *m* with $T_{qmeaureIS} = g(m)$ which is the time needed to calculate class-wise frequency of a feature. Second, the time T_{in} needed to find whether a certain feature from one set is present *in* another set (to find an intersection, or to subtract these features from the set) is relatively small. Thus, the updated computational complexity of IS feature selection will be smaller than $\mathcal{O}((nT_{in})^2 + nT_{qmeaureIS})$ which can be smaller than $\mathcal{O}(nT_{qmeaureIG})$ of IG. We will prove this in Section 10.4.

10.3.4 Theoretical assessment

In this subsection, we discuss potential outcomes of the IS feature selection. As we already mentioned, IS feature selection is potentially faster than a more common IG feature selection. This makes IS attractive for the high dimensional datasets. However, speed comes with a price. Let's look at the potential disadvantages of IS feature selection. As we described at the beginning of this section, the use of this method makes more sense when we are interested in finding features the *presence* of which poses particular interest. However, it might happen, that in the dataset will be no class-unique features. In other words, it will be impossible to say, that if a certain feature of a sample takes value 1, then this sample belongs to a certain class. In such case, it will be impossible to find an intersection of two feature sets. The other problem is potential information loss due to intersection removal. Imagine we have a dataset that is represented in the Table 10.1. It has 4 features and 4 samples labeled into two classes C1 and C2. IS feature selection will remove features f1 and f3 since they obtain value 1 (are present) in both classes. The remaining features f2 and f4 will not allow us to generate a rule that will be able to distinguish between samples s2 and s4. This example is quite small, but on the larger dataset removing a feature that takes value *1* in e.g. all samples

Table 10.1: Sample dataset 1

	f1	f2	f3	f4	
s1	1	1	1	0	C1
s2	1	0	0	0	C1
s3	1	0	1	1	C2
s4	0	0	1	0	C2

of one class and only in one sample of another class can lead to the inability of building a model with good performance. Such feature would be most likely selected by IG feature selection. The last disadvantage of the IS feature selection is potentially poor performance on the multinomial datasets. If we increase the number of classes we will end up in the situation of growing intersection size. In such case, the IS will remove more features from the feature space resulting in increased information loss. We begin with the description of our dataset and experimental environment. Later, we explain the basics of memory access operations and explain the way we record and process the data.

10.4 Experimental evaluation

In this section we describe experimental evaluation of the IS feature selection method. We show how IS feature selection can be applied for malware detection. During experimental evaluation we compare performance of features selected by IS and IG. On the Figure 10.1 we depict general data-flow of our experiments. We start by recording memory access operations produced by benign and malicious executables. After, we split sequences of memory access operations into n-grams. Then we apply feature selection methods to select best features (n-grams). In the end, we use these features to train machine learning models and compare performance of the models trained with a use of features selected by different feature selection methods. Before presenting the results achieved by machine learning models, we show the experimental time complexity of the IS and IG feature selection methods. We also check how similar the feature sets selected by different methods are.

We now proceed with the description of our dataset, experimental environment and the way we collect and process the data.

10.4.1 Dataset

In this work we use dataset similar to the one used in [7]. It consists of 2098 benign and 2005 malicious Windows executables. Malicious executables were downloaded as part of *VirusShare_00360* pack available at VirusShare [35]. Malicious samples belong to the following malware families: Fareit, Occamy, Emotet,

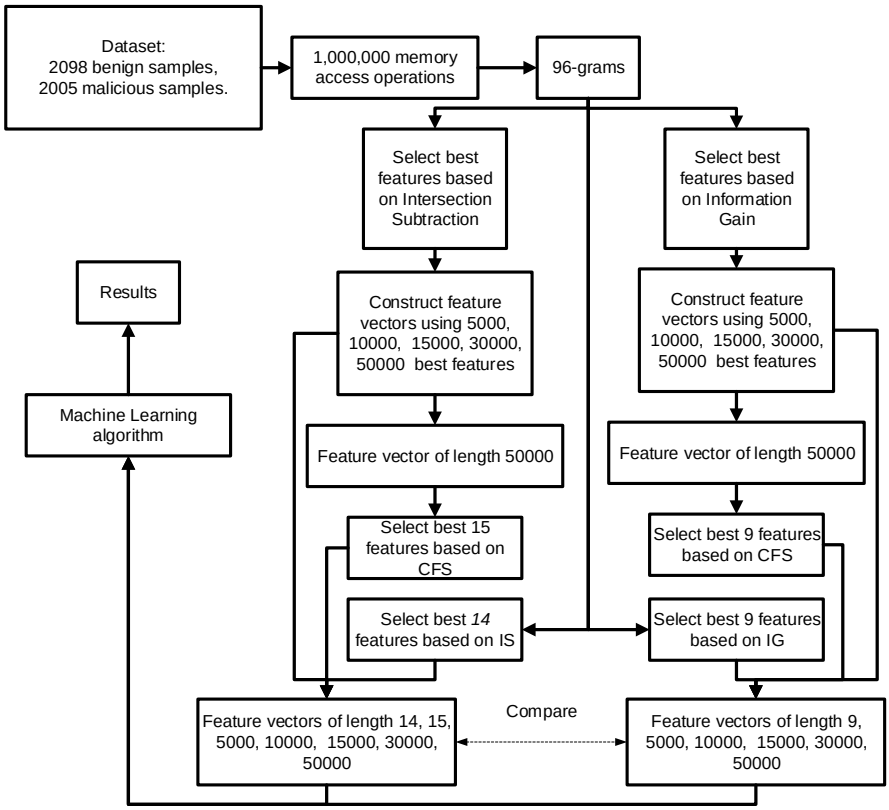


Figure 10.1: The flow of data collection and feature selection

VBInject, Ursnif, Prepscram, CeeInject, Tiggre, Skeeyah, GandCrab. According to the VirusTotal [32] reports, our samples were first seen (first submission date) between March 2018 and March 2019. Benign executables are the real software downloaded from Portable Apps [27] in September 2019.

10.4.2 Experimental environment

In order to perform dynamic malware analysis, we need to avoid the influence of any environmental changes, so that all executables are launched in similar conditions. To achieve this we used an isolated Virtual Box virtual machine (VM) with Windows 10 guest operating system. VMs were launched on the Virtual Dedicated Server (VDS) with 4-cores Intel Xeon CPU E5-2630 CPU running at 2.4GHz and 32GB of RAM with Ubuntu 18.04 as a main operating system.

10.4.3 Memory access operations

The executables used on Windows operating systems are compiled into the files in PE32 format. Files of PE32 format contain *header* and *sections*. The header contains the metadata that is used by operating system in order to properly load an executable into memory and prepare all the necessary resources. The sections contain information about imported and exported functions, resources, data and the executable code. The executable code is stored in the binary form which can be represented as *opcodes*. Opcodes (or assembly commands) are the basic instructions that are executed by the CPU. Execution of some instructions will not require memory access. Foe example execution of *MOV EAX,EBX* opcode will not result in memory access, since data is being moved between registers in the CPU. At the same time, *MOV EDI, DWORD PTR [ebp-0x20]* will generate a *Read (R)* memory access, since the data has to be read from the memory. On its turn, the *ADD DWORD PTR [EAX],ECX* will require *Reading (R)* the value from the memory location addressed by *[EAX]* and then *Writing (W)* the result of the addition to the memory. The sequences of opcodes were previously proven to be a source of effective features for malware detection [10] [34] [30]. When the sequence of opcodes is executed it generates a sequence of memory access operations. Two previous statements allow for memory access sequences to be a potential source of features for malware detection [8]. Under our experimental design we use only the type of memory access operation: *R* for Read and *W* for Write. We do not use the value that is transferred to or from the memory as well as the address of the memory region in use.

10.4.4 Data collection

Each malware sample was launched on the clean snapshot of VM. During the execution of each sample, we recorded the first million of memory access operations produced after the launch. This was done with the help of a custom-built Intel Pin [20] tool that was launched together with the sample inside the VM. The VM had all built-in anti-virus features disabled to make malware run properly and also because they kept interrupting the work of Intel Pin. The automation of VM and data collection were performed with the help of Python 3.7 scripts.

The memory access traces were first stored in the separate files. After, they were split into the sequence of overlapping n-grams of the size 96 (96-grams). We choose n-gram size (as well as the amount of recorded memory accesses) based on the conclusions of their effectiveness drawn in [8]. The n-grams of memory access operations for each sample are then stored in the MySQL table. This table took 28.5 GB of storage.

10.4.5 Feature selection and machine learning algorithms

We implemented IS feature selection algorithm with Python. The custom implementation of IG feature selection algorithm was similar to one in [7]. That implementation allows to run feature selection in multiple threads, which significantly speeds up the process. We found, that samples produced more than 5.5M of unique n-grams (features). Benign samples generated more than 4.5M of features, while malicious - more than 1M of features. When performing IS feature selection we found, that benign and malicious samples shared almost 600K common features. Subtraction of those features resulted in almost 4M and 430K of class-unique benign and malicious features respectively. According to the algorithm from Section 10.3 we selected 50,30,15,10 and 5 thousands of features. We selected a similar amount of features with the IG feature selection algorithm as well. Similarly to [5], [6] and [7] we wanted to reduce feature space even more, so that our models are simple enough for future human analysis. Thus, we used CFS feature selection method from Weka [19] to select the most relevant and least redundant features from 50K features selected by IS and IG. As the result we obtained 15 features from IS-based 50K feature set, and 9 features from IG-based 50K feature set. As CFS appends features to the feature set until the increase of its merit is no longer possible, it is impossible to control the final amount of selected features unless the GreedyStepwise search is applied. However, such search never finishes its work when applied to the larger feature sets in our experimental environment. We wanted to compare the performance of IS and IG with the CFS as well. So we tried to select the same number of features with IG and IS. However, CFS selected 15 features. And as the IS have to select equal amount of features from each class (Section 10.3) we decided to select 14 features with IS (7 from each class).

The selected features were later used to build machine learning models. The data that is actually fed into mechine learning algorithms is basically a *bitmap of presence* [8]: if a certain sample (row) generates a certain feature (column), then this feature takes value *1* for this sample. In the opposite case the feature takes value *0*. We used the following machine learning algorithms from Weka: k-Nearest Neighbors (kNN), RandomForest (RF), Decision Trees (J48), Support Vector Machines (SVM) and Naive Bayes (NB) with the default Weka [19] parameters. We assessed the quality of the models with 5-fold cross validation [22]. Accuracy (ACC) as the amount of correctly classified samples and F1-measure (F1M) that takes into account precision and recall were chosen as evaluation metrics. Further in this section, we present the classification performance of the machine learning models.

10.4.6 Time complexity

One of the reasons to use IS feature selection is that it is relatively faster than the other common methods. In this subsection, we provide time taken by IS and IG methods to select 50K of features from the initial 5.5M distinct features. It took 302 seconds (∼5 minutes) for IS to select 50K features. In contrast, the IG used 18,560 seconds (∼5.15 hours) to select 50K features when running in one thread. While being launched in 16 threads, IG used 1168 seconds (∼20 minutes) to select 50K features. Further increase in the number of threads does not make sense, since this is the maximum amount of threads available at our VDS. As we can see, single-threaded IS works 3.8 times faster than IG ran with 16 threads and 61.5 times faster than IG ran with one thread. To find an intersection of benign and malicious feature sets the IS used 1.18 seconds as the average of 1000 runs. It has used an additional 0.7 seconds to subtract intersection from both feature vectors. The actual implementation of our feature selection algorithms did not load the entire dataset at the same time. Thus, it is impossible to directly measure the time needed to calculate the quality measure of a single feature, since it is calculated in iterations. But indirect assessment (we divide overall time by the total amount of features to go through) showed, that IS needed around $5.5 \cdot 10^{-5} s$ to assess a single feature, and IG needed $2.12 \cdot 10^{-4} s$ and $3.4 \cdot 10^{-3} s$ to assess a single feature with 16 and 1 thread respectively. It is important to mention, that the times provided are relevant to our data structure and the way we store our data. For instance, the fact that we stored memory access n-grams for each sample in a separate cell of the database table could affect the time needed to perform feature selection.

10.4.7 Analysis of selected feature sets

Here we analyze how different are the feature sets selected by IS and IG. In the Table 10.2 the Feature amount column shows the size of the feature set for IS and IG methods; the Common features column shows the number of similar features selected by IG and IS for the corresponding feature set size; the Difference ratio column shows the ratio of the distinct features and is calculated as *(Feature amount - Common features)/Feature amount*. As we can see, most of the features selected by the IS method are different from those selected by IG. It complies with the theoretical assessment of IS (see Section 10.3), where we explained that IS may discard features with potentially high information gain only because they get value *1* in both classes. As we mentioned before, we used CFS feature selection on the feature sets of the size 50K. It is worth mentioning that CFS selected completely different features when working with 50K feature sets selected by IS or IG. When using IG and IS to select the same amount of features as selected by CFS we also obtained completely different feature sets.

Table 10.2: Difference between feature sets selected by IS and IG.

Feature amount	Common features	Difference ratio
50K	994	0.98
30K	994	0.97
15K	979	0.93
10K	955	0.9
5K	812	0.84
IG/IS 9/14	0	1

10.4.8 Classification performance

In this subsection, we present the classification performance achieved by the machine learning algorithms. Tables 10.3 and 10.4 contain evaluation metrics of machine learning models trained with the feature sets of a different length selected by different feature selection algorithms. There, *FSL* stands for feature set length, *ACC* stands for accuracy and F1M stands for F1-measure. As we can see, both feature vectors allowed to achieve a quite high classification accuracy. The best performing RF model that used 10K features selected by IG managed to classify 99.9% of the samples correctly. On its turn, features selected by IS allowed to build kNN and RF models with an accuracy of 99.8%. As we can see, in most cases models built with the use of features selected by IS have slightly lower classification performance. However, the difference in accuracy or F1-measure between IS and IG features is most of the time less than 1%. Thus, it is hard to conclude whether the features selected by IG is significantly better than those selected by IS. There is one exception for NB models built with the use of 50K features. As it is possible to see, the NB model trained with 50K features selected by IG has significantly lower accuracy and F1-measure than the one trained with 50K features selected by IS. This difference might be explained by the nature of features selected by IS and the limitations of the NB method. While building the model, Naive Bayes assumes that features are independent. However, Information Gain feature selection potentially selects a lot of mutually correlated features. The IS does not take into account the mutual correlation between features as well. However, there should be less correlated features selected by IS, since one half of the features will not have *1s* in one of the classes and vice versa. These properties of Naive Bayes were studied in [29]. Even though CFS selected completely different features in IS and IG cases, the models built with those features showed a quite similar classification performance. We will discuss this in Section 10.5. When we used IS and IG to select the number of features similar to CFS we found, that models built with these features perform slightly worse if compared to the models built with features selected by CFS. This finding can be explained by the natures of CFS and IS al-

Table 10.3: Classification performance with a use of features selected by IG

		kNN		RF		J48		SVM		NB	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
InfoGain	50K	0.996	0.996	0.996	0.996	0.997	0.997	0.983	0.983	0.693	0.671
	30K	0.996	0.996	0.997	0.997	0.998	0.998	0.986	0.986	0.983	0.983
	15K	0.996	0.996	0.998	0.998	0.998	0.998	0.991	0.990	0.983	0.983
	10K	0.998	0.998	**0.999**	**0.999**	0.998	0.998	0.992	0.991	0.983	0.983
	5K	0.995	0.995	0.997	0.997	0.997	0.997	0.988	0.988	0.983	0.983
	9	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988	0.988
CFS	9	0.997	0.997	0.997	0.997	0.996	0.996	0.997	0.997	0.988	0.988

Table 10.4: Classification performance with a use of features selected by IS

		kNN		RF		J48		SVM		NB	
Method	FSL	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M	ACC	F1M
IS	50K	0.991	0.991	0.997	0.997	0.997	0.997	0.983	0.983	0.982	0.982
	30K	0.996	0.996	0.997	0.997	0.997	0.997	0.983	0.983	0.985	0.985
	15K	**0.998**	**0.998**	**0.998**	**0.998**	0.997	0.997	0.984	0.984	0.983	0.983
	10K	0.998	0.998	0.997	0.997	0.997	0.997	0.985	0.985	0.983	0.983
	5K	**0.998**	**0.998**	**0.998**	**0.998**	0.997	0.997	0.985	0.985	0.983	0.983
	14(7+7)	0.983	0.983	0.983	0.983	0.983	0.983	0.983	0.983	0.983	0.983
CFS	15	0.997	0.997	0.997	0.997	0.997	0.997	0.997	0.997	0.983	0.983

gorithms. The IS will select features with higher class-wise frequency. However, such features might correlate with each other. Thus, these features might have a strong correlation with each other bringing redundant information to the model. In contrast, CFS will try to select a feature set that has as little redundant information as possible. Looking once again in the Tables 10.3 and 10.4 we can conclude, that both feature selection methods performed quite good under our experimental setup while selecting feature sets that are very different to each other.

Important notice. The results from the Table 10.3 is similar to part of the results provided in [7]. This happened because our papers share the same dataset. Also the data collection processes have only minor differences: in this paper we recorded the first million of memory access operations, while methodology of [7] is to record the first million of memory access operations unless a certain stopping criteria is met.

10.5 Discussion and Future work

In this section we discuss our findings and limitations that should be applied to the possible conclusions made based on the presented results. As we were able to see, IS feature selection works faster than IG. The main reason to this is the fact that the selection of features based on its class-wise frequency requires less computations. However, it is important to understand, that all measurements of

time complexity presented in this paper are specific to our conditions (available computational resource, structure of the data, implementation of feature selection algorithms) and might differ in other conditions. The theoretical assessment of the IS feature selection method predicted, that features selected by IS might bring less information about samples and classes than those selected by IG. But the experimental evaluation showed only marginal difference in classification performance. Under our experimental setup, only the amount of features selected by CFS could be considered as a proof of our theoretical assessment. The CFS selected more features from IS-selected feature set to gain similar merit (what resulted in similar classification performance). As we mentioned before, CFS adds features to the feature set until its merit stops growing. These facts show, that features selected by IS possess less information. Thus, on the small feature sets, we need more features selected by IS than those selected by IG. As we compared classification performance of machine learning methods we found, that under certain conditions NB might perform better when using IS-selected features. This fact can be explored more thoroughly in the future work. The method was tested on a nearly balanced dataset, and we selected the equal amount of features to represent both classes. The use of other approach in the selection of the desired amount of features or applicability on the imbalanced datasets is left for the future work.

The IS feature selection method is quite simple in implementation. However, as we discussed in Section 10.3, its applicability limited to the cases where we are interested in the fact of presence of a certain feature in the class. Thus, when features are not binary or discreet, the applicability of IS feature selection is questionable. It is possible, however, to binarize continuous variables [22], but this a separate topic and it is out of scope of this paper. There is also a number of possible improvements and modifications that can be applied to the IS feature selection method in the future. For example, we can decrease the time complexity of IS in the following way. When we calculate class-wise frequencies of features we might limit the search space by the samples that *produce* this feature. Rough estimation suggest, that it may halve the time needed to perform IS feature selection. Another modification that can be implemented in IS feature selection is introduction of the degree of membership to the intersection. For example, a certain feature f might occur in both classes *C1* and *C2*. These classes have m_{C1} and m_{C2} samples respectively. The feature f is present in m^f_{C1} samples of a class *C1* and m^f_{C2} samples of class *C2*. For example, we may exclude feature from the intersection if:

$$\frac{max(\frac{m^f_{C1}}{m_{C1}}, \frac{m^f_{C2}}{m_{C2}})}{min(\frac{m^f_{C1}}{m_{C1}}, \frac{m^f_{C2}}{m_{C2}})} > \epsilon$$

Basically, we keep a feature if it represents ϵ times bigger fraction of samples of one class than fraction of samples of the other class. Such approach may decrease an information loss, but will contribute to the increase of computational complexity of IS feature selection method. And thus will make IS less attractive feature selection method.

It is also important to outline the following observation. IS and IG selected quite different feature sets. Moreover, CFS selected completely different features from those preselected by IS and IG. Nevertheless, classification performance of the machine learning models appeared to be very similar when using different feature sets. This raises the following question: do the mentioned feature selection methods always select the best feature set or do they find *one* of the several similarly good feature sets? This question is left open for the future studies.

10.6 Conclusions

In this paper, we studied the performance of Intersection Subtraction feature selection on malware detection problem. We showed, that with the use of IS feature selection on HDLSS dataset it is possible to correctly classify more than 99% of the benign and malicious samples. The main contribution of this paper is the direct comparison of IS and IG feature selection methods under the same conditions. We found, that most of the features selected by IS and IG are different. The classification performance of the machine learning models trained with the use of quite different feature sets appeared to be very similar. Even though the models trained with IG-selected features showed marginally better performance, the single-thread implementation of the IS feature selection method worked 3.8 times faster than the 16-threads implementation of IG. This makes Intersection Subtraction feature selection attractive when it comes to the analysis of HDLSS datasets. The IS feature selection may help when it is not known yet whether the data is useful for the classification task at all. The number of features might so big, that it is pointless to spend time running more common (also slower) feature selection methods. Thus, with certain above-mentioned limitations, the IS feature selection may be successfully applied to HDLSS datasets.

10.7 Bibliography

[1] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Information security governance: the art of detecting hidden malware. In *IT security governance innovations: theory and research*, pages 293–315. IGI Global, 2013.

[2] Manoun Alazab, Robert Layton, Sitalakshmi Venkataraman, and Paul Wat-

ters. Malware detection based on structural and behavioural features of api calls. 2010.

[3] AVTEST. The independent IT-Security Institute. Malware. https://www.av-test.org/en/statistics/malware/, 2020.

[4] Ahmad Azab, Mamoun Alazab, and Mahdi Aiash. Machine learning based botnet identification traffic. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1788–1794. IEEE, 2016.

[5] Sergii Banin and Geir Olav Dyrkolbotn. Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107–S117, 2018.

[6] Sergii Banin and Geir Olav Dyrkolbotn. Correlating high-and low-level features. In *International Workshop on Security*, pages 149–167. Springer, 2019.

[7] Sergii Banin and Geir Olav Dyrkolbotn. Detection of running malware before it becomes malicious. In *International Workshop on Security*, pages 57–73. Springer, 2020.

[8] Sergii Banin, Andrii Shalaginov, and Katrin Franke. Memory access patterns for malware detection. *Norsk informasjonssikkerhetskonferanse (NISK)*, pages 96–107, 2016.

[9] Max Bramer. *Principles of data mining*, volume 180. Springer, 2007.

[10] Domhnall Carlin, Philip O'Kane, and Sakir Sezer. Dynamic analysis of malware using run-time opcodes. In *Data analytics and decision support for cybersecurity*, pages 99–125. Springer, 2017.

[11] Kevin K Dobbin and Richard M Simon. Sample size planning for developing classifiers using high-dimensional dna microarray data. *Biostatistics*, 8(1):101–117, 2007.

[12] David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture*, 1(2000):32, 2000.

[13] Subhajit Dutta and Anil K Ghosh. On some transformations of high dimension, low sample size data for nearest neighbor classification. *Machine Learning*, 102(1):57–83, 2016.

[14] Jianqing Fan, Fang Han, and Han Liu. Challenges of big data analysis. *National science review*, 1(2):293–314, 2014.

[15] Rosa L Figueroa, Qing Zeng-Treitler, Sasikiran Kandula, and Long H Ngo. Predicting sample size required for classification performance. *BMC medical informatics and decision making*, 12(1):8, 2012.

[16] Ke Gong, Yong Wang, Maozeng Xu, and Zhi Xiao. Bssreduce an o (u) incremental feature selection approach for large-scale and high-dimensional data. *IEEE Transactions on Fuzzy Systems*, 26(6):3356–3367, 2018.

[17] Lars Strande Grini, Andrii Shalaginov, and Katrin Franke. 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*, pages 337–350. Springer, 2018.

[18] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[19] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[20] IntelPin. A dynamic binary instrumentation tool, 2017.

[21] Khaled N Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2015.

[22] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[23] Malay Haldar. How much training data do you need? https://medium.com/@malay.haldar/how-much-training-data-do-you-need-da8ec091e956, 2015.

[24] Olga Ogorodnyk, Ole Vidar Lyngstad, Mats Larsen, Kesheng Wang, and Kristian Martinsen. Application of machine learning methods for prediction of parts quality in thermoplastics injection molding. In *International Workshop of Advanced Manufacturing and Automation*, pages 237–244. Springer, 2018.

[25] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] PortableApps.com. Portableapps.com. https://portableapps.com/apps, 2020.

[28] Python.org. Time complexity. https://wiki.python.org/moin/TimeComplexity, 2020.

[29] Jason D Rennie, Lawrence Shih, Jaime Teevan, and David R Karger. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 616–623, 2003.

[30] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[31] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. Machine learning aided static malware analysis: A survey and tutorial. In *Cyber Threat Intelligence*, pages 7–45. Springer, 2018.

[32] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[33] B Venkatesh and J Anuradha. A review of feature selection and its methods. *Cybernetics and Information Technologies*, 19(1):3–26, 2019.

[34] P Vinod, Vijay Laxmi, and Manoj Singh Gaur. Reform: Relevant features for malware analysis. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 738–744. IEEE, 2012.

[35] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

[36] Kazuyoshi Yata and Makoto Aoshima. Effective pca for high-dimension, low-sample-size data with singular value decomposition of cross data matrix. *Journal of multivariate analysis*, 101(9):2060–2077, 2010.

[37] Kazuyoshi Yata and Makoto Aoshima. Effective pca for high-dimension, low-sample-size data with noise reduction via geometric representations. *Journal of multivariate analysis*, 105(1):193–215, 2012.

[38] Çağatay Yücel and Ahmet Koltuksuz. Imaging and evaluating the memory access for malware. *Forensic Science International: Digital Investigation*, 32:200903, 2020.

[39] Hossam M Zawbaa, Eid Emary, Crina Grosan, and Vaclav Snasel. Large-dimensionality small-instance set feature selection: a hybrid bio-inspired heuristic approach. *Swarm and Evolutionary Computation*, 42:29–42, 2018.

[40] Lingsong Zhang and Xihong Lin. Some considerations of classification for high dimension low-sample size data. *Statistical methods in medical research*, 22(5):537–550, 2013.

Chapter 11

P6: Detection of Previously Unseen Malware using Memory Access Patterns Recorded Before the Entry Point

Sergii Banin, Geir Olav Dyrkolbotn

Abstract

Recently it has been shown, that it is possible to detect malware based on the memory access patterns produced before executions reaches its Entry Point. In this paper, we investigate the usefulness of memory access patterns over time, i.e to what extent can machine learning algorithm trained on "old" data, detect new malware samples, that was not part of the training set and how does this performance change over time. During our experiments, we found that machine learning models trained on memory access patterns of older samples can provide both high accuracy and a high true positive rate for the period from several months to almost a year from the update of the model. We also perform a substantial analysis of our findings that may aid researchers who work with malware and Big Data.

Keywords: information security, malware detection, low-level features, memory access patterns

11.1 Introduction

Detection and analysis of malware is one of the important areas in the information security research[3]. Malware analysis can be divided into two categories:

static and dynamic analysis. While static analysis uses features derived from the file itself, dynamic makes use of the behavioral traces generated when malware is launched. Behavioral or dynamic features can be categorized into high- and low-level features[5]. Low-level features emerge from the hardware of the system: hardware performance counters, opcodes, or memory access patterns are the low-level or hardware-based features. In our paper we utilize dynamic malware analysis using low-level features (memory access patterns). Malware analysis often involves dealing with Big Data: for example, the database table containing memory access patterns of 4000 executables can take several tens of gigabytes[3]. Thus, in this paper, we show how one can analyze the classification performance results obtained after processing the big amounts of data.

It has been recently shown, that it is possible to detect Windows malware based on the behavioral traces produced before the Entry Point (BEP)[6]. It is an important finding since malware malware stopped upon detection BEP can not harm the system where it was launched. However, the authors of [6] used k-fold cross-validation to assess the performance of the machine learning model used for malware detection. Such approach has some limitations: for example, the feature selection made on the full dataset may affect the validity of the classification performance results. As the new malware samples are detected every day, and the amount of newly discovered malware constantly grows[2] it is important to study, how a novel malware detection approach can handle samples that were not involved in training. Thus, we decided to test how memory access traces produced BEP can be used to detect newer, previously unseen malware. We split our dataset into the train set and several time-arranged test sets which will emulate "aging" of the model. We also update the train set with newer malware and observe changes in the performance of the model. To conduct this study we outlined the following research questions. **RQ1:** Is it possible to use memory access traces recorded BEP to distinguish between previously unseen malicious and benign executables? **RQ2:** How long, since the update, the ML model trained on memory access traces recorded BEP can provide a good detection rate? While answering the RQ1 we expect, that the detection of previously unseen malware *should* be possible with the use of BEP memory access traces. At the same time, while answering the RQ2 we expect, that the classification performance and detection rate should be worse the further away in time a train and test set are from each other. However, the 13 months time span of our dataset may affect the results. We assume a detection rate equal or more than 0.95 to be good. While conducting our studies, we found some unexpected trends in classification performance. Since the amounts of data we worked with were very big, we perform analysis which may pose an interest to other researchers working with malware and Big Data. The remainder of the paper is arranged as follows: in Section 11.2 we make a short literature over-

view; in Section 11.3 we describe our methods; in Section 11.4 we describe our experimental setup; we provide the results in Section 11.5; in Section 11.6 we perform an analysis of the findings and outline a need for an additional evaluation which is present in Section 11.7; in the end, we discuss our findings and provide concluding remarks in Section 11.8.

11.2 Background

Here we present a brief overview of the related literature. Malware detection with the use of memory access patterns was first described in [7] by Banin et al. There it has been shown, that it is possible to distinguish between malicious and benign executables with the accuracy of up to 98%. It was [7] where the required amount of memory access operations (1M) and the size of n-grams (96) were shown to be sufficient for such tasks. The technique proposed in [7] was recently extended by Yucel et al. in [26], where authors used memory access patterns to explore the similarity between different malware categories. Later, Banin et al. in [4] showed, that memory access patterns can be successfully used to classify malware into 10 families and 10 types with an accuracy of 78% and 66% respectively. In that paper, it has also been shown, that one needs very few features to perform this type of classification. However, the memory access pattern by itself does not give any information regarding the functionality of the malware to the human analyst. Thus, in their next paper [5] authors performed an attempt to correlate memory access patterns (low-level features) with API calls (high-level features) to bring more context to the human analysts. During the analysis of findings made in [5] authors found, that most of the memory access patterns they recorded emerged from BEP. These findings lead to another work [6], where authors showed that memory access patterns from BEP can be used to detect malware with an accuracy similar to the one achieved with memory access patterns emerged from after the Entry Point (AEP). In particular, they achieved a classification accuracy of more than 99% when distinguishing between malicious and benign executables with a help of only 9 BEP memory access patterns. To the best of our knowledge, memory access patterns have not been tested against previously unseen malware that was not used to train the model. However, many works provide examples of splitting the malware dataset into train and test sets to emulate the detection of previously unseen malware. In [23] authors randomly selected 50% of the dataset to be used as a train set, while the remainder was used as a test set. On the test set, containing roughly 3K malicious and 2.2K benign executable, they managed to achieve accuracy of 100% with the Random Forest algorithm. Similarly, authors of [22] used around 10K malicious and 2.5K benign samples for training, 750 malicious and 610 benign executables for testing, and achieved up to 89% of accuracy. Authors of [15] split their dataset into train and test sets based on the year when samples

were submitted to the VirusTotal[24]. With a train set containing samples from the year 2012, and a test set from the year 2013, they managed to achieve 72% detection rate without a human reviewer and 89% with. The different approach in training and testing was presented in [8]. The authors used a dataset of benign and malicious Android applications from the years 2010-2017. They performed consecutive training on a certain year and testing on the years newer than the one used for training. Their results show, that e.g. precision may both drop or rise as the test set becomes more distant in time from the train set. Similarly, authors of [17] test the performance of their Android malware detection approach on test sets arranged on the monthly basis. In their paper, it is possible to observe the decay of the performance of the model trained on samples from the year 2014 as test sets become more distant in time from train one. Authors of [19] elaborate on the good practices for building the relevant malware dataset and conducting time-aware malware studies. Among the other recommendations they give, there are several that we follow in our research: describe an experimental environment, OS, network connectivity, etc.; describe the dataset; provide family names of the malware samples in use. Authors of [1] state, that train and test sets combination that is built based on a certain time metric will generally yield the performance worse than the one of a random split (e.g. k-folds cross-validation).

11.3 Methodology

This section is dedicated to the description of the methods used in this paper. Our choice of methods is based on findings made in [7],[5] and [6]. We begin with the description of our data collection process. Then we explain the way we preprocess and select features. Later, we explain the way we split our data into train and test sets. In the end, we describe the evaluation metrics and machine learning algorithm used in this paper.

11.3.1 Data collection

Our data collection is based on the BEP-AEP approach that was first presented and described in [6] and analysis of memory access patterns first used in [7]. The key concept involves splitting the behavioral trace of the process into two main parts: the one that occurs before the Entry Point (BEP) and after (AEP). In this paper, we focus only on the trace produced BEP. With a help of Intel Pin binary instrumentation framework [11] we record the memory access operations produced by the process from the moment it starts. We record only the type of memory access operation: *R* for read and *W* for write. We record the sequence of the first 1M of memory access operations. However, if the sample does not produce 1M million memory access operations BEP we still keep its data, thereby making our

experiments more realistic. Similarly to [6] we stop recording the trace as soon as the execution flow reaches the first instruction from the main module of executable.

11.3.2 Data preprocessing and feature selection

The sequence of up to 1M of memory access operations is recorded for each sample in the dataset. Each sequence is later split into the set of overlapping n-grams of the size *n=96*: memory access patterns. Each next n-gram overlaps the previous one on *n-1* operations. These 96-grams later serve as features for ML algorithms. For classification purposes, each feature describes the presence or absence of a certain pattern in a trace of a sample: it takes value *1* if a pattern is present in a trace of a sample and *0* if not. When working with memory access traces the amount of features (unique patterns) is always big and can reach millions of features[3][5][6][7][4]. It is unlikely, that all features contain valuable information. Moreover, regular machine learning packages are not suitable to work with data of such a high dimensionality. So it is important to perform feature selection before feeding the data into ML algorithm. To reduce the feature space, we perform a two-step feature selection process that was described in more detail in [6]. First, we select 50K best features from the training set based on their Information Gain (IG)[12]. Later, we use these 50K features to select the best feature subset using Correlation-based feature selection (CFS)[9] from ML package Weka[10]. CFS searches for the best subset of the given feature space and selects features that have a high correlation with classes in the dataset but low correlation between each other. We use CFS with the default for Weka parameters. With current implementations, it is challenging to use CFS on the full feature set, since performing the CFS requires a calculation of correlation matrix between all features, the process that requires an infeasible amount of time and computational resources when we are talking about millions of features. It is important to note, that CFS adds features to the feature set until the merit of the feature set stops growing more than a certain threshold[9]. Thus, it is challenging to choose the desired amount of features to be selected by CFS.

11.3.3 Splitting the dataset

Different authors utilize different approaches to test their malware detection method on previously unseen malware. Some simply split the dataset into train and test sets. While others make their dataset time coherent: samples arranged based on a certain time property. This allows emulating the updates of the models with time. In this paper, we arrange our dataset based on the first seen time from the VirusTotal (VT)[24]. There are not many other sources of time-related information when talking about the Windows executables, as compilation time available in PE header can be forged[21]. We split our dataset into bins based on the month the malware samples were first seen on VT. Note, that finding enough benign samples

from a certain period of time is a quite challenging task. Thus, even though we also arrange benign samples based on VT data, we add them into bins based on their position on the benign timeline and the number of malicious samples in the same bin (see Section 11.4.2). This approach allows us to make training and test sets to have almost equal amounts of malicious and benign executables. We consider samples that are present in a certain bin to be *unseen* to those present in the older bins. Thus, newer benign and malicious samples do not contribute to the model and do not affect the feature selection process. We try to keep the amount of malicious and benign executables in bins equal. We also keep all malware samples in the bins regardless of the malware family they belong to. We decided to use our dataset as is since samples from the same family evolve over time and the distribution of families across the bins is not uniform what adds more realism to our experiments.

11.3.4 Evaluation

To check the applicability of memory access traces recorded BEP for the detection of previously unseen malware we train the ML model on the training set that consists of one or several bins and separately test it on the bins that were not used for training. We iteratively increase the training set by adding newer bins into it. As an ML algorithm we have chosen Random Forest (RF) algorithm from Weka[10] package, since it has shown one of the best results in [6]. RF constructs a number of decision trees, which are used for the classification. We use RF with default, for Weka, parameters where the number of trees is 100. To evaluate the quality of the models we use several metrics. Accuracy, as the amount of correctly classified samples. True positive rate (TPR), as the amount of actual malware that is detected as malware (detection rate). False positive rate (FPR), as the amount of actual benign executables classified as malware (potential false alarms in the system). In this paper, we show, how these metrics change with the increased amount of time passed since the "last update" of the model (latest bin added to the training set).

11.4 Experimental setup

In this section, we describe our experimental environment, provide details about our dataset, and explain our experimental flow.

11.4.1 Experimental environment

When using dynamic malware analysis, it is important to avoid the influence of changes in the experimental environment and ensure equal launching conditions for all samples. It is also important to isolate malware so that the host system or network are not affected by the malicious behavior. To ensure security and

repeatability we use Virtual Box virtual machine (VM) with Windows 10 guest operating system. The VM was isolated from the internet. All our VMs were launched on the Virtual Dedicated Server (VDS) with 4-cores Intel Xeon CPU E5-2630 CPU running at 2.4GHz and 32GB of RAM with Ubuntu 18.04 as a host operating system.

11.4.2 Dataset

In this subsection, we describe the content of our dataset and explain how the dataset is split into bins which are later used to construct different train- and test-set combinations. This dataset was previously used in [6] and [3]. Our dataset can be divided into two main parts: benign and malicious executables. Malware samples were obtained from *VirusShare_00360* collection from VirusShare[25]. *VirusShare_00360* contained 65518 samples, out of which 2973 were PE executables. For each malicious sample, we got a report from VirusTotal(VT) [24]: an online malware analysis tool that also allows seeing how different Anti-Virus engines react to a certain sample. We left only samples that were recognized as malicious by at least 20 engines. In the final dataset, we included samples that belonged to the 10 most common families: Fareit, Occamy, Emotet, VBInject, Ursnif, Prepscram, CeeInject, Tiggre, Skeeyah, GandCrab. According to the VT reports, resulted samples were first seen (first submission date) between March 2018 and March 2019. Not all the samples were launched successfully, so the amount of malware samples that generated traces is 2005. Benign samples were downloaded from Portable Apps [18] in September 2019 and is a set of free Portable applications. It contains various software such as graphical, text, and database editors; games; browsers; office, music, audio, and other types of Windows software. According to the VT, benign samples were first seen between December 2006 and December 2019. Some benign samples were first seen on VT after their download date because it was we who first uploaded them to the VT to check whether they are truly benign. We left only samples that were not recognized as malicious by any of the AV engines available on VT. After running the samples 2098 of them produced traces.

As it is outlined in the literature[19], it is important to present the distribution of malware categories in the dataset. In Table 11.1 we show the number of samples that belong to each of the families. As it is possible to see, the dataset is not balanced in terms of malware families. However, we did not polish this aspect of our dataset since we only cared about samples being benign and malicious.

We looked at two possible splitting approaches to create time-ordered subsets from the original dataset. In the first approach, we split malware into 13 bins based on the month they were first seen on VT. We also included benign samples into the monthly bins based on their VT first seen date. However, this approach resulted in

Table 11.1: Distribution of malware families in the dataset

Total	Fareit	Occamy	Emotet	VBInject	Ursnif	Prepscram	CeeInject	Tiggre	Skeeyah	GandCrab
2005	573	307	196	164	162	143	127	117	115	101

Table 11.2: Amount of benign and malicious samples in bins.

Bin #	0	1	2	3	4	5	6	7	8	9	10	11	12
Benign	64	119	174	88	167	66	39	130	149	264	214	307	317
Malicious	64	119	174	88	167	66	39	130	149	264	214	307	224

highly imbalanced subsets, where the malware to goodware ratio sometimes was as high as 50 to 1. It is quite problematic to find the desired amount of benign samples from the desired time period. So we decided to discard the first approach due to this imbalance. Instead, we decided to split benign samples into 13 time-ordered bins and align the number of samples in them according to monthly bins created from malware samples. Each bin would contain as many benign samples as the corresponding malicious bin. Only the last bin would contain more benign samples since the amount of benign samples is bigger in the original dataset. This way, every next bin will have benign samples that are newer than those in the previous one. In Table 11.2 we present the amount of benign and malicious samples that were put into each of the 13 resulting bins. As we can see, the bins have different amounts of samples in them. To describe our bins in an even more detailed way, in Fig. 11.1 we present the distribution of the above-mentioned malware families among the malware samples in each of the 13 bins. As we can see, malware families are not evenly distributed among the bins.

Having 13 bins with equal malware to goodware ratio except for the bin #13 we use them to construct train and test sets that are used to training and test ML models. The training set is a combination of one or several consecutive bins. From here the training sets will be named in the following way: training set based on the bin #0 is called *T0* while training set built from bins #0 to 7 is called *T0_7* and so on. For each training set, we have one or more test sets, made of the remaining bins that were not used in the construction of the train set. For example, for the training set *T0_10* we will have two test sets consist of the bin *11* and bin *12* respectively. With such approach, we obtain 12 combinations of train and test sets. As we previously described the distribution of malware families within the bins (which now also represent test sets) we now use Fig. 11.2 to show the distribution of the malware families within train sets. As we can see, after the train set *T0_1* the distribution of families within the train sets begins to stabilize itself and becomes quite similar between the train sets closer to the last one.

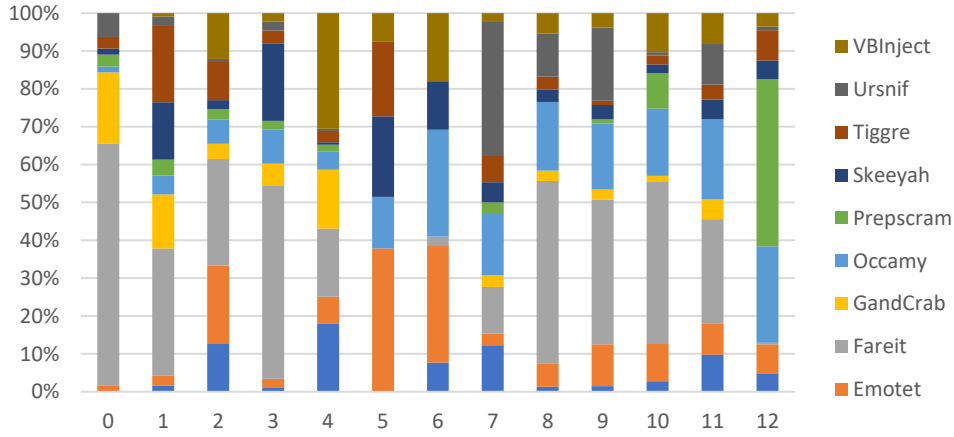


Figure 11.1: Distribution of malware families among the malware samples within each of the 13 bins

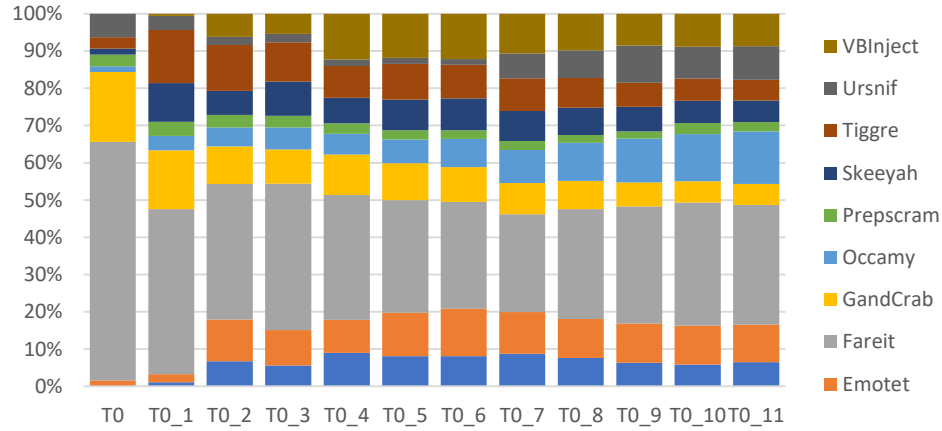


Figure 11.2: Distribution of malware families among the malware samples within train sets

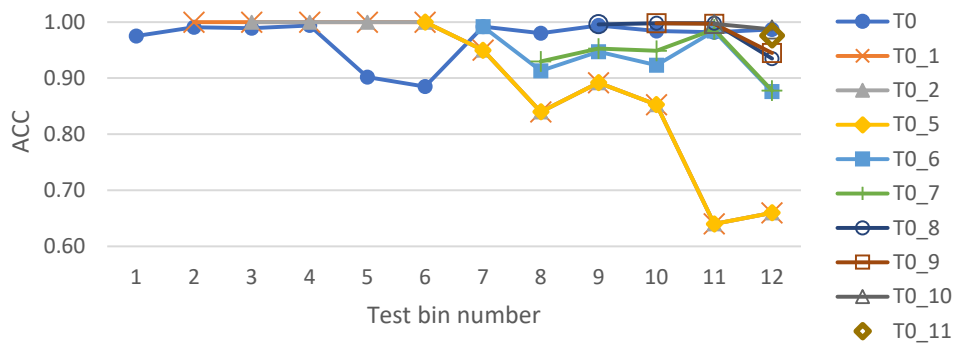
11.4.3 Experimental flow

Every sample from our dataset is first copied to the clean snapshot of the VM. Then, we launch it together with a customized Intel Pin tool. The Intel Pin tool records memory access operations and stores them into a trace file. The trace is later copied to the host system, and the VM is reverted to the previous state. It is important to note, that the benign executables from PortableApps were copied to the VM together with the content of their folder. This approach allowed us to provide more realistic results since benign executables often require additional

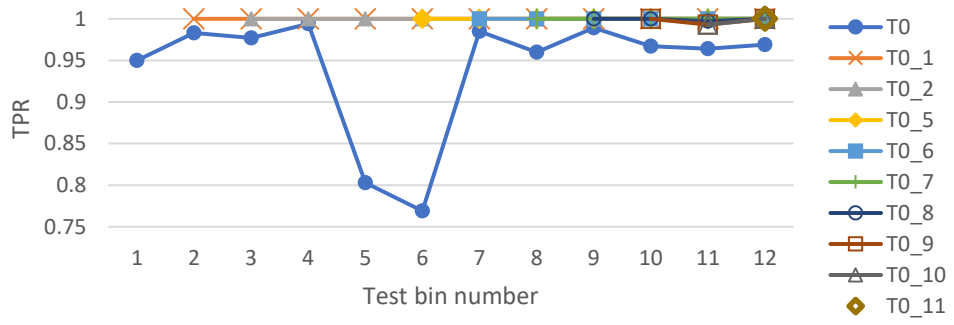
resources to be launched properly.

11.5 Results

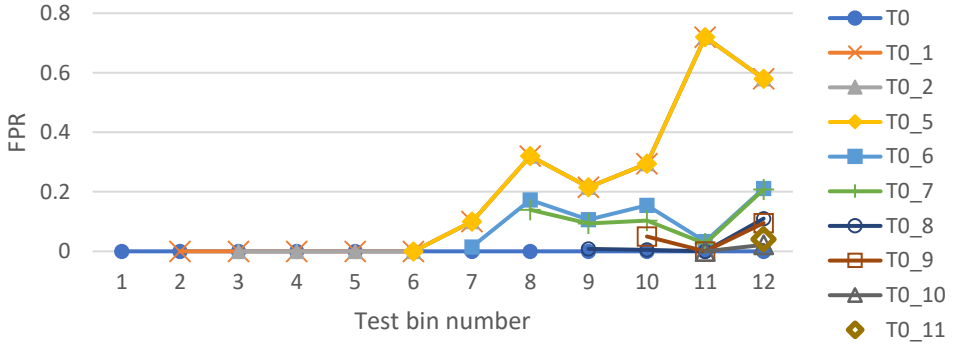
In this section, we provide the detection performance achieved with our approach by RF algorithm. As we outlined in Section 11.3 we decided to test our malware detection approach with help of the RF ML algorithm because it has previously shown good classification performance with a similar type of features. In Fig. 11.3 we show the accuracy, true positive rate, and false positive rate of RF algorithm. For table data see Appendix Appendix A. Each line on the chart represents a certain evaluation metric of the ML model trained on a certain train set. Each point of the line is the value of the metric obtained while attempting to classify samples from one of the test bins. Before looking into the achieved results it is important to mention, that accuracy, TPR, and FPR achieved by models trained on the sets *T0_1 - T0_5* match for all the corresponding test sets. Thus, corresponding points and lines on the charts merge. To simplify our charts we omit results achieved with *T0_3 - T0_4* since they are the same as those achieved with *T0_1, T0_2* and *T0_5*. First, let's take a look at the accuracy achieved by the RF algorithm. As it is possible to see from Fig. 11.3a we can outline two main trends in the classification performance. The first trend shows, that the further in time a test bin from the train set - the lower the classification accuracy. This trend, however, has several exceptions. First of all, the model trained on *T0* shows a drop in accuracy for test bins 5 and 6. For other test bins, it has a quite stable accuracy while showing minor improvements (e.g. accuracy on bin 7 is higher than the one on the bin 0). Lastly, models trained on *T0_6* and *T0_7* show a significant spike of accuracy on the *T0_11* and drop on the last test bin. The second trend shows, that the closer train set to the test set (the more up-to-date it is) the higher classification accuracy on the test set becomes. For example, accuracy on the test set 12 improves when the train set is updated with newer bins. Now let's look at the TPR and FPR showed by the RF algorithm. As we can see, most of the test bins are classified with TPR that is equal to or higher than 0.95. Moreover, models trained on sets *T0_1 - T0_7* always show TPR of 1 for all test bins. It means, that such a model will not miss any of the previously unseen malicious samples. However, from the FPR chart, we can also see that some models (especially *T0_1 - T0_5*) show an increasing amount of false positives for test bins that are further away from the training set. FPR can become as high as 0.72 which in reality will result in a significant amount of false alarms and may seriously affect the operations of the system that uses such models in AV solutions. It is also important to mention, that model trained on *T0* shows 0 FPR for all of the test bins while missing some of the malware samples. The overall trend of FPR and TPR is the following. Most of the models, while keeping high TPR (detection rate) over time develop higher



(a) Accuracy



(b) True positive rate



(c) False positive rate

Figure 11.3: Performance of RF algorithm

FPR which clearly shows that even the models with good detection capabilities have to be regularly updated.

While acquiring the data present on the Fig. 11.3a, 11.3b and 11.3c we derived several interesting findings. The performance of the models does not change when trained on sets *T0_1 - T0_5* as they show very low accuracy towards the last test bins. But when the model is trained on the sets *T0_6* and beyond, the

accuracy rapidly improves. We also noticed, that when a train set is changed from *T0* to *T0_1* the model starts performing worse for many of the test bins. It is quite counter-intuitive since normally we would expect a better performance of the model that used more recent samples (bigger and updated training set) to train. The rapid improvement of accuracy and a counter-intuitive difference in performance between models trained on *T0* and *T0_1* raised our attention and we analyze these findings in Section 11.6.

As we can see, the performance of most of the models built with the use of memory access patterns recorded BEP degrade over time. Some models degrade more than the others, acquiring high FPR. But at the same time, the TPR of most of the models remains relatively high, thus even an outdated model built with BEP memory access traces will protect the potential system over a long period of time (while producing a high amount of false alarms).

11.6 Analysis

When we discovered a rapid improvement in the model's performance with the change of train set from *T0_5* to *T0_6* and significant difference between models trained on *T0* and *T0_1* we had several hypotheses about the reason for these changes.

11.6.1 Influence of families

The simplest idea was the influence of the malware families' distribution in training and test sets. For example, different distribution of families in train sets could lead to models biased towards a certain category of malware. However, if looking into Fig. 11.2 and 11.1 from Section 11.4 we may see, that the distribution of families in training sets *T0_5* and *T0_6* are almost identical. The family distribution does not also explain the difference in performance between models trained on *T0* and *T0_1*: it is easy to see, that family distribution in *T0_1* is closer to e.g. test bin 9 than the one in *T0*. Thus we rejected this hypothesis.

11.6.2 Influence of features

The other potential reason for the model performance changes could be the features. As we are not using the entire feature set and using a two-step feature selection process it could be, that the features we select as well as their amount can affect the potential performance of the model trained on data built with such features. First of all, let's take a look in Table 11.3 where the amounts of features selected by CFS for each of the train sets are present.

First of all, it is easy to see that train sets can be grouped into three categories based on the number of features selected on them by CFS. In the first group, it will be a single set *T0*: 555 features were selected from it. In the second group there

Table 11.3: Amount of features selected by CFS feature selection method for all train sets

Train Set	T0	T0_1	T0_2	T0_3	T0_4	T0_5	T0_6	T0_7	T0_8	T0_9	T0_10	T0_11
Features	555	133	135	136	134	134	134	8	10	11	11	10

will be sets *T0_1 - T0_6* with the number of features ranging from 133 to 136. And in the third group there will be sets *T0_7 - T0_11* with the number of features ranging from 8 to 11.

The amounts of features selected from the sets in the third group did not surprise us, since they are quite similar to what can be seen in [6]. It was already known, that in some cases we need very few of the BEP memory access patterns to achieve 0.99 classification accuracy.

When talking about *T0*, 555 is a quite high amount of features, since *T0* has only 128 samples. It is generally considered in the literature, that fewer features in the dataset improves the performance of ML algorithms [7] [4] [6] [12]. In some articles authors suggest using *the rule of 10*: to train a good performing ML model, it is advised to have ten times more training samples than features [13]. However, in our case, a bigger amount of features allowed to eliminate false positives and contributed towards relatively stable TPR and accuracy along the test sets. Based on the experience from [6] we were surprised by the fact that CFS has selected so many features for a relatively small dataset. So we calculated IG for all of the 555 features selected from *T0* and found, that 531 of them had IG of *1*. For a two-class dataset, it means that each of these features can be solely used to correctly classify all samples of the train set. As CFS stops adding features to the feature set when the merit of the set stops to increase, it becomes clear that the high amount of selected features is due to their high quality. Even if they correlate with each other, there is no way to distinguish between features with exact same values for all samples if they carry a lot of information.

While evaluating the second group, we found, that the number of features selected for the sets *T0_5* and *T0_6* is the same. So the number of features has no influence over the rapid increase in model performance. Thus we decided to check how feature sets change with the change of the train sets. To observe changes in feature sets we built a Table 11.4. In this table, each row and column is named *T0_X_N* where *T0_X* represents one of the training sets, while *N* is the number of features selected from this training set. Each cell of the table shows the amount of features common between the feature sets whose row and column cross in this cell.

It is easy to see, that feature sets from *T0* to *T0_6* share many common features between them. However, as it was shown in Section 11.5, model trained on *T0* show better performance on the last 6 test bins than models trained on *T0_1-T0_-*

Table 11.4: Amount of common features between the feature sets.

	T0_555	T0_1_133	T0_2_135	T0_3_136	T0_4_134	T0_5_134	T0_6_134	T0_7_8	T0_8_10	T0_9_11	T0_10_11	T0_11_10
T0_555	555	126	126	127	126	126	126	2	2	3	3	3
T0_1_133		133	128	126	126	126	126	1	1	1	1	1
T0_2_135			135	127	127	127	127	1	1	1	1	1
T0_3_136				136	127	127	127	1	1	1	1	1
T0_4_134					134	128	128	1	1	1	1	1
T0_5_134						134	128	1	1	1	1	1
T0_6_134							134	1	1	2	1	1
T0_7_8								8	0	0	0	0
T0_8_10									10	4	2	0
T0_9_11										11	1	1
T0_10_11											11	5
T0_11_10												10

6. This can be a sign of the drawback in our feature selection approach, as it can not select the same features from e.g. train sets *T0* and *T0_1*. Let's now look into the feature sets *T0_5_134* and *T0_6_134*, a place where the RF model gets rapid improvement in classification accuracy. These feature sets share 128 of 134 features. However, the latter allows for higher classification accuracy. We examined 6 "old" unique features from *T0_5_134* and 6 "new" from *T0_6_134* in terms of the information they carry. We found, that 5 out of 6 of old and new features have the same IG in their respective train sets. The remaining features have their IG different in the 5th digit after the decimal point. We believe, that such an insignificant difference in feature set quality could not result in the improvement of classification accuracy that we've seen in Section 11.5. It is also worth to mention, that feature sets from *T0_7* to *T0_11* share more common features with *T0* then with *T0_1-T0_6*. And the RF models trained on them generally perform better than those trained on *T0_1-T0_6*.

Another thing we could check emerges from the nature of our features and the way we process experimental data. As we explained in Section 11.3 the feature takes value *1* when a certain memory access pattern is generated by a sample and *0* otherwise. What can happen, that the majority of selected features take value *1* for more samples of one class that of another. This means, that the feature set *represents* behavior of a certain class. In other words, one class can be described by the presence of certain memory access patterns while another by *absence* of such. And since malware and benign software evolve over the time it might happen,

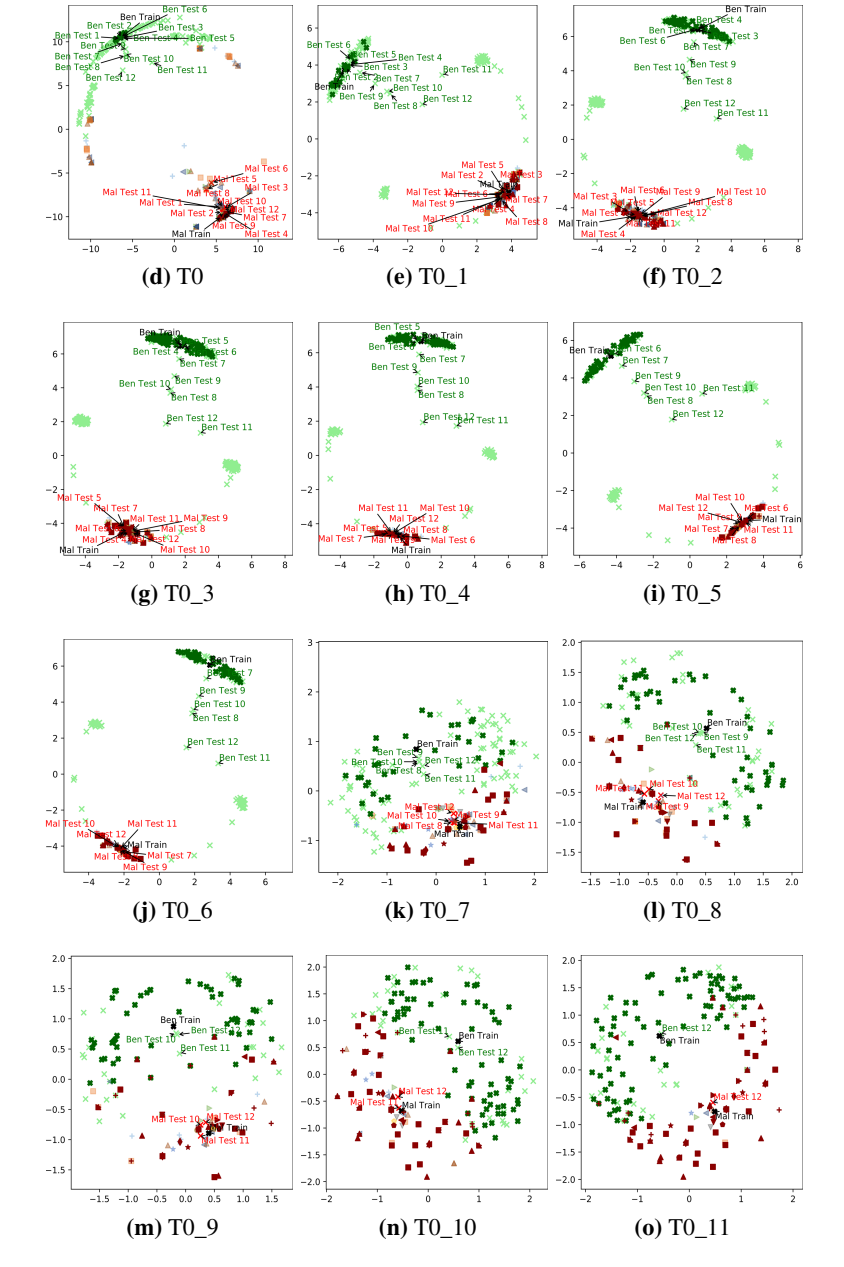
Table 11.5: Proportion of features that represent one class more than another.

Feature Set	T0_555	T0_1_133	T0_2_135	T0_3_136	T0_4_134	T0_5_134	T0_6_134	T0_7_8	T0_8_10	T0_9_11	T0_10_11	T0_11_10
Mal	0.771	0.030	0.037	0.037	0.022	0.022	0.022	0.125	0.2	0.273	0.364	0.5
Ben	0.229	0.970	0.963	0.963	0.978	0.978	0.978	0.875	0.8	0.727	0.636	0.5

that newer samples of a certain class will start generating patterns that were not generated by samples of this class at the time of training (in the training set) and vice versa. Thus, we decided to explore how selected features represent classes in train sets. To do this we counted the proportion of features that represent more malicious or more benign samples. Basically, we found the number of features that take value *1* in more samples of one class than in another. In Table 11.5, values in column Mal reflect the portion of the entire feature set features that take value *1* more often in malicious samples, while column Ben reflect similar values for the benign class. As we can see from this table, feature sets that contribute to the good performance of RF models (*T0,T0_7-T0_11*) have a smaller imbalance between the number of features that represent malicious and benign classes than other feature sets. However, the feature set *T0_6* stands out. It has one of the highest imbalances but allows for better performance than feature sets with similar feature balance. Thus, we can not conclude that the way features represent classes affects the performance of models. We also trained RF models with 50K best features (see Section 11.3) but the results were the same as with CFS-selected features.

11.6.3 Influence of feature space

As we found, that feature qualities have no direct influence on the classification performance we decided to check whether the entire feature sets can be a reason for the classification performance we observed in Section 11.5. Samples in a dataset can be considered as points in the multidimensional space, where dimensionality is defined by the number of features and coordinates of the point are the values of features for the particular sample. It is known, that in general the further away in given feature space samples of a different class from each other - the easier it is to distinguish between them[12][20]. Some ML algorithms, e.g. k-Nearest Neighbors or Support Vector Machines, use distances between samples directly. But even if the distance measure between samples is not used in the ML algorithm, two samples of two different classes that have the same coordinates (are in the same point of feature space) are impossible to distinguish from each other. So we decided to visualize how selected features allow to separate samples



(d) T0 **(e)** T0_1 **(f)** T0_2

(g) T0_3 **(h)** T0_4 **(i)** T0_5

(j) T0_6 **(k)** T0_7 **(l)** T0_8

(m) T0_9 **(n)** T0_10 **(o)** T0_11

(p) Legend

Figure 11.3: Distance-preserving projection of train and test samples from multidimensional feature spaces into the two dimensional plane.

of different classes. It is impossible to draw a space which dimensionality exceeds 3. However, there is a way to reduce the dimensionality of a dataset and draw it on the 2D plane while keeping relative distances between point intact: multi-dimensional scaling[14] (MDS). With a help of MDS, it is possible to visualize how samples from the multidimensional dataset are located relative to each other on the two-dimensional plane. Using MDS implementation from *scikit-learn* [16] Python package we built Fig. 11.3. In this Figure, each subfigure is an illustration of the location of the train and test samples in the feature space of a certain training set. For example, in Fig. 11.3f we can see how samples are located in the feature space of the *T0_2*. On each of the subfigures we depicted the following elements (for colors and shapes see the legend on Subfigure 11.3p):

- Train samples of benign and malicious classes. Train samples of a malicious class have different shapes according to their family (see Legend) but use the same dark-red color.
- Test samples of benign and malicious classes. Test samples of a malicious class have different shapes and colors according to their family (see Legend).
- We have also marked centers of malicious and benign parts of the train and each of the test sets. A *center* here is a point that has coordinates equal to the mean of the samples in the group: it can be considered as a centroid of the corresponding cluster. For example in Fig. 11.3o a point Named "Ben Test 12" is a center of benign samples for test set 12.

From Fig. 11.3 it is possible to understand some of the classification results. For example, in Fig. 11.3d we can see, that benign parts of train and test samples lay relatively close to each other, while several groups of malicious test samples are located closer to benign samples than to the malicious train set. This explains 0 FPR achieved by ML model on this train and test sets combination and non-ideal TPR since some malicious samples are closer to the benign part of the train set than to the malicious one. On its turn, Fig. 11.3e shows why FPR grows and accuracy drops for a model trained on *T0_1*: both benign samples and centers of benign parts of test sets become closer to the malicious train part over the time. We may also observe in Fig. 11.3k- 11.3o that malicious and benign sets slightly overlap, but still quite distinguishable. On the other hand, a comparison of Fig. 11.3i and 11.3j does not explain the rapid improvement in the classification accuracy of the models. Moreover, the relative positioning of the benign and malicious sets almost doesn't differ[1]. At this point, we had to conclude, that feature spaces analysis does

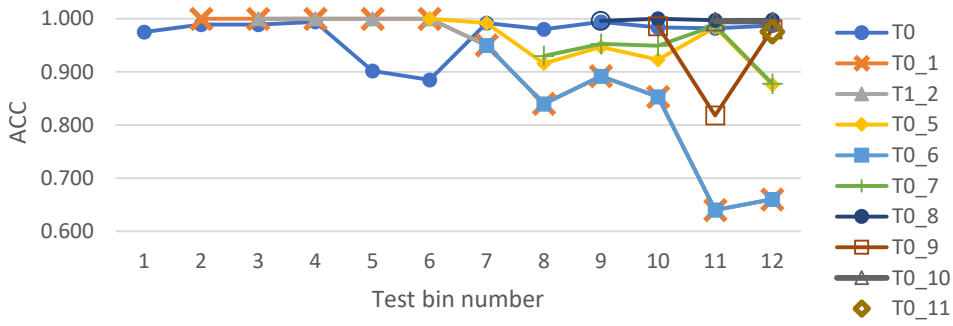
[1]It is important to note, that Fig. 11.3i and 11.3j look rotated against each other only because we had no control over how exactly the points from multidimensional space are placed on the two-dimensional plane by the MDS algorithm.

not help to understand the classification accuracy difference between *T0_5* and *T0_6* models. We must also admit, that the counter-intuitive difference between the performance of models trained on *T0* and *T0_1* can not be explained with this approach. Our next hypothesis about the unexpected classification performance was about the potential limitations of the RF algorithm. Thus, we decided to train models with several different ML algorithms and compare their performance to the RF algorithm. We decided to find out whether it is possible to obtain models that can detect malware with the use of BEP memory access traces better.

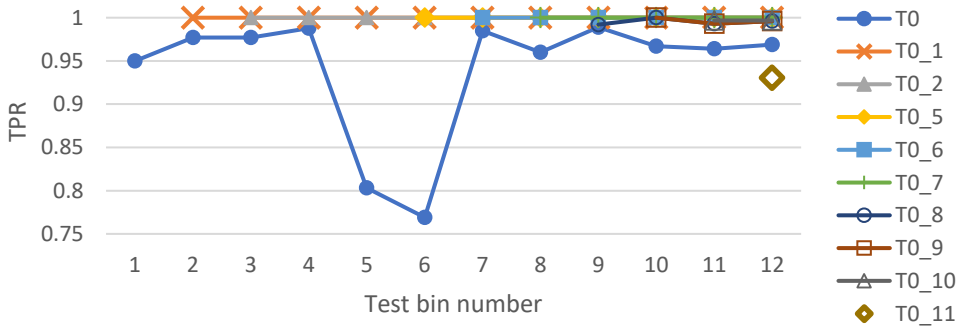
11.7 Additional evaluation

This section is dedicated to answering the question "Is it possible to classify malicious and benign samples better than with use of RF algorithm?[2]". When we decided to test classification performance of other algorithms we first checked the k-Nearest Neighbors(kNN) algorithm as it performed quite well in [6],[4] or [7]. However, it showed very similar to RF performance so we do not present the results achieved by kNN. The next algorithm we decided to check was the J48 (Decision Trees) algorithm from Weka [10]. The main difference between RF and J48 is the *number* of trees that are used. By default, RF from Weka uses 100 trees while J48 builds a single tree. In Fig. 11.4 we present accuracy, TPR, and FPR of J48. For table data see Appendix Appendix B. First of all, J48 trained on sets *T0-T0_4* show exactly the same performance as RF. On the Fig. 11.4, similarly to Fig. 11.3, we omit results on *T0_3* and *T0_4* because the results on the remaining test sets do not change. However, when we look at the performance of J48 trained on *T0_5* we can easily see, that J48 performs better with this train set than RF. Moreover, J48 trained on *T0_5* classifies test sets 7-11 with exactly the same accuracy as RF trained on *T0_6*. At the same time, J48 trained on *T0_6* classifies test sets 7-12 with exactly the same accuracy as RF trained on *T0_5*. For the rest of the training sets, J48 and RF behave mostly similarly but there are some visible differences. E.g. J48 trained on *T0_8* performs better than RF, while RF trained on *T0_9* performs better than J48. As RF is a set of many decision trees, we explored how the number of trees affects the performance of RF. It was found, that the number of trees of 4 or less RF performs the same way J48 does. This means that we could have observed a case of overfitting of the Random Forest algorithm when trained on *T0_5*. So far we showed, that there can be specific cases when one tree-based algorithm outperforms another. But in an attempt to improve the classification accuracy of models trained on *T0_1-T0_5* we decided to utilize Locally-Weighted Learning (LWL) algorithm from Weka. LWL is a combination of kNN and any other ML algorithm that supports weighted learning and has relatively low training time.

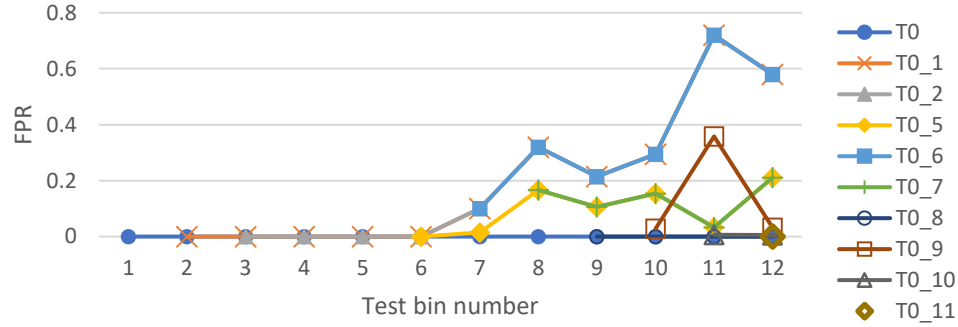
[2]Under the current conditions: same dataset and features.



(a) Accuracy



(b) True positive rate



(c) False positive rate

Figure 11.4: Performance of J48 algorithm

The basic principle of LWL is the following: to classify a test sample, at the time of classification LWL trains an ML model with a use of k train samples that are close to the test sample. The k samples are weighted according to their distance. This way every test sample is classified by a separate ML model. The default ML algorithm that is used in LWL in Weka is Decision Stump (DS). Decision stump is a simple decision tree that consists of only one node. In the Fig. 11.5 we present

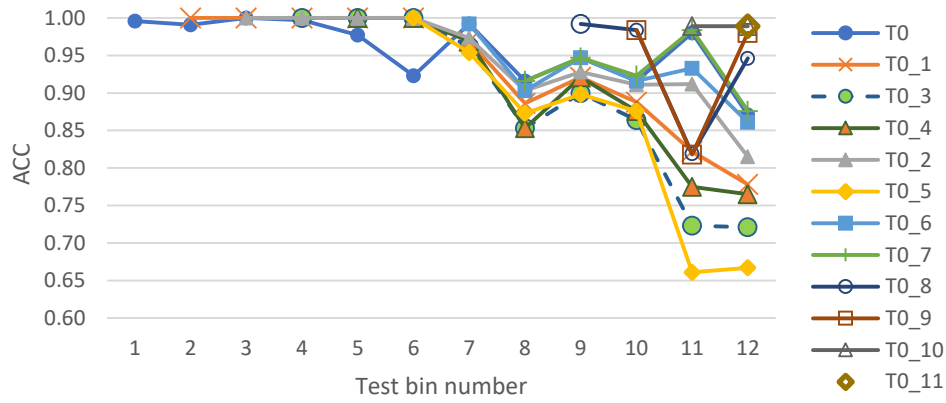
the performance of the LWL algorithm. For table data see Appendix Appendix C What is easy to see in Fig. 11.5b is the improvement of TPR if compared to RF and J48. We can also observe the FPR values (Fig. 11.5c) became more diverse between the models trained on different train sets. From the accuracy chart (Fig. 11.5a) we can see, that the model trained on *T0* performs not as well as similar models of RF and J48. Its performance almost matches the one of a model trained on *T0_7* on the test sets 8-12. We also observe a decline in the performance of the models *T0_1 - T0_5* if compared to *T0*. However, the LWL models *T0_1 - T0_4* perform better than those of RF and J48: accuracy is higher, while FPR is lower. So we were able to improve the performance of some of the low-performing models by changing the ML algorithm. But the LWL trained on *T0_5* performs almost as bad as the RF and significantly worse than J48. When switching to the *T0_6* model we see the rapid improvement of the performance that is similar to the one we have seen with RF.

As we were able to see, for some combinations of train and test sets it is possible to improve classification performance by choosing the different ML algorithms. Thus, we can answer positively to the question outlined at the beginning of this section. Some algorithms will perform better under certain conditions while worse under the other. But the final choice of the ML algorithm is always up to the developers of the potential AV system and should be based on the requirements of the system in interest.

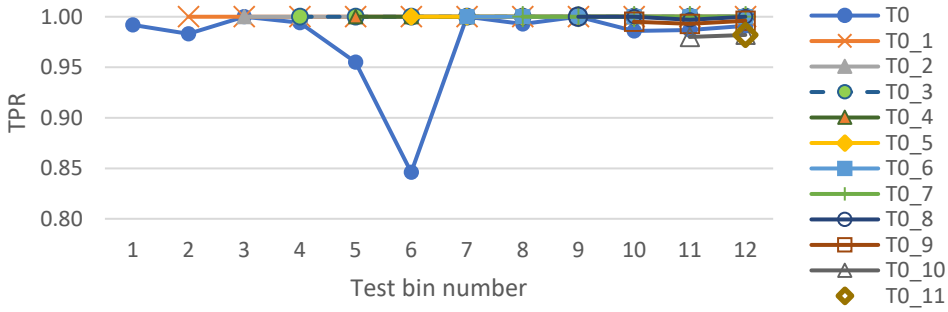
11.8 Discussion and Conclusions

In this section, we discuss our findings, present the conclusions, and outline possible improvements that can be implemented in future work.

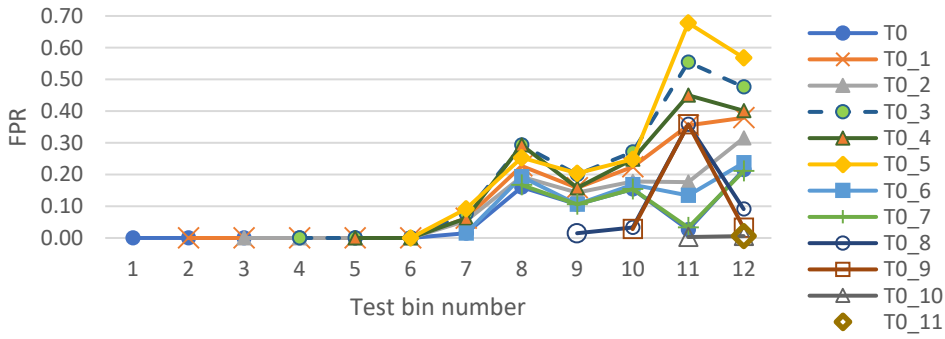
In this paper, we have shown, that behavioral traces recorded before the Entry point have the potential to be used for the detection of previously unseen malware. It is an important finding since malware detected BEP has no chance to harm the system even though it was launched. The results presented in Sections 11.5 and 11.7 show, that we can answer *yes* to the *RQ1* outlined in Section 11.1. The memory access traces recorded before the Entry Point can be successfully used to distinguish between previously unseen malicious and benign executables. To answer the *RQ2* we have also shown, that most of the ML models trained on the BEP memory access traces can provide a good detection rate ($TPR > 0.95$) for the significant periods of time since the update of the model. Some models provide high TPR for a period of at least 11 months. But it is important to remember, that they also tend to develop high FPR which is a clear sign of the need for regular updates of the model since high FPR will disrupt operations of the system by raising a lot of false alarms. Thus, we have to conclude that memory access traces recorded BEP can be used for malware detection, but such an approach has its



(a) Accuracy



(b) True positive rate



(c) False positive rate

Figure 11.5: Performance of LWL algorithm

limitations that should be taken into account.

We have also performed an attempt to explain some of the cases of difference in the performance of ML models. Under our approach, we were not able to show that features or feature spaces influence the classification performance of the models. But we were able to show, that in some cases (for certain combinations of

train and test sets) some ML algorithms perform better than the others. But it is still important to pay attention to the TPR and FPR when making a choice of the ML algorithm to be used in a real system. Some trends of the classification performance results remain similar between different ML algorithms: e.g. models trained on *T0* outperform those trained on *T0_1 - T0_6* on the last 7 test bins (true for RF, J48, LWL); models trained on *T0_5* are among the worst-performing models (RF, LWL); model trained on *T0* shows a drop of performance for the test bins 5 and 6 (RF, J48, LWL). Based on these findings we have to conclude, that such performance of models can be a sign of a potential weakness of our BEP memory access patterns approach. In future work, one may try to understand, whether this is a weakness in the use of memory access patterns or the fact that we are focusing on the BEP activity. To do this, different types of features and features recorded AEP may be used on the same dataset. It may also be the result of some special properties of our dataset, that we had no control over since we used all of the available samples. So in future research, one may use our approach on the different, potentially larger, and more diverse dataset. It may also be useful to perform the analysis of the misclassified samples[19], as it may help to understand the classification performance as it was shown in [4]. During the analysis phase we also observed a case of overfitting of Random Forest algorithm. We believe, that this finding has a potential to be investigated by machine learning researchers working on improving of understanding the performance of common ML algorithms.

We also believe, that our paper provides an important example of the analysis of the classification performance. Analysis of subcategories, the influence of the amount and quality of features in the updated feature set, and graphical analysis of feature space can help other researchers to understand their results. We think, that such approach can be used not only in malware analysis but in many areas where ML is used. Together with the feature selection approach, where we can reduce the feature space from hundreds of thousands and millions of features to hundreds and even fewer features, our paper provides valuable solutions for those working with Big Data and high-dimensional datasets.

11.9 Bibliography

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Are your training datasets yet relevant? In *International Symposium on Engineering Secure Software and Systems*, pages 51–67. Springer, 2015.

[2] AVTEST. The independent IT-Security Institute. Malware. https://www.av-test.org/en/statistics/malware/, 2020.

[3] Sergii Banin. *Malware Analysis using Artificial Intelligence and Deep Learn-*

ing: Fast and straightforward feature selection method: A case of high dimensional low sample size dataset in malware analysis. Springer, 2020.

[4] Sergii Banin and Geir Olav Dyrkolbotn. Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107–S117, 2018.

[5] Sergii Banin and Geir Olav Dyrkolbotn. Correlating high-and low-level features. In *International Workshop on Security*, pages 149–167. Springer, 2019.

[6] Sergii Banin and Geir Olav Dyrkolbotn. Detection of running malware before it becomes malicious. In *International Workshop on Security*, pages 57–73. Springer, 2020.

[7] Sergii Banin, Andrii Shalaginov, and Katrin Franke. Memory access patterns for malware detection. *Norsk informasjonssikkerhetskonferanse (NISK)*, pages 96–107, 2016.

[8] Haipeng Cai. Assessing and improving malware detection sustainability through app evolution studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–28, 2020.

[9] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[11] IntelPin. A dynamic binary instrumentation tool, 2017.

[12] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[13] Malay Haldar. How much training data do you need? https://medium.com/@malay.haldar/how-much-training-data-do-you-need-da8ec091e956, 2015.

[14] Al Mead. Review of the development of multidimensional scaling methods. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 41(1):27–39, 1992.

[15] Bradley Austin Miller. *Scalable platform for malicious content detection integrating machine learning and manual review*. PhD thesis, UC Berkeley, 2015.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[17] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746, 2019.

[18] PortableApps.com. Portableapps.com. https://portableapps.com/apps, 2020.

[19] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79. IEEE, 2012.

[20] Saket Sathe and Charu C Aggarwal. Nearest neighbor classifiers versus random forests and support vector machines. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 1300–1305. IEEE, 2019.

[21] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. Machine learning aided static malware analysis: A survey and tutorial. In *Cyber Threat Intelligence*, pages 7–45. Springer, 2018.

[22] Ashu Sharma, Sanjay K Sahay, and Abhishek Kumar. Improving the detection accuracy of unknown malware by partitioning the executables in groups. In *Advanced computing and communication technologies*, pages 421–431. Springer, 2016.

[23] Sanjay Sharma, C Rama Krishna, and Sanjay K Sahay. Detection of advanced malware by machine learning techniques. In *Soft Computing: Theories and Applications*, pages 333–342. Springer, 2019.

[24] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[25] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

[26] Çağatay Yücel and Ahmet Koltuksuz. Imaging and evaluating the memory access for malware. *Forensic Science International: Digital Investigation*, 32:200903, 2020.

Appendix A Classification results achieved by RF

Table 6: RF accuracy

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.975	0.991	0.989	0.994	0.902	0.885	0.992	0.980	0.994	0.984	0.982	0.987
T0_1		1.000	1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_2			1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_5						1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_6							0.992	0.913	0.947	0.923	0.984	0.876
T0_7								0.930	0.953	0.949	0.987	0.878
T0_8									0.996	0.998	0.998	0.935
T0_9										0.998	0.997	0.945
T0_10											0.997	0.987
T0_11												0.976

Table 7: RF TPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.95	0.983	0.977	0.994	0.803	0.769	0.985	0.96	0.989	0.967	0.964	0.969
T0_1		1	1	1	1	1	1	1	1	1	1	1
T0_2			1	1	1	1	1	1	1	1	1	1
T0_5						1	1	1	1	1	1	1
T0_6							1	1	1	1	1	1
T0_7								1	1	1	1	1
T0_8									1	1	0.997	1
T0_9										1	0.993	1
T0_10											0.993	1
T0_11												1

Table 8: RF FPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0	0	0	0	0	0	0	0	0	0	0	0
T0_1		0	0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_2			0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_5						0	0.1	0.32	0.215	0.294	0.72	0.58
T0_6							0.015	0.173	0.106	0.154	0.033	0.211
T0_7								0.14	0.094	0.103	0.026	0.208
T0_8									0.008	0.005	0	0.11
T0_9										0.05	0	0.095
T0_10											0	0.022
T0_11												0.041

Appendix B Classification results achieved by J48

Table 9: J48 accuracy

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.975	0.989	0.989	0.994	0.902	0.885	0.992	0.980	0.994	0.984	0.982	0.987
T0_1		1.000	1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_2			1.000	1.000	1.000	1.000	0.950	0.840	0.892	0.853	0.640	0.660
T0_5						1.000	0.992	0.916	0.947	0.923	0.984	0.876
T0_6							0.950	0.840	0.892	0.853	0.640	0.660
T0_7								0.916	0.947	0.923	0.984	0.876
T0_8									0.996	1.000	0.997	0.998
T0_9										0.986	0.818	0.980
T0_10											0.995	0.995
T0_11												0.959

Table 10: J48 TPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.95	0.977	0.977	0.988	0.803	0.769	0.985	0.96	0.989	0.967	0.964	0.969
T0_1		1	1	1	1	1	1	1	1	1	1	1
T0_2			1	1	1	1	1	1	1	1	1	1
T0_5						1	1	1	1	1	1	1
T0_6							1	1	1	1	1	1
T0_7								1	1	1	1	1
T0_8									0.992	1	0.993	0.996
T0_9										1	0.993	0.996
T0_10											0.997	0.997
T0_11												0.931

Table 11: J48 FPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0	0	0	0	0	0	0	0	0	0	0	0
T0_1		0	0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_2			0	0	0	0	0.1	0.32	0.215	0.294	0.72	0.58
T0_5						0	0.015	0.167	0.106	0.154	0.033	0.211
T0_6							0.1	0.32	0.215	0.294	0.72	0.58
T0_7								0.167	0.106	0.154	0.033	0.211
T0_8									0	0	0	0
T0_9										0.028	0.358	0.032
T0_10											0.007	0.006
T0_11												0

Appendix C Classification results achieved by LWL

Table 12: LWL accuracy

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.996	0.991	1.000	0.997	0.977	0.923	0.992	0.916	0.947	0.916	0.980	0.871
T0_1		1.000	1.000	1.000	1.000	1.000	0.969	0.886	0.921	0.888	0.822	0.778
T0_2			1.000	1.000	1.000	1.000	0.973	0.903	0.928	0.911	0.912	0.815
T0_3				1.000	1.000	1.000	0.962	0.853	0.900	0.864	0.723	0.721
T0_4					1.000	1.000	0.969	0.853	0.921	0.876	0.775	0.765
T0_5						1.000	0.954	0.873	0.898	0.876	0.661	0.667
T0_6							0.992	0.903	0.947	0.916	0.933	0.861
T0_7								0.916	0.947	0.923	0.984	0.876
T0_8									0.992	0.984	0.819	0.946
T0_9										0.984	0.818	0.980
T0_10											0.989	0.989
T0_11												0.989

Table 13: LWL TPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0.992	0.98	1	0.994	0.96	0.85	1	0.99	1	0.986	0.987	0.991
T0_1		1	1	1	1	1	1	1	1	1	1	1
T0_2			1	1	1	1	1	1	1	1	1	1
T0_3				1	1	1	1	1	1	1	1	1
T0_4					1	1	1	1	1	1	1	1
T0_5						1	1	1	1	1	1	1
T0_6							1	1	1	1	1	1
T0_7								1	1	1	1	1
T0_8									1	1	0.997	1
T0_9										0.995	0.993	0.996
T0_10											0.98	0.982
T0_11												0.982

Table 14: LWL FPR

Train set	Test bin number											
	1	2	3	4	5	6	7	8	9	10	11	12
T0	0	0	0	0	0	0	0.015	0.16	0.106	0.154	0.026	0.215
T0_1		0	0	0	0	0	0.062	0.23	0.158	0.224	0.355	0.379
T0_2			0	0	0	0	0.054	0.19	0.143	0.178	0.176	0.315
T0_3				0	0	0	0.077	0.29	0.2	0.271	0.554	0.476
T0_4					0	0	0.062	0.29	0.158	0.248	0.45	0.401
T0_5						0	0.092	0.25	0.204	0.248	0.678	0.568
T0_6							0.015	0.19	0.106	0.168	0.134	0.237
T0_7								0.17	0.106	0.154	0.033	0.211
T0_8									0.015	0.033	0.358	0.091
T0_9										0.028	0.358	0.032
T0_10											0.003	0.006
T0_11												0.006

Chapter 12

S1: Machine Learning Aided Static Malware Analysis: A Survey and Tutorial

Andrii Shalaginov, **Sergii Banin**, Ali Dehghantanha, Katrin Franke

Abstract

Malware analysis and detection techniques have been evolving during the last decade as a reflection to development of different malware techniques to evade network-based and host-based security protections. The fast growth in variety and number of malware species made it very difficult for forensics investigators to provide an on time response. Therefore, Machine Learning (ML) aided malware analysis became a necessity to automate different aspects of static and dynamic malware investigation. We believe that machine learning aided static analysis can be used as a methodological approach in technical Cyber Threats Intelligence (CTI) rather than resource-consuming dynamic malware analysis that has been thoroughly studied before. In this paper, we address this research gap by conducting an in-depth survey of different machine learning methods for classification of static characteristics of 32-bit malicious Portable Executable (PE32) Windows files and develop taxonomy for better understanding of these techniques. Afterwards, we offer a tutorial on how different machine learning techniques can be utilized in extraction and analysis of a variety of static characteristic of PE binaries and evaluate accuracy and practical generalization of these techniques. Finally, the results of experimental study of all the method using common data was given to demonstrate the accuracy and complexity. This paper may serve as a stepping stone for future researchers in cross-disciplinary field of machine learning aided malware

forensics.

12.1 Introduction

Stealing users' personal and private information has been always among top interests of malicious programs [7]. Platforms which are widely used by normal users have always been best targets for malware developers [8].

Attackers have leveraged malware to target personal computers [21], mobile devices [61], cloud storage systems [12], Supervisory Control and Data Acquisition Systems (SCADA) [11], Internet of Things (IoT) network [81] and even big data platforms [67].

Forensics examiners and incident handlers on the other side have developed different techniques for detection of compromised systems, removal of detected malicious programs [22, 13], network traffic [63], and even log analysis [69]. Different models have been suggested for detection, correlation and analyses of cyber threats [19] (on a range of mobile devices [43] and mobile applications [45], cloud applications [10], cloud infrastructure [46] and Internet of Things networks [47]). Windows users are still comprising majority of Internet users hence, it is not surprising to see Windows as the most adopted PC Operating System (OS) on top of the list of malware targeted platforms [1]. In response, lots of efforts have been made to secure Windows platform such as educating users [54, 25], embedding an anti-virus software [40], deploying anti-malware and anti-exploitation tools [53, 52], and limiting users applications privilege [41].

In spite of all security enhancements, many malware are still successfully compromising Windows machines [36, 1] and malware is still ranked as an important threat to Windows platforms [33]. As result, many security professionals are still required to spend a lot of time on analyzing different malware species [9]. This is a logical step since malware analysis plays a crucial role in Cyber Threats Intelligence (CTI). There has been proposed a portal to facilitate CTI and malware analysis through interactive collaboration and information fusion [56].

There are two major approaches for malware analysis namely *static* (code) and *dynamic* (behavioral) malware analysis [15, 7]. In dynamic malware analysis, samples are executed and their run time behavior such as transmitted network traffic, the length of execution, changes that are made in the file system, etc. are used to understand the malware behavior and create indications of compromise for malware detection [15]. However, dynamic analysis techniques can be easily evaded by malware that are aware of execution conditions and computing environment [34]. Dynamic malware analysis techniques can only provide a snapshot view of the malware behavior and hence very limited in analysis of Polymorph or Metamorph species [44]. Moreover, dynamic malware analysis techniques are quite resource hungry which limits their enterprise deployment [37].

In static malware analysis, the analyst is reversing the malware code to achieve a deeper understanding of the malware possible activities. [28]. Static analysis relies on extraction of a variety of characteristics from the binary file such that function calls, header sections, etc. [83]. Such characteristics may reveal indicators of malicious activity that are going to be used in CTI [57]. However, static analysis is quite a slow process and requires a lot of human interpretation and hence [7].

Static analysis of PE32 is a many-sided challenge that was studied by different authors. Static malware analysis also was used before for discovering interconnections in malware species for improved Cyber Threat Intellifence [42, 66]. As 32-bit malware are still capable of infecting 64-bit platforms and considering there are still many 32-bit Windows OS it is not surprising that still majority of Windows malware are 32-bit Portable Executable files [7]. To authors knowledge there has not been a comparative study of ML-based static malware using a single dataset which produces comparable results. We believe that utilization of ML-aided automated analysis can speed up intelligent malware analysis process and reduce human interaction required for binaries processing. Therefore, there is a need for thorough review of the relevant scientific contributions and offer a taxonomy for automated static malware analysis.

The remainder of this paper is organized as follows. We first offer a comprehensive review of existing literature in machine learning aided static malware analysis. We believe this survey paves the way for further research in application of machine learning in static malware analysis and calls for further development in this field. Then, taxonomy of feature construction methods for variety of static characteristics and corresponding ML classification methods is offered. Afterwards, we offer a tutorial that applies variety of set of machine learning techniques and compares their performance. The tutorial findings provide a clear picture of pros and cons of ML-aided static malware analysis techniques. To equally compare all the methods we used one benign and two malware datasets to evaluate all of the studied methods. This important part complements the paper due to the fact that most of the surveyed works used own collections, sometimes not available for public access or not published at all. Therefore, experimental study showed performance comparison and other practical aspects of ML-aided malware analysis. Section 12.4 gives an insight into a practical routine that we used to establish our experimental setup. Analysis of results and findings are given in the Section 12.5. Finally, the paper is concluded and several future works are suggested in the Section 12.6.

12.2 An overview of Machine Learning-aided static malware detection

This section provides an analysis of detectable static properties of 32 bit PE malware followed by detailed description of different machine learning techniques to develop a taxonomy of machine learning techniques for static malware analysis.

12.2.1 Static characteristics of PE files

PE file format was introduced in Windows 3.1 as PE32 and further developed as PE32+ format for 64 bit Windows Operating Systems. PE files contain a Common Object File Format (COFF) header, standard COFF fields such as header, section table, data directories and Import Address Table (IAT). Beside the PE header fields a number of other static features can be extracted from a binary executable such as strings, entropy and size of various sections.

To be able to apply Machine Learning PE32 files static characteristics should be converted into a machine-understandable features. There exist different types of features depending on the nature of their values such that *numerical* that describes a quantitative measure (can be integer, real or binary value) or *nominal* that describes finite set of categories or labels. An example of the *numerical* feature is CPU (in $\%$) or RAM (in Megabytes) usage, while *nominal* can be a file type (like $*.dll$ or $*.exe$) or Application Program Interface (API) function call (like $write()$ or $read()$).

1. *n-grams of byte sequences* is a well-known method of feature construction utilizing sequences of bytes from binary files to create features. Many tools have been developed for this purpose such as *hexdump* [39] created 4-grams from byte sequences of PE32 files. The features are collected by sliding window of *n* bytes. This resulted in 200 millions of features using 10-grams for about two thousands files in overall. Moreover, feature selection (FS) was applied to select 500 most valuable features based on Information Gain metric. Achieved accuracy on malware detection was up to 97% using such features. Another work on byte n-grams [51] described usage of 100-500 selected n-grams yet on a set of 250 malicious and 250 benign samples. Similar approach [31] was used with $10, \ldots, 10,000$ best n-grams for $n = 1, \ldots, 10$. Additionally, ML methods such that Naive Bayes, C4.5, k-NN and others were investigated to evaluate their applicability and accuracy. Finally, a range of 1-8 n-grams [27] can result in 500 best selected n-grams that are used later to train AdaBoost and Random Forests in addition to previously mentioned works.

2. *Opcode sequences* or operation codes are set of consecutive low level machine abstractions used to perform various CPU operations. As it was shown [62]

such features can be used to train Machine Learning methods for successful classification of the malware samples. However, there should be a balance between the size of the feature set and the length of n-gram opcode sequence. N-grams with the size of 4 and 5 result in highest classification accuracy as unknown malware samples could be unveiled on a collection of 17,000 malware and 1,000 benign files with a classification accuracy up to 94% [58]. Bragen [5] explored reliability of malware analysis using sequences of opcodes based on the 992 PE-files malware and benign samples. During the experiments, about 50 millions of opcodes were extracted. 1-gram- and 2-gram-based features showed good computational results and accuracy. Wang et al. [79] presented how the 2-tuple opcode sequences can be used in combination with density clustering to detect malicious or benign files.

3. *API calls* are the function calls used by a program to execute specific functionality. We have to distinguish between System API calls that are available through standard system DLLs and User API calls provided by user installed software. These are designed to perform a pre-defined task during invocation. Suspicious API calls, anti-VM and anti-debugger hooks and calls can be extracted by PE analysers such as PEframe [3]. [83] studied 23 malware samples and found that some of the API calls are present only in malwares rather than benign software. Function calls may compose in graphs to represent PE32 header features as nodes, edges and subgraphs [84]. This work shows that ML methods achieve accuracy of 96% on 24 features extracted after analysis of 1,037 malware and 2,072 benign executables. Further, in [71] 20,682 API calls were extracted using PE parser for 1,593 malicious and benign samples. Such large number of extracted features can help to create linearly separable model that is crucial for many ML methods as Support Vector Machines (SVM) or single-layer Neural Networks. Another work by [55] described how API sequences can be analysed in analogy with byte n-grams and opcode n-grams to extract corresponding features to classify malware and benign files. Also in this work, an array of API calls from IAT (PE32 header filed) was processed by Fisher score to select relevant features after analysis of more than 34k samples.

4. *PE header* represents a collection of meta data related to a Portable Executable file. Basic features that can be extracted from PE32 header are *Size of Header*, *Size of Uninitialized Data*, *Size of Stack Reserve*, which may indicate if a binary file is malicious or benign [14]. The work [75] utilized Decision Trees to analyse PE header structural information for describing malicious and benign files. [76] used 125 raw header characteristics, 31 sec-

tion characteristics, 29 section characteristics to detect unknown malware in a semi-supervised approach. Another work [80] used a dataset containing 7,863 malware samples from Vx Heaven web site in addition to 1,908 benign files to develop a SVM based malware detection model with accuracy of 98%. [38] used F-score as a performance metric to analyse PE32 header features of 164,802 malicious and benign samples. Also [29] presented research of two novel methods related to PE32 string-based classier that do not require additional extraction of structural or meta-data information from the binary files. Moreover, [84] described application of 24 features along with API calls for classification of malware and benign samples from VxHeaven and Windows XP SP3 respectively. Further, ensemble of features was explored [59], where authors used in total 209 features including structural and raw data from PE32 file header. Further, Le-Khac et al. [35] focused on Control Flow Change over first 256 addresses to construct n-gram features.

In addition to study of specific features used for malware detection, we analyzed articles devoted to application of ML for static malware analysis published between 2000 and 2016, which covers the timeline of Windows NT family that are still in use as depicted in the Figure 12.1. We can see that the number of papers that are relevant to our study is growing from 2009 and later, which can be justified on the basis of increase in the number of Windows users (potential targets) and corresponding malware families.

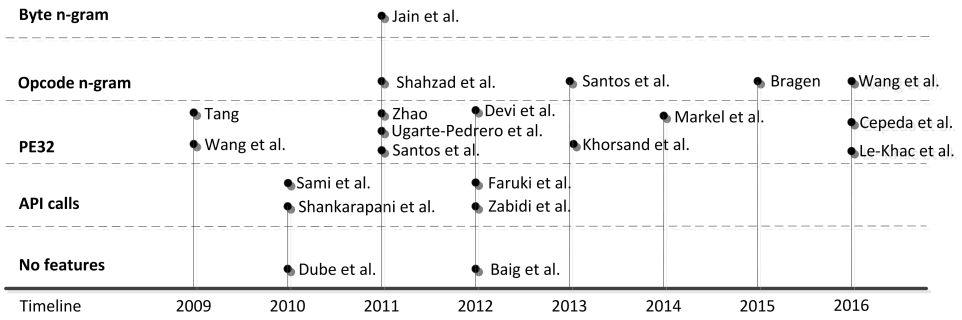


Figure 12.1: Timeline of works since 2009 that involved static analysis of Portable Executable files using method characteristics using also ML method for binary malware classification

Challenges. Despite the fact that some of the feature construction techniques reflected promising precision of 90+ % in differentiation between malicious and benign executables, there are still no best static characteristic that guarantee 100% accuracy of malware detection. This can be explained by the fact that malware are using obfuscation and encryption techniques to subvert detection mechanisms.

In addition, more accurate approaches such as bytes N-GRAMS are quite resource intensive and hardly practical in the real world.

12.2.2 Machine Learning methods used for static-based malware detection

Statistical methods

Exploring large amounts of binary files consists of statistical features may be simplified using so called frequencies or likelihood of features values. These methods are made to provide prediction about the binary executable class based on statistics of different static characteristics (either automatically or manually collected) which are applicable to malware analysis too as describe by Shabtai et al. [60]. To process such data, extract new or make predictions the following set of statistical methods can be used:

Naive Bayes is a simple probabilistic classifier, which is based on Bayesian theorem with naive assumptions about independence in correlation of different features. The Bayes Rule can be explained as a following conditional independences of features values with respect to a class:

$$P(C_k|V) = P(C_k)\frac{P(V|C_k)}{P(V)} \tag{12.1}$$

where $P(C_k)$ is a prior probability of class $C_k, k = 1, ..., m_0$ which is calculated from collected statistics according to description of variables provided by Kononenko et al. [32]. This method is considered to tackle just binary classification problem (benign against malicious) since it was originally designed as multinomial classifier. $V = \langle v_1, ..., v_a \rangle$ is the vector of attributes values that belongs to a sample. In case of Naive Bayes *input* values should be symbolical, for example strings, opcodes, instruction n-grams etc. $P(V)$ is the prior probability of a sample described with vector V. Having training data set and given vector V we count how many samples contain equal values of attributes (e.g. based on the number of sections or given opcode sequence). It is important to mention that V have not to be of length of full attribute vector and can contain only one attribute value. $P(V|C_k)$ is the conditional probability of a sample described with V given the class C_k. And $P(C_k|V)$ conditional probability of class C_k with V. Based on simple probability theory we can describe conditional independence of attribute values v_i given the class C_k:

$$P(V|C_k) = P(v_1 \wedge ... \wedge v_a|C_k) = \prod_{i=1}^{a} P(v_i|C_k) \tag{12.2}$$

Dropping the mathematical operations we get final version of Equation 12.1:

$$P(C_k|V) = P(C_k)\prod_{i=1}^{a}\frac{P(C_k|v_i)}{P(C_k)} \tag{12.3}$$

So, the task of this machine learning algorithm is to calculate conditional and unconditional probabilities as described in the Equation 12.3 using a training dataset. To be more specific, the Algorithm 1 pseudo code shows the calculation of the conditional probability.

Algorithm 1 Calculating $P(C_k|V)$ - conditional probability of class C_k with V

$Sample$ structure with class label and attribute values
$S \leftarrow$ array of training Samples
$V \leftarrow$ array of attribute values
$C_k \leftarrow$ class number
$P\_C_k = 0$
function $get\_P\_C_k(ClassNumber, Samples)$
    $output = 0$
    for all sample from Samples **do**
        if $sample.getClass() == ClassNumber$ **then**
            $output+ = \frac{1}{size[Samples]}$
        end if
    end for
    return $output$
end function
function $get\_P\_C_k\_v_i(ClassNumber, v, i, Samples)$
    $output = 0$
    for all sample from Samples **do**
        if $sample.getClass() == ClassNumber$ AND $sample.getAttribute(i) == v$ **then**
            $output+ = \frac{1}{size[Samples]}$
        end if
    end for
    return $output$
end function
$P(C_k|V = 0)$
$prod = 1$
i=0
for all $vfromV$ **do**
    $prod* = \frac{get\_P\_C_k\_v_i(ClassNumber,v,i,Samples)}{get\_P\_C_k(ClassNumber,Samples)}$
    $i+ = 1$
end for
$P(C_k|V) = prod * get\_P\_C_k(ClassNumber, Samples))$

So, we can see from the Equation 12.1 that given *output* is as a probability

that a questioned software sample belongs to one or another class. Therefore, the classification decision will be made by finding a maximal value from set of corresponding class likelihoods. Equation 12.4 provides formula that assigns class label to the *output*:

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} P(C_k)P(V|C_k). \tag{12.4}$$

Bayesian Networks is a probabilistic directed acyclic graphical model (sometimes also named as Bayesian Belief Networks), which shows conditional dependencies using directed acyclic graph. Network can be used to detect *"update knowledge of the state of a subset of variables when other variables (the evidence variables) are observed"* [32]. Bayesian Networks are used in many cases of classification and information retrieval (such as semantic search). The method's routine can be described as following. If edge goes from vertex A to vertex B, then A is a parent of B, and B is an ancestor of A. If from A there is oriented path to another vertex B exists then B is ancestor of A, and A is a predecessor of B. Let's designate set of parent vertexes of vertex V_i as $parents(V_i) = PA_i$. Direct acyclic graph is called Bayesian Network for probability distribution $P(v)$ given for set of random variables V if each vertex of graph has matched with random variable from V. And edge of a graph fits next condition: every variable v_i from V must be conditionally independent from all vertexes that are not its ancestors if all its direct parents PA_i are initialized in graph G:

$$\forall V_i \in V \Rightarrow P(v_i|pa_i, s) = P(v_i|pa_i) \tag{12.5}$$

where v_i is a value of V_i, S - set of all vertexes that are not ancestors of V_i, s - configuration of S, pa_i - configuration of PA_i. Then full general distribution of the values in vertexes could be written as product of local distributions, similarly to Naive Bayes rules:

$$\mathrm{P}(V_1, \ldots, V_n) = \prod_{i=1}^{n} \mathrm{P}(V_i \mid \mathrm{parents}(V_i)) \tag{12.6}$$

Bayesian Belief Networks can be used for classification [32], thus can be applied for malware detection and classification as well [58]. To make Bayesian Network capable of classification it should contain classes as parent nodes which don't have parents themselves. Figure 12.2 shows an example of such Bayesian network.

Rule based

Rule based algorithms are used for generating crisp or inexact rules in different Machine Learning approaches [32]. The main advantage of having logic rules

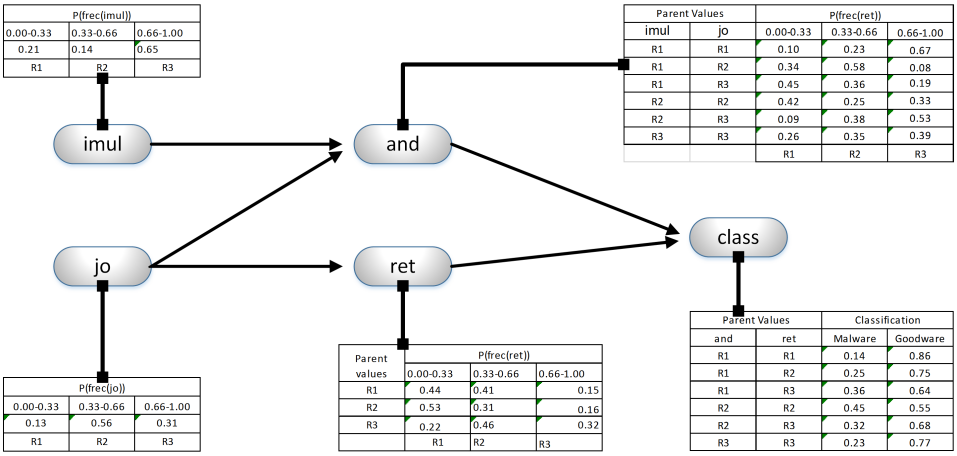


Figure 12.2: Bayesian network suitable for malware classification [58]

involved in malware classification is that logical rules that operate with statements like *equal*, *grater then*, *less or equal to* can be executed on the hardware level which significantly increases the speed of decision making.

C4.5 is specially proposed by Quinlan [50] to construct decision trees. These trees can be used for classification and especially for malware detection [74]. The process of trees training includes processing of previously classified dataset and on each step looks for an attribute that divides set of samples into subsets with the best information gain. C4.5 has several benefits in compare with other decision tree building algorithms:

- Works not only with discrete but with continuous attributes as well. For continuous attributes it creates threshold tp compares values against [49].
- Take into account missing attributes values.
- Works with attributes with different costs.
- Perform automate tree pruning by going backward through the tree and removing useless branches with leaf nodes.

Algorithm 2 shows a simplified version of decision tree building algorithm.

Algorithm 2 Decision tree making algorithm

1: $S = s_1, s_2, ...$ labelled training dataset of classified data
2: $x_{1i}, x_{2i}, ..., x_{pi}$ - p-dimensional vector of attributes of each sample s_i form S
3: Check for base cases
4: **for all** attributes x **do**
5: Find the normalized gain ratio from splitting set of sample on x
6: **end for**
7: Let x_{best} be the attribute with the highest normalized information gain.
8: Create decision $node$ that splits on x_{best}
9: Repeat on the subsets created by splitting with x_{best}. Newly gained nodes add as $children$ of current $node$.

Neuro-Fuzzy is a hybrid models that ensembles neural networks and fuzzy logic to create human-like linguistic rules using the power of neural networks. Neural network also known as artificial neural network is a network of simple elements which are based on the model of perceptron [65]. Perceptron implements previously chosen activation functions which take input signals and their weights and produces an output, usually in the range of $[0, 1]$ or $[-1, 1]$. The network can be trained to perform classification of complex and high-dimensional data. Neural Networks are widely used for classification and pattern recognition tasks, thus for malware analysis [72]. The problem is that solutions gained by Neural Networks are usually impossible to interpret because of complexity of internal structure and increased weights on the edges. This stimulates usage of Fuzzy Logic techniques, where generated rules are made in human-like easy-interpretable format: $IF\ \ X > 3\ \ AND\ \ X < 5\ \ THEN\ \ Y = 7$.

Basic idea of Neuro-Fuzzy (NF) model is a fuzzy system that is trained with a learning algorithm similar to one from neural networks theory. NF system can be represented as a neural network which takes input variables and produces output variables while connection weights are represented as encoded fuzzy sets. Thus at any stage (like prior to, in process of and after training) NF can be represented as a set of fuzzy rules. Self-Organising (Kohonen) maps [30] is the most common techniques of combining Neuro and Fuzzy approaches. Shalaginov et al. [64] showed the possibility of malware detection using specially-tuned Neuro-Fuzzy technique on a small dataset. Further, NF showed good performance on large-scale binary problem of network traffic analysis [63]. This method has also proven its efficiency on a set of multinomial classification problems. In particular, it is useful when we are talking about distinguishing not only "malware" or "goodware" but also detecting specific type of "malware" [68]. Therefore, it has been improved for the multinational classification of malware types and families by Shalaginov et al. [70].

Distance based

This set of methods is used for classification based on predefined distance measure. Data for distance-based methods should be carefully prepared, because computational complexity grows significantly with space dimensionality (number of features) and number of training samples. Thus there is a need for proper feature selection as well as sometimes for data normalization.

k-Nearest Neighbours or k-NN is classification and regression method. k-NN does not need special preparation of the dataset or actual "training" as the algorithm is ready for classification right after labelling the dataset. The algorithm takes a sample that is need to be classified and calculates distances to samples from training dataset, then it selects k nearest neighbours (with shortest distances) and makes decision based on class of this k nearest neighbours. Sometimes it makes decision just on the majority of classes in this k neighbours selection, while in other cases there is weights involved in process of making decision. When k-NN is used for malware classification and detection there is a need for careful feature selection as well as a methodology for dealing with outliers and highly mixed data, when training samples cannot create distinguishable clusters [58].

Support Vector Machine or SVM is a supervised learning method. It constructs one or several hyperplanes to divide dataset for classification. Hyperplane is constructed to maximize distance from it to the nearest data points. Sometimes kernel transformation is used to simplify hyperplanes. Building a hyperplane is usually turned into two-class problem (one vs one, one vs many) and involves quadratic programming. Let's have linearly separable data (as shown in Figure 12.3) which can be represented as $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^p,\ y_i \in \{-1, 1\}\}_{i=1}^{n}$. Where y_i is 1 or -1 depending on class of point x_i. Each x_i is p-dimensional vector (not always normalised). The task is to find hyperplane with maximum margin that divides dataset on points with $y_i = 1$ and $y_i = -1$: $w \cdot x - b = 0$. Where w is a normal vector to a hyperplane [20]. If dataset is linearly separable we can build two hyperplanes $w \cdot x - b = 1$ and $w \cdot x - b = -1$ between which there will be no (or in case of soft margin maximal allowed number) points. Distance between them (margin) is $\frac{2}{||w||}$, so to maximize margin we need to minimize $||w||$ and to find parameters of hyperplane we need to introduce Lagrangian multipliers α and solve Equation 12.7 with quadratic programming techniques.

$$\arg\min_{\mathbf{w},b}\max_{\boldsymbol{\alpha}\geq 0}\left\{\frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n}\alpha_i[y_i(\mathbf{w}\cdot\mathbf{x_i} - b) - 1]\right\} \tag{12.7}$$

Figure 12.3: Maximum margin hyperplane for two class problem [32]

Sometimes there is a need to allow an algorithm to work with misclassified data hence leaving some points inside the margin based on the degree of misclassification ξ. So the Equation 12.3 turns into Equation 12.8.

$$\arg\min_{\mathbf{w},\xi,b}\max_{\boldsymbol{\alpha},\boldsymbol{\beta}}\left\{\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i[y_i(\mathbf{w}\cdot\mathbf{x_i} - b) - 1 + \xi_i] - \sum_{i=1}^{n}\beta_i\xi_i\right\} \quad with\ \ \alpha_i, \beta_i \geq 0 \tag{12.8}$$

Also the data might be linearly separated, so there is a need for kernel trick. The basic idea is to substitute every *dot product* with non-linear kernel function. Kernel function can be chosen depending on situation and can be polynomial, Gaussian, hyperbolic etc. SVM is a very powerful technique which can give good accuracy if properly used, so it often used in malware detection studies as shown by Ye et al. [82].

Neural networks

Neural Network is based on the model of perceptron which has predefined activation function. In the process of training weights of the links between neurons are trained to fit train data set with minimum error with use of back propagation. Artificial Neural network (ANN) consists of input layer, hidden layer (layers) and output layer as it is shown on Figure 12.4.

Figure 12.4: Artificial neural network [32]

The input layer takes normalized data, while hidden output layer produces activation output using neuron's weighted input and activation function. Activation function is a basic property of neuron that takes input values given on the input edges, multiply them by weights of these edges and produces output usually in a range of [0,1] or [-1,1]. Output layer is needed to present results and then interpret them. Training of ANN starts with random initialization of weights for all edges. Then feature vector of each sample is used as an input. Afterwards, result gained on the output layer is compared to the real answers. Any errors are calculated and using back-propagation all weights are tuned. Training can continue until reaching desired number of training cycles or accuracy. Learning process of ANN can be presented as shown in the Algorithm 3. Artificial Neural networks can be applied for complex models in high-dimensional spaces. This is why it often used for malware research [72].

Algorithm 3 ANN training

$S = s_1, s_2, ...$ labeled training dataset of classified data
$x_{1i}, x_{2i}, ..., x_{pi}$ - p-dimensional vector of attributes of each sample s_i form S
N number of training cycles
L_{rate} learning rate
Random weight initialization
for all training cycles N **do**
    for all samples S **do**
        give features x_i as input to the ANN
        compare class of s_i with gained output of ANN
        calculate error
        using back-propagation tune weights inside the ANN with L_rate
    end for
    reduce L_{rate}
end for

Open Source and Freely available ML Tools

Today machine learning is widely used in many areas of research with many publicly available tools (Software products, libraries etc.).

Weka or Waikato Environment for Knowledge Analysis is a popular, free, cross platform and open source tool for machine learning. It supports many of popular ML methods with possibility of fine tuning of the parameters and final results analysis. It provides many features such as splitting dataset and graphical representation of the results. Weka results are saved in .arff file which is specially prepared CSV file with header. It suffers from couple of issues including no support for multi-thread computations and poor memory utilization especially with big datasets.

Python weka wrapper is the package which allows using power of Weka through Python programs. It uses javabridge to link Java-based Weka libraries to python. It provides the same functionality as Weka, but provides more automation capacities.

LIBSVM and LIBLINEAR are open source ML libraries written in C++ supporting kernelized SVMs for linear, classification and regression analysis. Bindings for Java, Mathlab and R are also present. It uses space-separated files as input, where zero values need not to be mentioned.

RapidMiner is machine learning and data mining tool with a user friendly GUI and support for a lot of ML and data mining algorithms.

Dlib is a free and cross-platform C++ toolkit which supports different -machine learning algorithms and allows multi-threading and utilization of Python APIs.

Feature Selection & Construction process

Next important step after the characteristics extraction is so-called Feature Selection process [32]. Feature Selection is a set of methods that focus on elimination of irrelevant or redundant features that are not influential for malware classification. This is important since the number of characteristics can be extremely large, while only a few can actually be used to differentiate malware and benign applications with a high degree of confidence. The most common feature selection methods are *Information Gain*, *a*nd *Correlation-based Feature Subset Selection (CFS)* [24]. The final goal of Feature Selection is to simplify the process of knowledge transfer from data to a reusable classification model.

Taxonomy of malware static analysis using Machine Learning

Our extensive literature study as reflected in Table 12.1 resulted to proposing a taxonomy for malware static analysis using machine learning as shown in Figure 12.5. Our taxonomy depicts the most common methods for analysis of static characteristics, extracting and selecting features and utilizing machine learning classification techniques. Statistical Pattern Recognition process [26] was used as the basis for our taxonomy modelling.

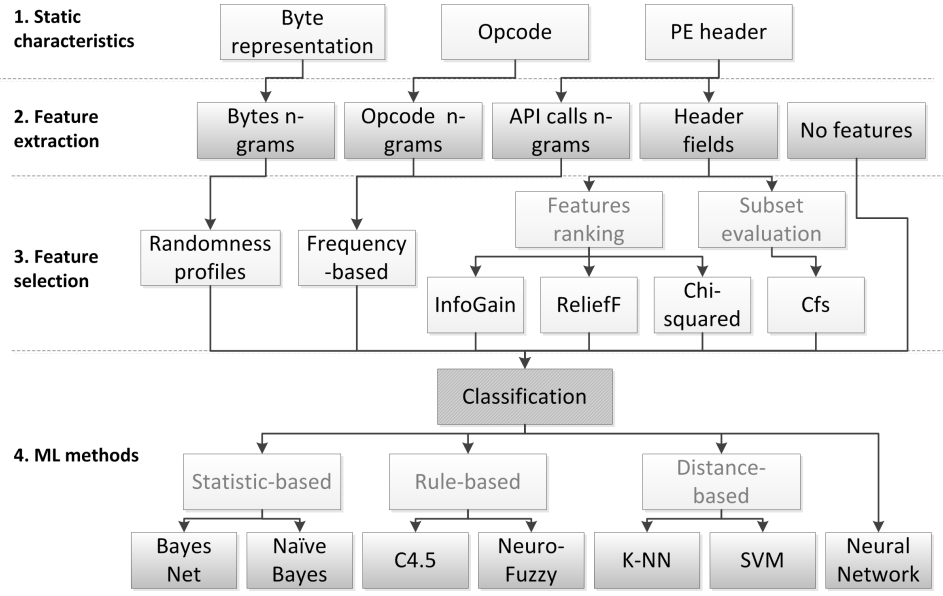


Figure 12.5: Taxonomy of common malware detection process based on static characteristics and Machine Learning

Year	Authors	Dataset	Features	FS	ML
			PE32 header		

2016	Cepeda et al. [6]	7,630 malware and 1.818 goodware	57 features from VirisTotal	ChiSqSelector with 9 features finally	SVM, RF, NN
2016	Le-Khac et al. [35]	Malicious: 94 ;Benign: 620	Control Flow Change and 2-6 n-grams	-	Naive Bayes
2014	Markel et al. [38]	Malicious: 122,799, Benign: 42,003	46 features use python 'pefile'	-	Naive Bayes, Logistic Regression, Classification and Regression Tree (CART)
2013	Khorsand et al. [29]	Benign: 850 "EXE" and 750 "DLL"; Malware: 1600 from VX heavens	eliminated	-	Prediction by partial matching
2012	Devi et al. [14]	4,075 PE files: 2954 malicious and 1121 Windows XP SP2 benign	2 + 5 features	-	BayesNet, k-NN, SVM, AdaBoostM1, Decision table, C4.5, Random Forest, Random Tree
2011	Zhao [84]	3109 PE: 1037 viruses from Vx Heavens and 2072 benign executable on Win XP Sp3	24 features from PE files using Control Flow Graph-based on nodes		Random Forest, Decision Tree, Bagging, C4.5
2011	Ugarte-Pedrero et al. [76]	500 benign from WinXP and 500 non-packed from Vxheaven; 500 packed + 500 Zeus	166 structure features of PE file	InfoGain	Learning with Local and Global Consistency, Random Forest
2011	Santos et al. [59]	500 benign and 500 malicious from VxHeavens, also packed and not packed	209 structural features	InfoGain	Collective Forest
2009	Tang [75]	361 executables and 449 normal trojan files	PE header structural features	-	Decision Tree
2009	Wang et al. [80]	Benign: 1,908, Malicious: 7,863	143 PE header entries	InfoGain, Gain raio	SVM
			bytes n-gram sequences		
2011	Jain et al. [27]	1,018 malware and 1,120 benign samples	1-8 byte, n-gram, best n-gram by documentwise frequency	-	NB, iBK, J48, AdaBoost1, RandomForest
2007	Masud et al. [39]	1st set - 1,435 executables: 597 of which are benign and 838 are malicious. 2nd set - 2,452 executables: 1,370 benign and 1,082 malicious	500 best n-grams	InfoGain	SVM

2006	Reddy et al. [51]	250 malware vs 250 benign	100-500 best n-gram	Document Frequency, InfoGain	NB, iBK, Decision Tree
2004	Kolter et al. [31]	1971 benign, 1651 malicious from Vx Heaven	500 best n-grams	InfoGain	Naive Bayes, SVM, C4.5
			opcode n-gram sequences		
2016	Wang et al. [79]	11,665 malware and 1,000 benign samples	2-tuple opcode sequences	information entropy	density clustering
2015	Bragen [5]	992 malwares, 771 benign from Windows Vista	1-4 n-gram opcode with vocabulary 530-714,390	Cfs, Chi-sqaured, InfoGain, ReliefF, SymUncert.	Random Forest, C4.5, Naive Bayes, bayes Net, Baggin, ANN, SOM, k-nn
2013	Santos et al. [58]	13,189 malware vs 13,000 benign	top 1,000 features	InfoGain	Random Forest, J48, k-Nearest Neighbours, Bayesian networks, SVM
2011	Shahzad et al. [62]	Benign: 300, Malicious: 300 on Windows XP	coabulary of 1,413 with n-gram=4	tf-idf	ZeroR, Ripper, C4.5, SVM, Naive Bayes, k-nn
			API calls		
2012	Zabidi et al. [83]	23 malware and 1 benign	API calls, debugger features, VM features	-	-
2012	Faruki et al. [18]	3234 benign, 3256 malware	1-4 API call-gram	-	Random Forest, SVM, ANN, C4.5, Naive bayes
2010	Shankarapani et al. [71]	1593 PE files:875 benign and 715 malicious	API calls sequence	-	SVM
2010	Sami et al. [55]	34,820 PE: 31,869 malicious and 2951 benign from Windows	API calls	Fisher Score	Random Forest, C4.5, Naive Bayes
			no features / not described		
2012	Baig et al. [4]	200 packed PE and 200 unpacked from Windows 7, Windows 2003 Server	file entropy	-	-
2010	Dube et al. [16]	40,498 samples: 25,974 malware, 14,524 benign	from 32 bit files	-	Decision Tree

Table 12.1: Analysis of ML methods applicability for different types of static characteristics

To get a clear picture on application domain of each machine learning and fea-

ture selection method we analysed reported performance as shown in Figure 12.6. Majority of researchers were using byte n-gram, opcode n-gram and PE32 header fields for static analysis while C4.5, SVM or k-NN methods were mainly used for malware detection. Information Gain is the prevalent method to define malware attributes. Also we can see that n-gram-based method tend to use corresponding set of feature selection like tf-idf and Symmetric Uncertainty that are more relevant for large number of similar sequences. On the other hand, PE32 header-based features tend to provide higher entropies for classification and therefore Control-Flow graph-based and Gain Ratio are more suitable for this task.

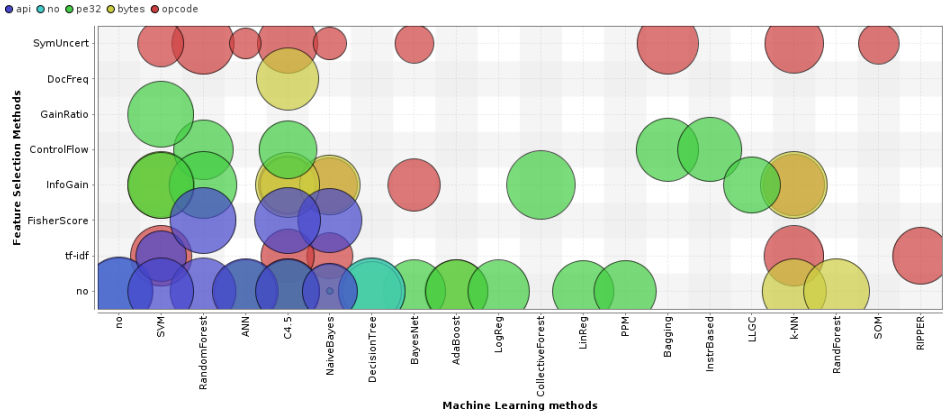


Figure 12.6: Comparison of accuracy of various static characteristics with respect to feature selection and machine learning methods. Colour of the bubbles shows used characteristics for detection, while size of the bubble denotes achieved accuracy

To conclude, one can say that majority of authors either extract features that offers good classification accuracy, or use conventional methods like Information Gain. However, n-gram based characteristics need other FS approaches to eliminate irrelevant features. Rule-based ML is the most commonly used classification method along with SVM. Forest-based method tends to be more applicable for PE32 header-based features. Also ANN is not commonly-used technique. While most of the works achieved accuracies of 80-100%, some Bayes-based methods offered much lower accuracy even down to 50% only.

12.3 Approaches for Malware Feature Construction

Similar the previous works, following four sets of static properties are suggested for feature classification in this paper:

PE32 header features characterize the PE32 header information using the *PEframe* tools [76]. Following numerical features will be used in our experi-

ments:

- *ShortInfo_Directories* describes 16 possible data directories available in PE file. The most commonly used are "Import", "Export", "Resource", "Debug", "Relocation".
- *ShortInfo_Xor* indicates detected XOR obfuscation.
- *ShortInfo_DLL* is a binary flag of whether a file is executable or dynamically-linked library.
- *ShortInfo_FileSize* measures size of a binary file in bytes.
- *ShortInfo_Detected* shows present techniques used to evade the detection by anti-viruses like hooks to disable execution in virtualized environment or suspicious API calls.
- *ShortInfo_Sections* is a number of subsections available in the header.
- *DigitalSignature* contains information about the digital signatture that can be present in a file
- *Packer* describes used packer detected by *PEframe*
- *AntiDebug* gives insight into the techniques used to prevent debugging process.
- *AntiVM* is included to prevent the execution in a virtualize environment.
- *SuspiciousAPI* indicates functions calls that are labelled by *PEframe* as suspicious.
- *SuspiciousSections* contains information about suspicious sections like ".rsrc \u0000 \u0000 \u0000"
- *Url* is a number of different url addresses found in the binary file.

Byte n-gram. N-gram is a sequence of some items (with minimum length of 1) that are predefined as minimal parts of the object expressed in Bytes. By having the file represented as a sequence of bytes we can construct 1-gram, 2-gram, 3-gram etc. N-grams of bytes, or byte n-grams are widely used as features for machine learning and static malware analysis [27, 51].

Reddy et al. [51] used n-grams of size 2,3 and 4 with combination of SVM, Instance-based learner and Decision Tree algorithms to distinguish between malicious and benign executables. After extracting n-grams they used class-wise

document frequency as a feature selection measure and showed that class-wise document frequency is performing better than Information Gain as a feature selection measure. Jain et al. [27] used n-grams in range of 1 to 8 as features and Naive Bayes, Instance Based Learner and AdaBoost1 [23] as machine learning algorithms for malware classification and reported byte 3-grams as the best technique.

Opcode n-gram represent a set of instructions that will be performed on the CPU when binary is executed. These instructions are called operational codes or **opcodes**. To extract opcodes from executable we need to perform disassembly procedure. After this opcodes will be represented as short instructions names such as *POP, PUSH, MOV, ADD, SUB* etc. Santos et al. [58] described a method to distinguish between malicious and benign executables or detecting different malware families using opcode sequences of length 1 to 4 using Random Forest, J48, k-Nearest Neighbours, Bayesian Networks and Support Vector Machine algorithms [23].

API calls is a set of tools and routines that help to develop a program using existent functionality of an operating system. Since most of the malware samples are platform dependent it is very much likely that their developers have use APIs as well. Therefore, analysing API calls usage among benign and malicious software can help to find malware-specific API calls and therefore are suitable to be used as a feature for machine learning algorithms. For example, [71] successfully used Support Vector Machines with frequency of API calls for malware classification. [77] provided a methodology for classification of malicious and benign executables using API calls and n-grams with n from 1 to 4 and achieved accuracy of 97.23% for 1-gram features. [18] used so-called API call-gram model with sequence length ranging from 1-4 and reached accuracy of 97.7% was achieved by training with 3-grams. In our experiments we are going to use 1 and 2 n-grams as features generated from API calls.

12.4 Experimental Design

All experiments were conducted on a dedicated Virtual machine (VM) on Ubuntu 14.04 server running on Xen 4.4. The server had an Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz with 4 cores (8 threads), out of which 2 cores (4 threads) were provided to the VM. Disk space is allocated on the SSD RAID storage based on Samsung 845DC. Installed server memory was Kingston PC-1600 RAM, out of which 8GB was available for the VM. Operating system was an Ubuntu 14.04 64 bit running on ae dedicated VM together with all default tools and utilities available in the OS's repository. Files pre-processing were performed using *bash* scripts due to native support in Linux OS. To store extracted features we utilised MySQL 5.5 database engine together with Python v 2.7.6 and PHP v 5.5.9 connectors.

For the experiments we used a set of benign and malicious samples. To authors knowledge there have not been published any large BENIGN SOFTWARE REFERENCE DATASETS, so we have to create our own set of benign files. Since the focus of the paper is mainly on PE32 Windows executables, we decided to extract corresponding known-to-be-good files from different versions of MS Windows, including different software and multimedia programs installations that are available. The OSes that we processed were 32 bit versions of Windows XP, Windows 7, Windows 8.1 and Windows 10. Following two Windows malware datasets were used in our research:

1. VX HEAVEN [2] dedicated to distribute information about the computer viruses and contains 271,092 sorted samples dating back from 1999.

2. VIRUS SHARE [78] represent sharing resource that offers 29,119,178 malware samples and accessible through VirusShare tracker as of 12th of July, 2017. We utilized following two archives: *VirusShare_00000.zip* created on 2012-06-15 00:39:38 with a size of 13.56 GB and *VirusShare_00207.zip* created on 2015-12-16 22:56:17 with a size of 13.91 GB, all together contained 131,072 unique, uncategorised and unsorted malware samples. They will be referred further as *malware_000* and *malware_207*.

To be able to perform experiments on the dataset, we have to filter out irrelevant samples (not specific PE32 and not executables), which are out of scope in this paper. However, processing of more than 100k samples put limitations and require non-trivial approaches to handle such amount of files. We discovered that common ways of working with files in directory such that simple *ls* and *mv* in *bash* take unreasonable amount of time to execute. Also there is no way to distinguish files by extension like **.dll* or **.exe* since the names are just *md5* sums. So, following filtering steps were performed:

1. Heap of unfiltered malware and benign files were placed into two directories "malware/" and "benign/".

2. To eliminate duplicates, we renamed all the files to their MD5 sums.

3. PE32 files were detected in each folder using *file* Linux command:

```
\$ file 000000b4dccfbaa5bd981af2c1bbf59a
000000b4dccfbaa5bd981af2c1bbf59a: PE32 executable (DLL) (GUI) Intel
    ↪ 80386, for MS Windows
```

4. All PE32 files from current directory that meet our requirement were scrapped and move to a dedicated one:

```
#!/bin/sh
cd ../windows1;
counter=0;
for i in *; do
counter=$((counter+1));
echo "$counter";
VAR="file␣$i␣|␣grep␣PE32␣" ;
 VAR1=$(eval "$VAR") ;
 len1=${#VAR1};
if [ -n "$VAR1" ] && [ "$len1" -gt "1" ] ;
then
 echo "$VAR1" | awk '{print␣$1}' | awk '{gsub(/:$/,"");␣print␣$1␣"␣
     ↪ ../windows/PE/"␣$1}'| xargs mv -f ;
else
 echo "other";
 file $i | awk '{print␣$1}' | awk '{gsub(/:$/,"");␣print␣$1␣"␣../
     ↪ windows/other/"␣$1}' | xargs mv -f ;
 fi
done
```

5. We further can see a variety of PE32 modifications for 32bit architecture:

```
PE32 executable (GUI) Intel 80386, for MS Windows
PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed
```

   Following our purpose to concentrate on 32bit architecture, only PE32 are filtered out from all possible variants of PE32 files shown about.

6. After extracting a target group of benign and malicious PE32 files, multiple rounds of feature extraction are performed according to methods used in the literature.

7. Finally, we insert extracted features into the corresponding MySQL database to ease the handling, feature selection and machine learning processes respectively.

After collecting all possible files and performing the pre-processing phase, we ended up with the sets represented in the Table 12.2.

Table 12.2: Characteristics of the dataset collected and used for our experiments after filtering PE files

Dataset	Number of files	Size
Benign	16,632	7.4GB
Malware_000	58,023	14.0GB
Malware_207	41,899	16.0GB

Further, feature construction and extraction routine from PE files was performed using several tools as follows:

1. PEFRAME [3] is an open source tool specifically designed for static analysis of PE malware. It extracts various information from PE header ranging from packers to anti debug and anti vm tricks.

2. HEXDUMP is a standard Linux command line tool which is used to display a file in specific format like ASCII or one-byte octal.

3. OBJDUMP is a standard Linux command line too to detect applications instructions, consumed memory addresses, etc.

12.5 Results & Discussions

Before testing different ML techniques for malware detection it is important to show that our datasets actually represent the real-world distribution of the malware and goodware. Comparison of "Compile Time" field of PE32 header can be utilized for this purpose. Figure 12.7 represents log-scale histogram of the compilation time for our benign dataset. Taking into consideration the Windows OS timeline we found a harmony between our benign dataset applications compile time and development of Microsoft Windows operating systems. To start with, **Windows 3.1** was originally released on April 6, 1992 and our plot of benign applications indicates the biggest spike in early 1992. Later on in 1990th, **Windows 95** was due on 24 August 1995, while next **Windows 98** was announced on 25 June 1998. Further, 2000th marked release of **Windows XP** on October 25, 2001. Next phases on the plot correspond to the release of **Windows Vista** on 30th January 2007 and **Windows 7** on 22nd October 2009. Next popular version (Windows 8) appeared on 26th October 2012 and the latest major spike in the end of 2014 corresponds to the release of **Windows 10** on 29th July 2015.

Further, compilation time distribution for the first malware dataset *malware_-000* is given in the Figure 12.8. We can clearly see that release of newer Windows version is always followed by an increase of cumulative distribution of malware samples in following 6 to 12 month. It can be seen that the release of 32bit Windows 3.1 cause a spike in a number of malware. After this the number of malware compiled each year is constantly growing. Then, another increase can be observed in second half of the 2001 which corresponds to the release of the Windows XP and so on.

Considering the fact that MS DOS was released in 1981 it makes compilation times before this day look like fake or just obfuscated intentionally. On the other hand the dataset *malware_000* cannot have dates later than June 2012. Therefore, malware with compilation time prior to 1981 or later than Jan 2012 are tampered

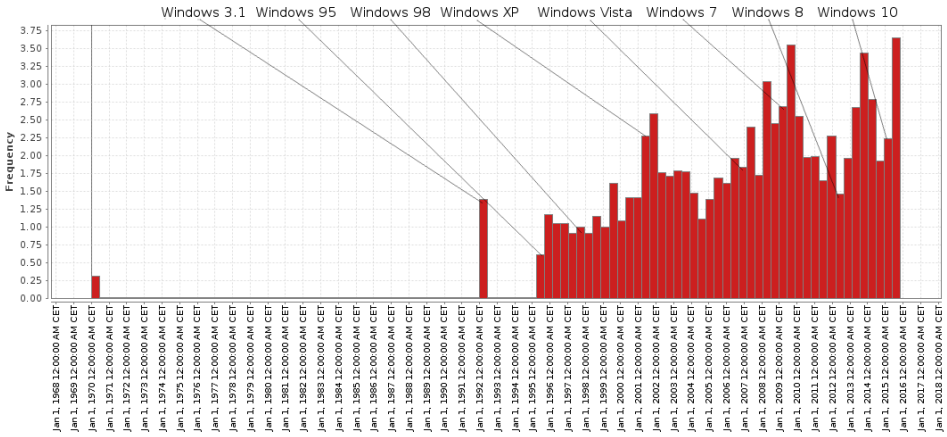


Figure 12.7: Log-scale histogram of compilation times for *benign* dataset

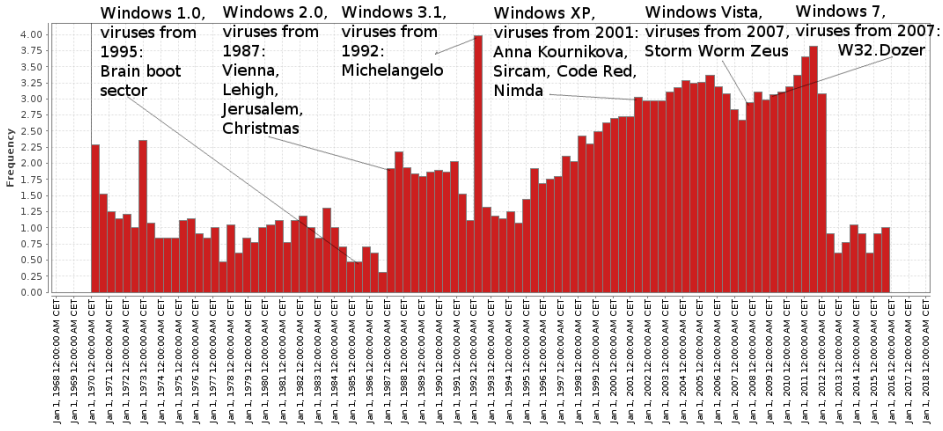


Figure 12.8: Log-scale histogram of compilation times for *malware_000* dataset

12.5.1 Accuracy of ML-aided Malware Detection using Static Characteristics

This part presents results of apply Naive Bayes, BayesNet, C4.5, k-NN, SVM, ANN and NF machine learning algorithms against static features of our dataset namely PE32 header, Bytes n-gram, Opcode n-gram, and API calls n-gram.

PE32 header

PE32 header is one of the most important features relevant to threat intelligence of PE32 applications. We performed feature selection using *Cfs* and *InfoGain* methods with 5-fold cross-validation as presented in the Table 12.3.

We can clearly see that the features from the *Short Info* section in PE32 head-

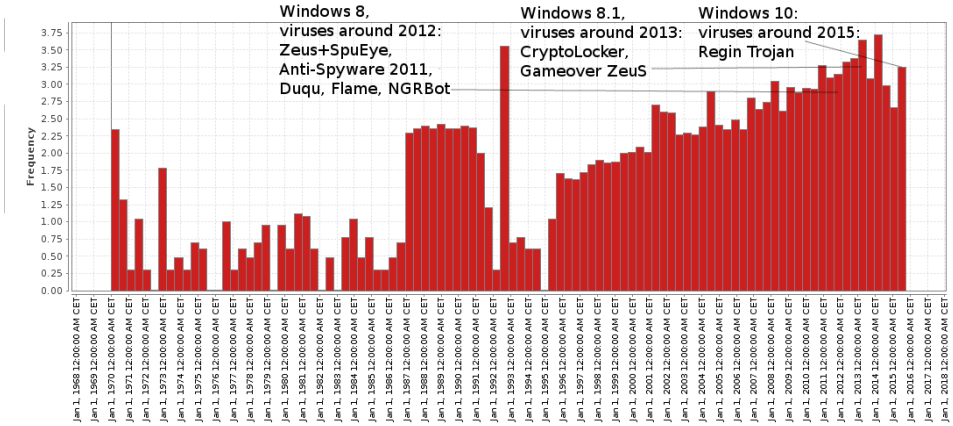


Figure 12.9: Log-scale histogram of compilation times for *malware_207* dataset

Table 12.3: Feature selection on PE32 features. Bold font denotes selected features according to *InfoGain* method

Benign vs Malware_000		**Benign vs Malware_207**		**Malware_000 vs Malware_207**	
Information Gain					
merit	attribute	merit	attribute	merit	attribute
0.377	**ShortInfo_Directories**	0.369	**ShortInfo_DLL**	0.131	**ShortInfo_FileSize**
0.278	**ShortInfo_DLL**	0.252	**ShortInfo_Directories**	0.094	ShortInfo_Detected
0.118	**AntiDebug**	0.142	**ShortInfo_FileSize**	0.064	SuspiciousAPI
0.099	Packer	0.105	**SuspiciousSections**	0.044	ShortInfo_Directories
0.088	SuspiciousSections	0.101	**SuspiciousAPI**	0.036	Packer
0.082	ShortInfo_Xor	0.089	AntiDebug	0.028	AntiDebug
0.076	SuspiciousAPI	0.084	ShortInfo_Detected	0.017	SuspiciousSections
0.045	ShortInfo_FileSize	0.054	ShortInfo_Xor	0.016	Url
0.034	ShortInfo_Detected	0.050	Packer	0.015	AntiVM
0.022	Url	0.036	Url	0.012	ShortInfo_Xor
0.004	AntiVM	0.002	AntiVM	0.002	ShortInfo_DLL
0	ShortInfo_Sections	0	ShortInfo_Sections	0	ShortInfo_Sections
0	DigitalSignature	0	DigitalSignature	0	DigitalSignature
Cfs					
attribute		attribute		attribute	
ShortInfo_Directories		ShortInfo_Directories		ShortInfo_Directories	
ShortInfo_Xor		ShortInfo_Xor		ShortInfo_FileSize	
ShortInfo_DLL		ShortInfo_DLL		ShortInfo_Detected	
ShortInfo_Detected				Packer	
Url					

ers can be used as a stand-alone malware indicators, including different epochs. Number of directories in this section as well as file size and flag of EXE or DLL

have bigger merits in comparison to other features. To contrary, *Anti Debug* and *Suspicious API* sections from *PEframe* cannot classify a binary file. Finally, we can say that digital signature and *Anti VM* files in PE32 headers are almost irrelevant in malware detection. Further, we performed exploration of selected ML methods that can be used with selected features. By extracting corresponding numerical features mentioned earlier, we were able to achieve classification accuracy levels presented in Table 12.4. Table 12.3 presents also accuracy of ML method after performing feature selection. Here we used whole sub-sets defined by *Cfs* method and features with merit of ≥ 0.1 detected by *InfoGain*.

Table 12.4: Comparative classification accuracy based on features from PE32 header, in %. Bn, Ml_000 and Ml_207 are benign and two malaware datasets respectively

Dataset	Naive Bayes	BayesNet	C4.5	k-NN	SVM	ANN	NF
All features							
Bn vs Ml_000	90.29	91.42	**97.63**	97.30	87.75	95.08	92.46
Bn vs Ml_207	88.27	91.21	**96.43**	95.99	84.88	93.24	89.03
Ml_000 vs Ml_207	63.41	71.59	**82.45**	82.11	73.77	69.99	69.01
Information Gain							
Bn vs Ml_000	88.32	89.17	**94.09**	94.01	94.09	93.51	87.53
Bn vs Ml_207	87.25	90.39	**95.06**	94.58	84.55	92.37	87.88
Ml_000 vs Ml_207	58.26	67.05	67.77	**70.70**	69.46	63.19	51.31
Cfs							
Bn vs Ml_000	89.35	90.89	**95.39**	95.38	95.16	93.69	85.85
Bn vs Ml_207	86.88	89.67	91.61	**91.68**	91.68	91.68	81.91
Ml_000 vs Ml_207	67.45	70.95	76.98	**76.92**	72.15	68.18	67.06

Malware and goodware can be easily classified using full set as well as sub-set of features. One can notice that ANN and C4.5 performed much better than other methods. It can be also seen that the high quality of these features made them very appropriate to differentiate between the *benign* and *malware_000* dataset. Further, we can see that the two datasets *malware_000* and *malware_207* are similar and extracted features do not provide a high classification accuracy. Neural Network was used with 3 hidden layers making it a non-linear model and experiments were performed using 5-fold cross-validation technique.

Bytes n-gram

Bytes n-gram is a very popular method for static analysis of binary executables. This method has one significant benefit: in order to perform analysis there is no need of previous knowledge about file type and internal structure since we use its raw (binary) form. For feature construction we used random profiles that were first presented by Ebringer et al. [17] called *fixed sample count* (see Figure 12.10), which generates fixed number of random profiles regardless of the file size and

sliding window algorithm. In this method each file is represented in a hexadecimal format and frequencies of each byte are counted to build a Huffman tree for each file. Then using window of fixed size and moving it on fixed skip size the randomness profile of each window is calculated. A Randomness profile is sum of Huffman code length of each byte in a single window. The lower the randomness in a particular window the bigger will be the randomness of that profile.

Generate randomness measurements for a file, output proportional to file length

INPUT: An array of bytes $b_1, \ldots, b_n$, a window size w and a skip size a.

OUTPUT: An array of $\left\lceil \frac{n-w}{a} \right\rceil$ samples of the randomness, ranging from 0.0 to1.0.

1. Construct the Huffman tree for the input bytes.
2. Construct an array $e_1, \ldots, e_{256}$ containing the encoding length for each of the input bytes.
3. For i from 1 to $\left\lceil \frac{n-w}{a} \right\rceil$, do the following:
   a) $r_i \leftarrow \sum_{j=a(i-1)+1}^{a(i-1)+w+1} e_j$
4. Rescale r_i between 0.0 and 1.0, where $\min(r_i) = 0.0$ and $\max(r_i) = 1.0$.

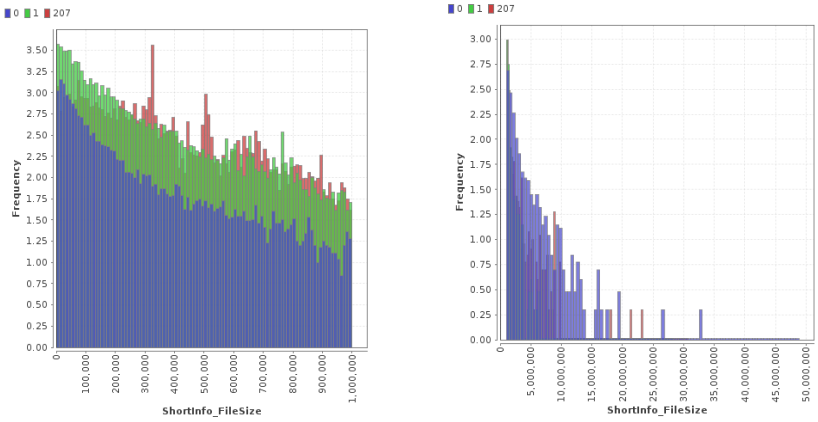
Figure 12.10: Sliding window algorithm [17]

We chose 32 bytes as the most promising sliding window size [17, 48, 73] and due to big variety of file sizes in our dataset, we chose 30 best features (or *pruning size* in terminology from [17]) which are the areas of biggest randomness (the most unique parts) in their original order. This features was fed into different machine learning algorithms as shown in Table 12.5. Our results indicate that the accuracy of this technique is not that high as it was originally developed to preserve local details of a file ([17])while the size of file affects localness a lot. In our case file sizes vary from around 0.5Kb to 53.7Mb which adversely affect the results. Despite worse results it is still easier to distinguish between benign executables and malware than between malware from different time slices. Also we can see that ANN is better in *Benign vs Malware_000* dataset, C4.5 in *Benign vs Malware_207* and *Malware_000 vs Malware_207* datasets.

Also it should be noted that we did not use feature selection methods as in the case of PE32 header features. Both Information gain and Cfs are not efficient due to the similarity of features and equivalence in importance for classification process. For the first dataset the Information Gain was in the range of 0.0473-0.0672 while for the second dataset it was in the range of 0.0672-0.1304 and for the last it was 0.0499-0.0725. Moreover, Cfs produces best feature subset nearly

Table 12.5: Classification accuracy based on features from bytes n-gram randomness profiles, in %

Dataset	Naive Bayes	BayesNet	C4.5	k-NN	SVM	ANN	NF
All features							
Bn vs Ml_000	69.9	60.4	76.9	75.6	78.3	78.3	74.8
Bn vs Ml_207	70.3	68.2	75.8	75.6	72.1	71.6	68.2
Ml_000 vs Ml_207	50.1	64.0	68.1	64.7	58.1	60.1	58.2



(a) Less than 1M **(b)** Bigger than 1M

Figure 12.11: Distribution of file size values in Bytes for three classes

equal to full set. Therefore, we decided to use all features as there is no subset that could possibly be better than original one.

Opcode n-gram

Opcode n-gram consists of assembly instructions which construct the executable file. The main limitation of this method is that in order to gain opcodes we need disassemble an application which sometimes fails to give correct opcodes due to different anti-disassembly and packing techniques used in executables hence we filtered out this kind of files from our dataset. We extracted 100 most common 3- and 4-grams from each of three file sets in our dataset. Then we extracted a set of 200 most common n-grams - which are called feature n-grams - to build a presence vector where value 1 was assigned if a certain n-gram from feature n-grams is present in top 100 most used n-grams of the file. Table 12.6 represents results of feature selection performed on the dataset with 3-grams. As can be seen the first two pair of datasets have a lot of common n-grams, while selected n-grams for the third pair of dataset is totally different. For Information Gain the threshold of

0.1 was used for both benign and malware datasets, while for the last set we used InfoGain of 0.02.

These data were passed to machine learning algorithms and results are shown in Tables 12.7 and 12.9. As can be seen C4.5 performed well and had the highest accuracy almost in all experiments. Also feature selection significantly reduced the number of n-grams from 200 down to 10-15, while overall accuracy on all methods did not dropped significantly. In fact, Naive Bayes performed even better that can be justified by reduced complexity of the probabilistic model. Also NF showed much better accuracy in comparison to other methods when using all features to distinguish between two malware datasets which can be linked to non-linear correlation in the data that are circumscribed in the Gaussian fuzzy patches.

Table 12.6: Feature selection on 3-gram opcode features. Bold font denotes features that present in both datasets that include nenign samples

Benign vs Malware_000		**Benign vs Malware_207**		**Malware_000 vs Malware_207**	
Information Gain					
merit	attribute	merit	attribute	merit	attribute
0.302483	**int3movpush**	0.298812	**int3movpush**	0.042229	pushlcallpushl
0.283229	**int3int3mov**	0.279371	**int3int3mov**	0.039779	movtestjne
0.266485	**popretint3**	0.227489	**popretint3**	0.037087	callpushlcall
0.236949	**retint3int3**	0.202162	**retint3int3**	0.031045	pushpushlcall
0.191866	**jmpint3int3**	0.193938	**jmpint3int3**		
0.134709	callmovtest	0.108580	retpushmov		
0.133258	movtestje				
0.115976	callmovpop				
0.114482	testjemov				
0.101328	poppopret				
0.100371	movtestjne				
Cfs					
attribute		attribute		attribute	
movtestje		movmovadd		pushpushlcall	
callmovtest		retpushmov		movtestjne	
callmovpop		xormovmov		movmovjmp	
retint3int3		**callmovtest**		jecmpje	
popretint3		**popretint3**		cmpjepush	
pushmovadd		**pushmovadd**		pushleacall	
int3int3mov		**int3int3mov**		callpopret	
callmovjmp		**callmovjmp**		leaveretpush	
jmpint3int3		**jmpint3int3**		pushmovadd	
int3movpush		**int3movpush**		pushcalllea	
				callpushlcall	
				callmovlea	
				pushlcallpushl	
				movmovmovl	
				calljmpmov	

Further, we investigated if there is any correlation between n-grams in files that

Table 12.7: Classification accuracy based on features from opcode 3-gram, in %

Dataset	Naive Bayes	BayesNet	C4.5	k-NN	SVM	ANN	NF
All features							
Bn vs Ml_000	83.51	83.52	**95.53**	93.82	94.43	94.51	95.28
Bn vs Ml_207	84.52	84.52	**93.93**	91.84	92.32	92.44	93.20
Mn_000 vs Ml_207	63.73	63.73	81.21	78.64	75.42	76.64	**83.13**
Information Gain							
Bn vs Ml_000	86.74	86.94	90.41	**90.45**	89.98	90.26	84.45
Bn vs Ml_207	86.22	86.22	86.22	86.22	87.46	**87.48**	83.36
Mn_000 vs Ml_207	63.19	62.55	71.19	**71.89**	69.54	67.36	69.14
Cfs							
Bn vs Ml_000	87.79	88.66	91.15	**91.22**	90.90	90.82	85.31
Bn vs Ml_207	86.24	86.33	**89.92**	89.73	89.17	89.34	81.58
Mn_000 vs Ml_207	86.24	86.33	**89.92**	89.73	89.17	89.34	69.25

belong to both benign and malicious classes. We extracted relative frequency of each n-gram according to the following formula $h_{n-gram} = \frac{N^{Class}_{files \in n-gram}}{N^{Class}_{files}}$, where $N^{Class}_{files \in n-gram}$ indicates number of files in class that has n-gram and N^{Class}_{files} is a total number of files in this class. The results for 3-gram is depicted in the Figure 12.12. As a reference we took top 20 most frequent n-grams from benign class and found frequency of the corresponding n-grams from both malware datasets. It can be seen that the frequency does not differ fundamentally, yet n-grams for both malicious classes tend to have very close numbers in comparison to benign files. Moreover, there is a clear dependency between both malicious classes. We also can notice that most of the features selected from two datasets that includes benign samples are same. This highlights reliability of the selected 4-grams and generalization of this method for malware detection.

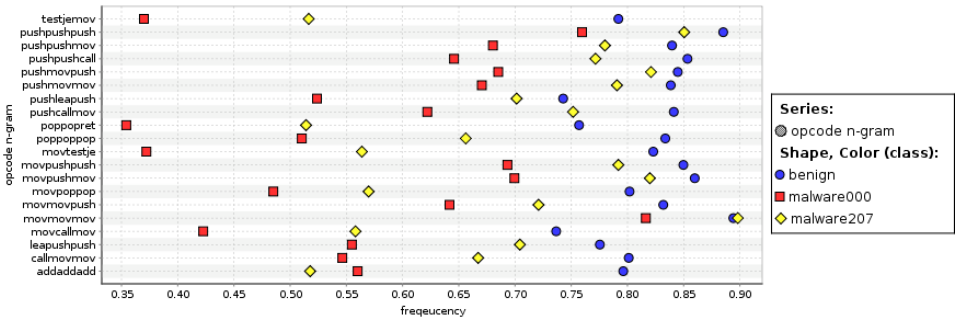


Figure 12.12: Distribution of the frequencies of top 20 opcode 3-grams from benign set in comparison to both malicious datasets

Additionally, we studied 4-gram features and extracted 200 features as shown in Table 12.8. Similar to the 3-grams features selected in the Table 12.6 one can see that two first pairs of datasets have a lot of common features, while the last one provides a significantly different set. As in case with 3-grams we used Information Gain with threshold of 0.1 for both benign and the first malware dataset, while for the last malware set we used InfoGain of 0.02, which looks reasonable with respect to number of selected‘ features.

Table 12.8: Feature selection on on 4-gram opcode features. Bold font denotes features that present in both datasets that include nenign samples

Benign vs Malware_000		**Benign vs Malware_207**		**Malware_000 vs Malware_207**	
Information Gain					
merit	attribute	merit	attribute	merit	attribute
0.303209	**int3int3movpush**	0.295427	**int3int3movpush**	0.047452	pushlcallpushlcall
0.295280	**int3movpushmov**	0.286378	**int3movpushmov**	0.045860	movpoppopret
0.285608	**int3int3int3mov**	0.266966	**int3int3int3mov**	0.044750	jepushcallpop
0.258733	**popretint3int3**	0.229431	**jmpint3int3int3**	0.044573	callpushlcallpushl
0.241215	**poppopretint3**	0.224318	**poppopretint3**	0.038822	cmpjepushcall
0.233205	**jmpint3int3int3**	0.210289	**popretint3int3**	0.035731	pushcallpopret
0.220679	**retint3int3int3**	0.170367	**retint3int3int3**	0.030460	pushcallpopmov
0.185178	**movpopretint3**	0.148442	**movpopretint3**	0.028564	movcmpjepush
0.151337	**movpushmovsub**	0.116760	**movpushmovsub**	0.025813	cmpjecmpje
0.125703	pushcallmovtest	0.103841	**movpushmovpush**	0.024372	leaveretpushmov
0.104993	**movpushmovpush**	0.102730	**movpushmovmov**	0.023374	pushpushpushlcall
0.104416	**movpushmovmov**			0.022312	pushcallpoppop
				0.021929	movtestjepush
				0.020003	pushpushleapush
Cfs					
attribute		attribute		attribute	
incaddincadd		addaddaddadd		leaveretpushmov	
movpushmovsub		movmovpushpush		callmovtestje	
jmpmovmovmov		**movpushmovsub**		jepushcallpop	
pushcallmovtest		**pushcallmovtest**		pushlcallpushlcall	
int3int3int3mov		**int3int3int3mov**		pushpushpushlea	
movpoppopret		movxormovmov		jecmpjecmp	
jmpint3int3int3		pushlcallpushlcall		movpoppopret	
movpopretint3		**jmpint3int3int3**		pushcallmovpush	
int3int3movpush		movpopretint3		pushmovmovcall	
int3movpushmov		**int3int3movpush**		movpopretint3	
poppopretint3		**int3movpushmov**		cmpjepushcall	
addpushpushpush		**poppopretint3**		movleamovmov	
pushpushcalllea				movmovjmpmov	
				pushpushcalllea	
				retnopnopnop	
				movaddpushpush	
				subpushpushpush	

The classification performance is given in Figure 12.9. As can be seen, 3-

grams can show a bit better result than 4-grams in case of distinguishing between benign and malware_000 or Benign and malware_207 with C4.5 classifier. At the same time 4-grams are better in order to distinguish between two malware datasets with C4.5 classifier. We can conclude that results are quite good, and can be used for malware detection. In our opinion results can be improved by extracting more features and usage of relative frequencies rather than pure vectors.

Table 12.9: Classification accuracy based on features from opcode 4-gram, in %

Dataset	Naive Bayes	BayesNet	C4.5	k-NN	SVM	ANN	NF
All features							
Bn vs Ml_000	86.92	86.92	95.31	93.73	94.28	94.23	**95.54**
Bn vs Ml_207	86.84	86.84	93.33	91.71	92.03	92.04	**93.75**
Ml_000 vs Ml_207	64.90	64.90	**81.58**	78.98	74.98	75.77	78.80
Information Gain							
Bn vs Ml_000	87.79	87.89	**91.48**	91.45	91.31	90.84	85.74
Bn vs Ml_207	84.64	84.57	**87.84**	87.83	87.25	87.70	48.67
Mn_000 vs Ml_207	62.73	63.20	69.96	**70.25**	68.40	67.24	68.90
Cfs							
Bn vs Ml_000	89.63	89.63	91.51	**91.52**	91.52	90.76	84.95
Bn vs Ml_207	86.41	86.64	89.36	**89.48**	89.16	89.12	81.13
Mn_000 vs Ml_207	66.28	66.17	72.00	**72.27**	68.96	69.17	69.32

In contrary to 3-grams we can see that the histograms of 4-grams have fundamental differences when it comes to malicious and benign sets as it is depicted in Figure 12.13. We can see that the frequencies correspond to malware_000 and malware_207 datasets are nearly similar and are far from the frequencies detected for the benign class. Moreover, there is a clear and strong correlation between two malware datasets. So, we can conclude that in case of probabilistic-based models like Bayes Network and Naive Bayes the classification could be a bit better due to differences in likelihood of appearance, which can be also found in Tables 12.7 and 12.9.

API call n-grams

API calls n-grams is the combination of specific operations invoked by the process in order to use functionalities of an operation system. In this study we used *peframe* to extract API calls from PE32 files. The bigger the n-gram size is the lower accuracy is possible to gain. The reason for this is that single API calls and their n-grams are far fewer in comparison with for example opcode n-grams. After extraction of API calls, we combined them into 1- and 2-grams. For each task we selected 100 most frequent features in a particular class and combined them into 200-feature vectors. Tables 12.10 and 12.11 presents results of machine learning evaluation on API call n-grams data.

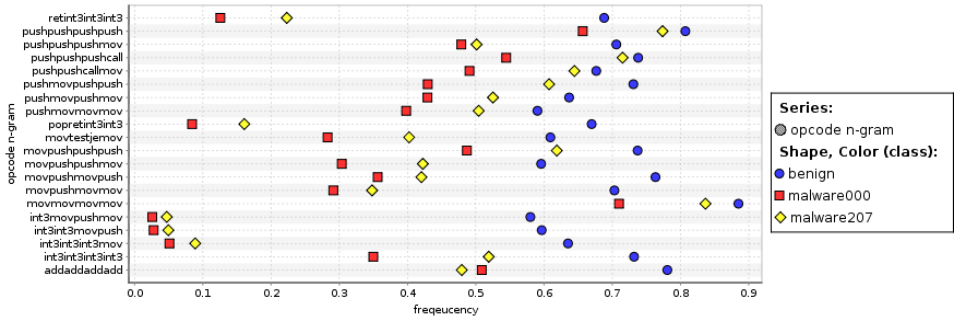


Figure 12.13: Distribution of the frequencies of top 20 opcode 4-grams from benign set in comparison to both malicious datasets

As we can see ANN, k-NN and C4.5 are the best classifiers similar to previous results. It is also more difficult to distinguish between files from *malware_000* and *malware_207*. We gained quite high accuracy, but it is still lower than in related studies. It could be explained by the size of datasets: other studies datasets usually consist of several hundreds or thousands of files while our dataset has more than 110,000 files. After analysing feature selection results we decided not to include them in the results section since most of the features are similar in terms of distinguishing between malware and goodware. It means that there is large number of unique API calls that can be found once or twice in a file in contrary to the byte or opcode n-gram

Table 12.10: Classification accuracy based on API call 1-gram features, %

Dataset	Naive Bayes	BayesNet	C4.5	k-NN	SVM	ANN	NF
			All features				
Bn vs Ml_000	90.79	90.79	93.39	93.47	**93.51**	93.43	82.44
Bn vs Ml_207	87.18	87.18	90.94	91.03	**91.37**	91.23	81.28
Ml_000 vs Ml_207	66.19	66.2	**78.44**	77.09	73.33	72.77	73.55

Table 12.11: Classification accuracy based on API call 2-gram features, %

Dataset	Naive Bayes	BayesNet	C4.5	k-NN	SVM	ANN	NF
			All features				
Bn vs Ml_000	86.54	86.55	90.88	91.53	**91.96**	91.85	75.24
Bn vs Ml_207	81.94	81.91	87.84	**88.82**	88.31	87.81	83.61
Ml_000 vs Ml_207	62.31	62.31	**73.69**	73.17	70.27	69.45	70.08

We also studied the difference between frequencies distributions of API calls. Figure 12.14 sketches extracted API 1-grams from three datasets. One can see

that there is a significant spread between numbers of occurrences in benign class in contrary to both malicious datasets. On the other hand, results for both malware datasets are similar, which indicates statistical significance of extracted features. It is important ho highlight that the largest scatter are in frequencies for *memset()*, *malloc()* and *free()* API calls. On the other hand, malicious programs tend to use *GetProcAddress()* function more often for retrieving the address of any function from dynamic-link libraries in the system.

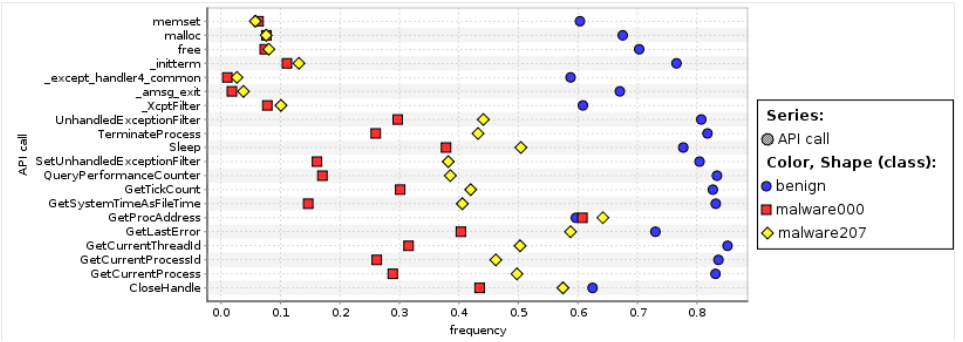


Figure 12.14: frequencies of 20 most frequent API 1-grams for three different datasets

12.6 Conclusion

In this paper we presented a survey on applications of machine learning techniques for static analysis of PE32 Windows malware. First, we elaborated on different methods for extracting static characteristics of the executable files. Second, an overview of different machine learning methods utilized for classification of static characteristics of PE32 files was given. In addition, we offered a taxonomy of malware static features and corresponding ML methods. Finally, we provided a tutorial on how to apply different ML methods on benign and malware dataset for classification. We found that C4.5 and k-NN in most cases perform better than other methods, while SVM and ANN on some feature sets showed good performance. On the other hand Bayes Network and Naive Bayes have poor performance compared to other ML methods. This can be explained by negligibly low probabilities which present in a large number of features such as opcode and bytes n-grams. So, it can see that static-analysis using ML is a fast and reliable mechanism to classify malicious and benign samples considering different characteristic of PE32 executables. Machine Learning- aided static malware analysis can be used as part of Cyber Threat Intelligence (CTI) activities to automate detection of indications of compromise from static features of PE32 Windows files.

12.7 Bibliography

[1] Infection rates and end of support for windows xp. https://blogs.technet.microsoft.com/mmpc/2013/10/29/infection-rates-and-end-of-support-for-windows-xp/. accessed: 01.04.2016.

[2] Vx heaven. http://vxheaven.org/. accessed: 25.10.2015.

[3] Gianni Amato. Peframe. https://github.com/guelfoweb/peframe. accessed: 27.10.2016.

[4] M. Baig, P. Zavarsky, R. Ruhl, and D. Lindskog. The study of evasion of packed pe from static detection. In *Internet Security (WorldCIS), 2012 World Congress on*, pages 99–104, June 2012.

[5] Simen Rune Bragen. Malware detection through opcode sequence analysis using machine learning. Master's thesis, Gjøvik University College, 2015.

[6] C. Cepeda, D. L. C. Tien, and P. Ordóñez. Feature selection and improving classification performance for malware detection. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pages 560–566, Oct 2016.

[7] Mohsen Damshenas, Ali Dehghantanha, and Ramlan Mahmoud. A survey on malware propagation, analysis, and detection. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2(4):10–29, 2013.

[8] F. Daryabar, A. Dehghantanha, and N. I. Udzir. Investigation of bypassing malware defences and malware detections. In *2011 7th International Conference on Information Assurance and Security (IAS)*, pages 173–178, Dec 2011.

[9] Farid Daryabar, Ali Dehghantanha, and Hoorang Ghasem Broujerdi. Investigation of malware defence and detection techniques. *International Journal of Digital Information and Wireless Communications (IJDIWC)*, 1(3):645–650, 2011.

[10] Farid Daryabar, Ali Dehghantanha, Brett Eterovic-Soric, and Kim-Kwang Raymond Choo. Forensic investigation of onedrive, box, googledrive and dropbox applications on android and ios devices. *Australian Journal of Forensic Sciences*, 48(6):615–642, 2016.

[11] Farid Daryabar, Ali Dehghantanha, Nur Izura Udzir, Solahuddin bin Shamsuddin, et al. Towards secure model for scada systems. In *Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on*, pages 60–64. IEEE, 2012.

[12] Farid Daryabar, Ali Dehghantanha, Nur Izura Udzir, et al. A review on impacts of cloud computing on digital forensics. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2(2):77–94, 2013.

[13] Ali Dehghantanha and Katrin Franke. Privacy-respecting digital investigation. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pages 129–138. IEEE, 2014.

[14] Dhruwajita Devi and Sukumar Nandi. Detection of packed malware. In *Proceedings of the First International Conference on Security of Internet of Things*, SecurIT '12, pages 22–26, New York, NY, USA, 2012. ACM.

[15] Dennis Distler and Charles Hornat. Malware analysis: An introduction. *SANS Institute InfoSec Reading Room*, pages 18–19, 2007.

[16] T. Dube, R. Raines, G. Peterson, K. Bauer, M. Grimaila, and S. Rogers. Malware type recognition and cyber situational awareness. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on*, pages 938–943, Aug 2010.

[17] Tim Ebringer, Li Sun, and Serdar Boztas. A fast randomness test that preserves local detail. *Virus Bulletin*, 2008, 2008.

[18] Parvez Faruki, Vijay Laxmi, M. S. Gaur, and P. Vinod. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks*, SIN '12, pages 130–137, New York, NY, USA, 2012. ACM.

[19] Anders Flaglien, Katrin Franke, and Andre Arnes. Identifying malware using cross-evidence correlation. In *IFIP International Conference on Digital Forensics*, pages 169–182. Springer Berlin Heidelberg, 2011.

[20] Tristan Fletcher. Support vector machines explained. *[Online]. http://sutikno. blog. undip. ac. id/files/2011/11/SVM-Explained. pdf.[Accessed 06 06 2013]*, 2009.

[21] Katrin Franke, Erik Hjelmås, and Stephen D Wolthusen. Advancing digital forensics. In *IFIP World Conference on Information Security Education*, pages 288–295. Springer Berlin Heidelberg, 2009.

[22] Katrin Franke and Sargur N Srihari. Computational forensics: Towards hybrid-intelligent crime investigation. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 383–386. IEEE, 2007.

[23] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[24] Mark A Hall and Lloyd A Smith. Practical feature subset selection for machine learning. *Proceedings of the 21st Australasian Computer Science Conference ACSC'98*, 1998.

[25] Chris Hoffman. How to keep your pc secure when microsoft ends windows xp support. http://www.pcworld.com/article/2102606/how-to-keep-your-pc-secure-when-microsoft-ends-windows-xp-support.html. accessed: 18.04.2016.

[26] Anil K Jain, Robert PW Duin, and Jianchang Mao. Statistical pattern recognition: A review. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(1):4–37, 2000.

[27] Sachin Jain and Yogesh Kumar Meena. Byte level n–gram analysis for malware detection. In *Computer Networks and Intelligent Computing*, pages 51–59. Springer, 2011.

[28] C. McMillan K. Kendall. Practical malware analysis. In *Black Hat Conference USA*, 2007.

[29] Z. Khorsand and A. Hamzeh. A novel compression-based approach for malware detection using pe header. In *Information and Knowledge Technology (IKT), 2013 5th Conference on*, pages 127–133, May 2013.

[30] Teuvo Kohonen and Timo Honkela. Kohonen network. *Scholarpedia*, 2(1):1568, 2007.

[31] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 470–478, New York, NY, USA, 2004. ACM.

[32] Igor Kononenko and Matjaz Kukar. *Machine learning and data mining: introduction to principles and algorithms*. Horwood Publishing, 2007.

[33] S. Kumar, M. Azad, O. Gomez, and R. Valdez. Can microsoft's service pack2 (sp2) security software prevent smurf attacks? In *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*, pages 89–89, Feb 2006.

[34] Lastline. The threat of evasive malware. white paper, Lastline Labs, https://www.lastline.com/papers/evasive_threats.pdf, February 2013. accessed: 29.10.2015.

[35] N. A. Le-Khac and A. Linke. Control flow change in assembly as a classifier in malware analysis. In *2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, pages 38–43, April 2016.

[36] Woody Leonhard. Atms will still run windows xp – but a bigger shift in security looms. http://www.infoworld.com/article/2610392/microsoft-windows/atms-will-still-run-windows-xp----but-a-bigger-shift-in-security-looms.html, March 2014. accessed: 09.11.2015.

[37] R. J. Mangialardo and J. C. Duarte. Integrating static and dynamic malware analysis using machine learning. *IEEE Latin America Transactions*, 13(9):3080–3087, Sept 2015.

[38] Z. Markel and M. Bilzor. Building a machine learning classifier for malware detection. In *Anti-malware Testing Research (WATeR), 2014 Second Workshop on*, pages 1–4, Oct 2014.

[39] M.M. Masud, L. Khan, and B. Thuraisingham. A hybrid model to detect malicious executables. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 1443–1448, June 2007.

[40] Microsoft. Microsoft security essentials. http://windows.microsoft.com/en-us/windows/security-essentials-download. accessed: 18.04.2016.

[41] Microsoft. Set application-specific access permissions. https://technet.microsoft.com/en-us/library/cc731858%28v=ws.11%29.aspx. accessed: 30.05.2016.

[42] C. Miles, A. Lakhotia, C. LeDoux, A. Newsom, and V. Notani. Virusbattle: State-of-the-art malware analysis for better cyber threat intelligence. In *2014 7th International Symposium on Resilient Control Systems (ISRCS)*, pages 1–6, Aug 2014.

[43] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, 2017.

[44] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti. Employing program semantics for malware detection. *IEEE Transactions on Information Forensics and Security*, 10(12):2591–2604, Dec 2015.

[45] Farhood Norouzizadeh Dezfouli, Ali Dehghantanha, Brett Eterovic-Soric, and Kim-Kwang Raymond Choo. Investigating social networking applications on smartphones detecting facebook, twitter, linkedin and google+ artefacts on android and ios platforms. *Australian journal of forensic sciences*, 48(4):469–488, 2016.

[46] Opeyemi Osanaiye, Haibin Cai, Kim-Kwang Raymond Choo, Ali Dehghantanha, Zheng Xu, and Mqhele Dlodlo. Ensemble-based multi-filter feature selection method for ddos detection in cloud computing. *EURASIP Journal on Wireless Communications and Networking*, 2016(1):130, 2016.

[47] Hamed Haddad Pajouh, Reza Javidan, Raouf Khayami, Dehghantanha Ali, and Kim-Kwang Raymond Choo. A two-layer dimension reduction and two-tier classification model for anomaly-based intrusion detection in iot backbone networks. *IEEE Transactions on Emerging Topics in Computing*, 2016.

[48] Shuhui Qi, Ming Xu, and Ning Zheng. A malware variant detection method based on byte randomness test. *Journal of Computers*, 8(10):2469–2477, 2013.

[49] J. Ross Quinlan. Improved use of continuous attributes in c4. 5. *Journal of artificial intelligence research*, pages 77–90, 1996.

[50] RC Quinlan. 4.5: Programs for machine learning morgan kaufmann publishers inc. *San Francisco, USA*, 1993.

[51] D Krishna Sandeep Reddy and Arun K Pujari. N-gram analysis for computer virus detection. *Journal in Computer Virology*, 2(3):231–239, 2006.

[52] Seth Rosenblatt. Malwarebytes: With anti-exploit, we'll stop the worst attacks on pcs. http://www.cnet.com/news/malwarebytes-finally-unveils-freeware-exploit-killer/. accessed: 30.05.2016.

[53] Neil J. Rubenking. The best antivirus utilities for 2016. http://uk.pcmag.com/antivirus-reviews/8141/guide/the-best-antivirus-utilities-for-2016. accessed: 30.05.2016.

[54] Paul Rubens. 10 ways to keep windows xp machines secure. http://www.cio.com/article/2376575/windows-xp/10-ways-to-keep-windows-xp-machines-secure.html. accessed: 18.04.2016.

[55] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1020–1025, New York, NY, USA, 2010. ACM.

[56] S. Samtani, K. Chinn, C. Larson, and H. Chen. Azsecure hacker assets portal: Cyber threat intelligence and malware analysis. In *2016 IEEE Conference on Intelligence and Security Informatics (ISI)*, pages 19–24, Sept 2016.

[57] SANS. Who's using cyberthreat intelligence and how? https://www.sans.org/reading-room/whitepapers/analyst/cyberthreat-intelligence-how-35767. accessed: 01.03.2017.

[58] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[59] Igor Santos, Xabier Ugarte-Pedrero, Borja Sanz, Carlos Laorden, and Pablo G. Bringas. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference*, CEAS '11, pages 23–30, New York, NY, USA, 2011. ACM.

[60] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.

[61] Kaveh Shaerpour, Ali Dehghantanha, and Ramlan Mahmod. Trends in android malware detection. *The Journal of Digital Forensics, Security and Law: JDFSL*, 8(3):21, 2013.

[62] R.K. Shahzad, N. Lavesson, and H. Johnson. Accurate adware detection using opcode sequence extraction. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 189–195, Aug 2011.

[63] Andrii Shalaginov and Katrin Franke. Automated generation of fuzzy rules from large-scale network traffic analysis in digital forensics investigations. In *7th International Conference on Soft Computing and Pattern Recognition (SoCPaR 2015)*. IEEE, 2015.

[64] Andrii Shalaginov and Katrin Franke. A new method for an optimal som size determination in neuro-fuzzy for the digital forensics applications. In *Advances in Computational Intelligence*, pages 549–563. Springer International Publishing, 2015.

[65] Andrii Shalaginov and Katrin Franke. A new method of fuzzy patches construction in neuro-fuzzy for malware detection. In *IFSA-EUSFLAT*. Atlantis Press, 2015.

[66] Andrii Shalaginov and Katrin Franke. Automated intelligent multinomial classification of malware species using dynamic behavioural analysis. In *IEEE Privacy, Security and Trust 2016*, 2016.

[67] Andrii Shalaginov and Katrin Franke. Big data analytics by automated generation of fuzzy rules for network forensics readiness. *Applied Soft Computing*, 2016.

[68] Andrii Shalaginov and Katrin Franke. *Towards Improvement of Multinomial Classification Accuracy of Neuro-Fuzzy for Digital Forensics Applications*, pages 199–210. Springer International Publishing, Cham, 2016.

[69] Andrii Shalaginov, Katrin Franke, and Xiongwei Huang. Malware beaconing detection by mining large-scale dns logs for targeted attack identification. In *18th International Conference on Computational Intelligence in Security Information Systems. WASET*, 2016.

[70] Andrii Shalaginov, Lars Strande Grini, and Katrin Franke. Understanding neuro-fuzzy on a class of multinomial malware detection problems. In *IEEE International Joint Conference on Neural Networks (IJCNN 2016)*, Jul 2016.

[71] M. Shankarapani, K. Kancherla, S. Ramammoorthy, R. Movva, and S. Mukkamala. Kernel machines for malware classification and similarity analysis. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–6, July 2010.

[72] Muazzam Ahmed Siddiqui. *Data mining methods for malware detection*. ProQuest, 2008.

[73] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. Pattern recognition techniques for the classification of malware packers. In *Information security and privacy*, pages 370–390. Springer, 2010.

[74] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[75] Shugang Tang. The detection of trojan horse based on the data mining. In *Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09. Sixth International Conference on*, volume 1, pages 311–314, Aug 2009.

[76] X. Ugarte-Pedrero, I. Santos, P.G. Bringas, M. Gastesi, and J.M. Esparza. Semi-supervised learning for packed executable detection. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 342–346, Sept 2011.

[77] R Veeramani and Nitin Rai. Windows api based malware detection and framework analysis. In *International conference on networks and cyber security*, volume 25, 2012.

[78] VirusShare. Virusshare.com. http://virusshare.com/. accessed: 12.10.2020.

[79] C. Wang, Z. Qin, J. Zhang, and H. Yin. A malware variants detection methodology with an opcode based feature method and a fast density based clustering algorithm. In *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pages 481–487, Aug 2016.

[80] Tzu-Yen Wang, Chin-Hsiung Wu, and Chu-Cheng Hsieh. Detecting unknown malicious executables using portable executable headers. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 278–284, Aug 2009.

[81] Steve Watson and Ali Dehghantanha. Digital forensics: the missing piece of the internet of things promise. *Computer Fraud & Security*, 2016(6):5–8, 2016.

[82] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. An intelligent pe-malware detection system based on association mining. *Journal in computer virology*, 4(4):323–334, 2008.

[83] M.N.A. Zabidi, M.A. Maarof, and A. Zainal. Malware analysis with multiple features. In *Computer Modelling and Simulation (UKSim), 2012 UKSim 14th International Conference on*, pages 231–235, March 2012.

[84] Zongqu Zhao. A virus detection scheme based on features of control flow graph. In *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*, pages 943–947, Aug 2011.

ISBN 978-82-326-6061-2 (printed ver.)
ISBN 978-82-326-6679-9 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

NTNU
Norwegian University of
Science and Technology