Classifying evasive malware

GUSTAF EKENSTEIN
DAVID NORRESTAM
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Classifying evasive malware

Gustaf Ekenstein, David Norrestam
dat11ge1@student.lu.se fys11dno@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors: Paul Stankovski (LTH),
David Olander (SecureLink)

Examiner: Thomas Johansson

June 17, 2017

© 2017
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Malware are become increasingly aware of their execution environment. In order to avoid detection by automated analysis solutions and to obstruct manual analysis, malware authors are coming up with new ways for their malware to decide whether it should express its malicious behavior or not.

Previous solutions to this problem focus on for example improving the stealth of analysis environments (to avoid detection by malware), or analyzing differences in malware behavior when analyzed in different environments.

This thesis proposes an alternative approach to the problem. We perform automatic dynamic analysis on two sets of malware, containing samples known to be evasive and non-evasive respectively. The dynamic analysis produces logs of system calls, which are used to train a machine learning model, capable of detecting evasive behavior. This resulting model is a proof of concept that evasive behaviour can be detected. A possible use case for the model, is as part of a pipelined solution for malware detection. When testing the developed model, it was shown that it could correctly label 75% of all samples, with an equal success rate when considering only the labeling of evasive samples.

Table of Contents

Chapter 1

Introduction

This chapter provides an introduction to the thesis divided into the following parts;

Background Provides the reader with an introduction to the field of study.

Problem description Description of the problem this thesis is trying to tackle.

Method Method used in order to try and solve the problem.

Scope Limitations to the scope of the problem.

Motivation Motivates the necessity of this thesis.

Disposition Disposition of the rest of the thesis.

1.1 Background

One of the major challenges when analyzing potentially malicious files in an automated fashion is how to teach the analysis system to give the analyzed files a just verdict. Furthermore, the system needs to be confident in the given verdict to minimize the number of false positives and false negatives. More and more malware use evasive techniques to detect when they are analyzed [12], and thus the need to understand these evasive techniques to be able to counter them is of great importance.

A technique for analyzing suspicious and potentially malicious executable files is called dynamic analysis. Dynamic analysis can be either manual, such as manually debugging a file, or automated. In an automated dynamic analysis, a file is executed in an isolated environment, while its behavior is closely monitored. This isolated monitoring environment is called a sandbox. A file has practically an infinite number of potential actions or combinations of actions to take on a system, and defining which actions or combinations of actions are malicious is not a trivial task. The first step to perform dynamic analysis is to be able to run the malicious code in this isolated environment. If evasive techniques are used and it recognizes that it is run in a sandbox, the execution of the malicious code will be suspended. Therefore, finding a way to tell if evasive techniques are used, and how they are used, will be of help when designing these sandboxes so that they are able to counter the evasive techniques effectively forcing the malicious code to be executed.

1.2 Problem description

The aim of this Master's thesis is to analyze the nature of evasive malware, and attempt to create a machine learning distinguisher able to tell if the analyzed file is using evasive techniques. To do this, background research on evasive malware, and the techniques they deploy to avoid sandboxes will be performed. Since new malware is found every single day, the hope is that this distinguisher will detect patterns that will allow it to detect these zero-day threats. To accomplish this, the following questions will be attempted to be answered.

- Can a suitable set of feature vectors that enables this distinguisher be created in order to determine whether evasive techniques are utilized?
- Can these results be used in order to detect evasive behavior in zero-day threats?
- How well does this distinguisher perform in terms of true/false positives and negatives?

1.3 Method

The method used in this Master's thesis project consists of three steps:

- Gathering data,
- Dynamic analysis,
- Machine learning.

Background study was done on the subject of evasive malware to make sure that the samples labeled as evasive really are evasive. This knowledge was needed when designing the feature vectors in order to encapsulate the evasive behaviour in the best possible way. Studies on machine learning and deep learning approaches to classifying malware was performed.

The next step was to gather samples to train the machine learning model with. Since security is a very community-driven subject, there exist several open databases with malware samples ready to be downloaded. Discussions with analysts at SecureLink were held to make sure that appropriate samples were used. Manual analysis of the samples were done in order to label them as evasive/non-evasive. The samples were split into three sets; train, test, and validation.

A dynamic analysis system based on an open source hypervisor was then set up in order to run the malware and produce logs of system calls. These logs were then sanitized, and engineered in order to best encapsulate the evasive behavior when training and testing the data (see Chapter 3 for further details). Different machine learning algorithms were used, and the best fitting parameters were chosen before running the model on the test data. A statistical model was selected to measure the performance of this machine learning model in terms of random sampling along with measurements in recall, accuracy, precision, and f1-scores.

1.4 Scope

The scope of this thesis is limited as follows:

- The analyzed files are limited to Portable Executable files, or more specifically .exe files (.dll-files are not considered).
- Files that fail to execute in the analysis environment will not be considered.
- Evasive techniques will be defined as anti-virtual machine and anti-sandbox techniques. Executables utilizing only anti-debugging will not be considered evasive.

1.5 Motivation

Last year, a Master's thesis project was performed at Coresec Systems (now SecureLink Sweden), where the authors developed test cases in order to evade detection by sandbox solutions. The conclusions were that on average, 43.4% of all their test cases were not detected by any sandbox [14, page 60]. Even though combining sandboxes led to a lower pass rate for the malware (14%), this solution would neither be time nor cost effective.

After discussion with the authors of that Master's Thesis [14], the study of detecting evasive behaviour was decided to be carried out since it can be of great importance as a proof of concept for designing future sandbox solutions.

The amount of malware that exhibits evasive behaviour is increasing rapidly. According to Lastline [12], over 70% of all malware analyzed employed different techniques to avoid detection in virtual analysis environments. For malware analysts, this presents a major problem. To be able to analyze the malware properly, a sandbox will have to be created with these evasion techniques in mind, as to trick the malware into executing its malicious content in the analysis machine.

Extensive research has been done on the topic of defining and classifying malicious behaviour using machine learning, and deep learning, with good results [7, 11]. Many of the techniques utilized on this subject are limited to static analysis [26], or dynamic analysis where the malicious code is assumed to be executed [7].

1.6 Disposition

This thesis is organized as follows:

Chapter 1. Introduction provides a brief background to the area of study. Then, the problem at hand is described followed by the method used to solve this problem, the limitations of the scope, and a motivation for the choice of problem.

Chapter 2. Theory provides the reader with the underlying theory necessary in order to grasp the contents of this thesis.

Chapter 3. Method gives a thorough presentation of the process used to solve the problem, and motivations of choices made along the way.

Chapter 4. Test results contains the results obtained when testing the resulting machine-learning model, summarized in tables.

Chapter 5. Discussion contains a discussion of the results, and choices made during the development process. Also, possible sources of error are presented.

Chapter 6. Conclusions & Future work presents conclusions drawn based on the test results and the previous discussion. Also, suggestions for future work are given.

Chapter 2
Theory

This chapter provides the reader with the theoretical background needed in order to understand this thesis. It is divided into the following parts;

Malware Gives a brief introduction to malware, and how malware analysis works.

Hybrid-Analysis Gives a quick introduction to the online analysis system *Hybrid-Analysis*.

Xen Presents *Xen*, a hypervisor used in this thesis.

DRAKVUF Presents *DRAKVUF*, a solution used for dynamic analysis of malware.

Machine Learning Introduces the concept of machine learning, and provides detailed information about the process of feature engineering.

Windows registry Introduces the Windows registry.

Windows API Introduces the Windows API.

Related work Presents previous work performed within the field.

2.1 Malware

There are several different definitions of malware. Cisco states on their webpage [19] that "It is code or software that is specifically designed to damage, disrupt, steal, or in general inflict some other “bad” or illegitimate action on data, hosts, or networks.". Or, as stated by Avast [1]: "Malware is considered an annoying or harmful type of software intended to secretly access a device without the user’s knowledge.".

The definition of malware vary from source to source, but common for them all is that the result if the malicious code executes will affect the user in a negative manner. Also, even though the definitions of malware may vary, there are more or less standardized ways of classifying them into families depending on their behaviour. Malwarebytes released a report in which they describe the ongoing trends in malware [15]. One of the clearest trends of 2016 is the dramatic increase in *ransomware*, a type of malware which encrypts files on the target’s device and then demands payment in exchange for the decryption key [15]. Even though a recent report by Symantec shows that the yearly amount of new malware is largely unchanged between 2015 and 2016, over 350 million unique malware variants are detected for the first time each year [29].

2.1.1 Malware analysis

The goal of manual malware analysis could be described as: "Given a malware, what can we determine about its behaviour?" To determine the behaviour of the malicious file, the malware analyst tasked with the problem will utilize a variety of techniques. These analysis techniques can roughly be divided into two categories: static and dynamic analysis. Characteristic for static analysis is that the executable is examined without running it. This involves looking at its imported libraries, looking for strings in the file that could provide indicators of its intent (e.g. IP-addresses). A more advanced approach would be attempting to reverse engineer the executable to look at its instructions, in an attempt to understand what the program does [27, page 3].

Since static analysis does not include running the executable, it can be done in any environment without the risk of exposing the system to infection by the malicious software. The downside is that sometimes the software has been obfuscated by the malware author. This obfuscation can occur by *packing* the malware, in which case the malware is compressed and does not show its real behaviour until it is decompressed during runtime [27, page 383]. Unpacking the software may not always be possible, which limits static analysis. This is one of the reasons static analysis is often combined with dynamic analysis.

Dynamic analysis requires a safe environment to work in. This is usually achieved through the use of a virtual machine, often a software-emulated operating system with the ability to revert itself to a previous state through the use of snapshots. This ability allows the analyst to run the executable in the virtualized operating system while monitoring the system, and reverting back to before

infecting the system when the analysis is complete. The monitoring step that occurs while running the malware includes looking at network traffic, checking added/removed files, looking at edits to the registry, etc [27]. An advantage with this approach is that most likely, if the executable is malicious, it will introduce changes to the system (e.g. register itself for autostart). Dynamic analysis is a means to track these changes. Packed files can be handled just as any other files, since the unpacking behavior is triggered upon execution. The risks with using dynamic analysis, in contrast to static analysis, involves exposing the underlying system that is running the virtual machine to the malware. It is possible for malware to escape the virtual machine by exploiting vulnerabilities in the virtual machine [23].

As malware analysts are improving their methods for detecting malware, the malware authors are constantly looking for new ways to avoid their malware getting caught. One way for the malware authors to accomplish this is to implement different kinds of evasive techniques.

2.1.2 Evasive techniques

Considering the definition of evasive behavior stated in Section 1.4, the goal with embedding evasive techniques in malware, is to avoid detection when analyzed in a virtual environment, either manually, or in an automated fashion (e.g. by a sandbox). Many techniques rely on detecting differences in the environment (compared to when running in a real system), e.g. by checking installed programs, hardware and registry entries. Other techniques include abusing the maximum analysis time of sandboxes, unimplemented instructions, and absence of user interaction (in automated analysis systems).

Lundsgård and Nedström [14] showed that is is possible to bypass several of the most popular sandbox solutions used today by implementing test cases that performed checks to see what type of system was running the application. These test cases were written in C++ and mainly depended on calling the Windows API. Table A.1 contains a list of some of these evasive techniques, including commonly used techniques as listed by *Unprotect project* [24].

2.2 Hybrid-Analysis

Hybrid-Analysis provides an online automated analysis tool, capable of analyzing PE, Office, PDF, APK files and more. It was created by Payload Security and uses their VxStream Sandbox v6.50 in the backend.

2.3 Xen

Xen is an open-source type-1 hypervisor, supporting two types of virtualization: paravirtualization and hardware-assisted virtualization.

The guest domains (or virtual machines) are virtualized environments, each running their own operating system and applications. They are completely isolated from the hardware, i.e. they have no privilege to access hardware or I/O functionality, which is why they are sometimes called "Unprivileged domains", or "DomU". The first virtual machine (VM) started by Xen is the control domain (or Domain 0) which is the only VM that has privileges to access the hardware directly. It is responsible for interacting with the other virtual machines, and it exposes an interface through which the system is controlled.

The commands used for creating domains (*xl create* and *xl restore*), take a configuration file as a parameter (see Listing 1 for the configuration file used in this project). This configuration file makes it possible to configure among other things;

- Name of the domain,
- Amount of RAM,
- Number of virtual CPUs,
- Boot device (cd or drive),
- Network interface,
- Storage devices (hard-drives/CD-drives).

There are two main choices for local storage of guest images (i.e. their hard-drives): block devices (e.g. LVM-partitions), and image files (raw, qcow2, or vhd format). Section 2.3.1, provides an introduction to the QCOW2-format.

2.3.1 QCOW2

QCOW2 stands for "QEMU copy-on-write" and is a representation of a fixed-size block device in a file [16].

QCOW2 allows the creation of new Image files, with an already existing backing Image specified. To the user, the new Image file will look like it is a copy of the backing Image (without taking up much extra space), and any changes made to the copy will not be reflected in the backing Image, and deleting the copy does not affect the backing Image either. An illustration of this can be seen in Figure 2.1, where Image 1 and 2 are both based on the backing Image (base Image in the Figure), and used as bases for Images 1B and 2B, respectively. Modifying/removing Image 1, for instance, would not have any effect on the base Image or Image 2/2B, however, Image 1B would be corrupted as its backing Image was changed. Since these copies can also be used as storage device for virtual machines, it acts as a powerful tool when analyzing malicious files[1].

[1]By booting an Image with a backing file, any changes made to the Image will be discarded when deleting the Image

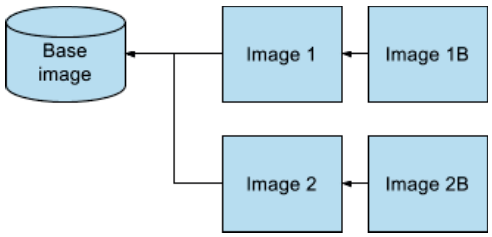

Figure 2.1: Illustration of QCOW2-Images created with a backing Image

2.4 DRAKVUF

According to Lengyel et al. [13]: "DRAKVUF is a virtualization-based agentless black-box binary analysis system. DRAKVUF allows for in-depth execution tracing of arbitrary binaries (including operating systems), all without having to install any special software within the virtual machine used for analysis".

DRAKVUF runs on Xen, on domain 0 in order to get access to hypervisor features. System calls are monitored by trapping internal kernel functions via #BP-Injection[2]. DRAKVUF contains no in-guest agents, and thus, in order to analyze samples, they have to be started manually. This is done by hijacking an arbitrary process within the VM and use it to initiate the start of the sample.

2.5 Machine learning

More data is created than ever before. According to IBM more than 2.5 quintillion bytes of data is created every day [8]. This, in combination with prices for storage space dropping rapidly, has led to an increase in the amount of data kept in storage. Instead of deleting unnecessary data, it is often stored and later searched through for meaningful information. This in turn has led to an increased usage of machine learning. By applying algorithms that are able to learn from data, to find patterns by themselves instead of having to write customized code for each problem, machine learning can be used to make predictions on the given data.

There exist several free software products for machine learning on the market today, where Scikit-learn [21] and Weka [6] are among the most popular ones. In these software, the algorithms are already implemented, and the user can instead focus on e.g. developing proper features to represent the data.

2.5.1 Feature engineering

One of the challenges in machine learning is determining the appropriate feature representation. This process is called *feature engineering* and involves the follow-

[2]A breakpoint instruction (INT3, instruction opcode 0xCC) is written into the VM's memory at the entry point of internal kernel functions.

ing parts [4];

- Feature selection - select the most relevant features to train on, among the existing features,
- Feature extraction - combining existing features to produce new and more useful features,
- Create new features by gathering new data.

Feature selection methods aid in this challenge by choosing features that will provide as good, or better accuracy[3], while requiring less data. They work by identifying and removing irrelevant features that do not contribute much to the accuracy, or may even decrease the accuracy of the model. Fewer attributes are also desirable since it reduces the complexity of the model, decreasing the required run-time. Or, in the words of Guyon and Elisseeff [5]:

> "The objective of variable selection is three-fold: improving the prediction performance of the predictors, providing faster and more cost-effective predictors, and providing a better understanding of the underlying process that generated the data."

The feature selection step can be performed in a variety of ways. One approach, implemented in Scikit-Learn, is called Randomized Logistic Regression, described as follows:

> "Randomized Logistic Regression works by subsampling the training data and fitting a L1-penalized LogisticRegression model where the penalty of a random subset of coefficients has been scaled. By performing this double randomization several times, the method assigns high scores to features that are repeatedly selected across randomizations. This is known as stability selection. In short, features selected more often are considered good features."

For this method to outperform standard statistical tests (such as removing features with low variance), the underlying model needs to be sparse, i.e. only a few of the features are relevant.

2.5.2 Testing and validating

The only way to know how well a machine learning model generalizes to new cases is by trying it out on new test cases. In order to perform such a test, the existing data needs to be split into two subsets, one *training set* and one *test set*, see Figure 2.2. The model can then be trained on the training set, and tested using the test set. The resulting error rate on the test set is called the *generalization error*.

By creating many models using different configurations, e.g. changing hyperparameters or feature representation, the generalization error can be used as

[3]When compared to the original features.

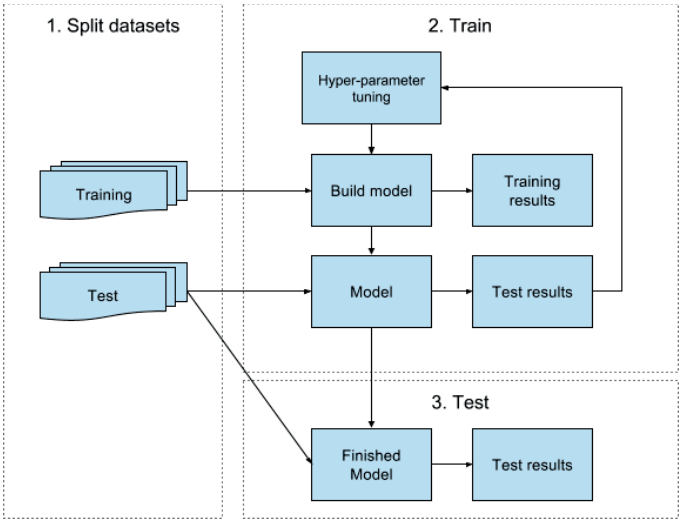

Figure 2.2: Machine learning process using train/test split

a means to find the best configuration. However, this approach suffers the risk of providing an optimistic generalization error. Since the generalization was used as a means to adapt the model, the resulting model is trained with the configuration best suited for the test set, and is thus unlikely to generalize equally well to new data.

In order to address this problem, a common approach used is to create a third set called the *validation set*. Now, the model is trained on the training set, and adapted to minimize the generalization error as measured on the validation set. Once a satisfactory result has been reached, the performance of the model is tested on the test set. See Figure 2.3 for an overview of this process.

An improvement to the previously mentioned approach, is to include a cross-validation step. Instead of validating the model performance based on a single run (using the same training and validation set), multiple iterations are performed using a different split each time, see Figure 2.4. This approach generally provides a more accurate performance measure, and is commonly referred to as *cross-validation*.

2.5.3 Random Forest Classifier

According to Raschka [22]: "The goal behind ensemble methods is to combine different classifiers into a meta-classifier that has a better generalization performance than each individual classifier alone.".

One of the most prominent machine learning algorithms is the random forest algorithm. Conceptually, the algorithm works by iteratively splitting the data into randomized subsets of train/test-data (known as bootstrapped samples), growing decision tree classifiers (see below) and giving a decision based on the

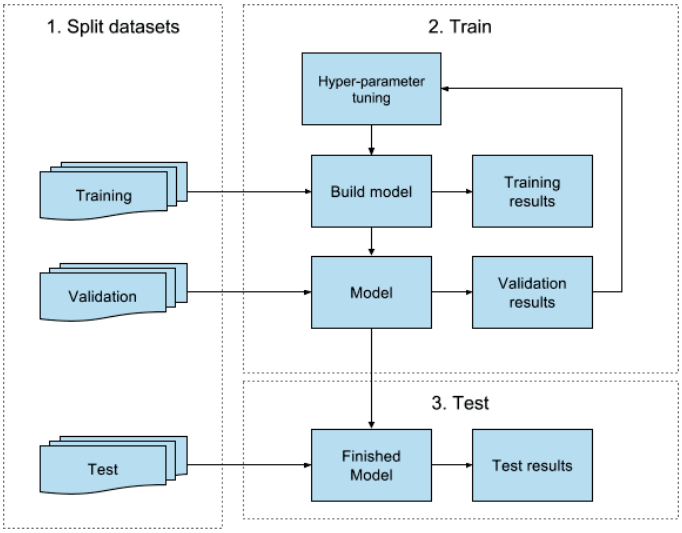

Figure 2.3: Machine learning process using train/test/validate split

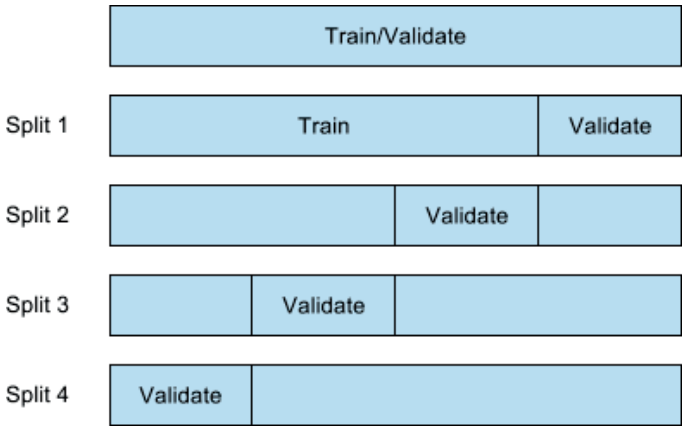

Figure 2.4: Train/Validate split in four iterations

result of the entire ensemble of trees. This tends to improve the predictive accuracy, compared to a model using a single decision tree classifier.

Decision Tree Classifier

Simply put, decision tree classifiers work in the following way:

1. Start with the entire data-set.
2. Select the attribute (feature) along dimension that gives the "best" split.
3. Create child nodes based on split.
4. Recurse on child nodes until one of the stopping criteria is reached.
    - All samples have the same class, or
    - the number of samples are too few, or
    - a maximum depth reached.

2.6 Windows Registry

In [17] the registry is described as follows: "The registry is a system-defined database in which applications and system components store and retrieve configuration data. The data stored in the registry varies according to the version of Microsoft Windows".

The entries in the registry are called keys and can contain other keys, or values. The registry is divided into six root keys, each containing information about the different parts of the system. Information available through the registry includes (but is not limited to) [25, page 281];

- Which applications should be run on startup,
- information about the running environment,
- connected devices,
- software settings.

This information is made available to applications run on the system. They can access it by e.g. using Windows API calls.

2.7 Windows API

The Windows API provides services used by all Windows-based applications. It exposes functionality to create files, write to files, read registry keys, etc. The API consists of thousands of functions, which are divided into the following major categories [25, pages 1,2].

- Base services
- Component services
- User interface services
- Graphics and Multimedia Services
- Messaging and Collaboration
- Networking
- Web Services

Some functions require the processor to be running in a special privileged mode called *Kernel mode*. When running in Kernel mode, the processor has access to all instructions and memory of the system. A reason for not giving this privilege to every application is that they would read and/or write to the same memory with a risk of overwriting information currently used by another application. This is accomplished using *system calls*. A system call causes the executing thread to transition into kernel mode and enter the *System Service Dispatcher*. The System Service Dispatcher then executes the requested function after which the thread returns to user mode [25, page 113].

2.8 Related work

Moser et al. [18] introduced a system for analyzing different execution paths of malware. By taking snapshots at different checks made by the analyzed malware and making changes to the system, the authors were able to analyze how the malware behaved differently depending on which path it took. Balzarotti et al. [2] use a technique of running the malware in two separate environments, one stealthy sandbox along with a reference environment. This technique was used as inspiration for manually labeling samples as evasive/non-evasive in this thesis.

In the area of bypassing sandboxes Lundsgård and Nedström [14] studied techniques used by evasive malware, and developed their own test cases that intended to evade sandboxes.

Chapter 3
Method

This chapter presents the method followed in this thesis. It is divided into the following parts;

Gathering samples Describes the process when gathering executable files.

Analyzing samples Provides a description of the environment created to perform dynamic analysis on executable files.

Machine learning Contains a thorough description of the process used to develop the final distinguisher using machine learning.

3.1 Gathering samples

A set of malicious samples were gathered from Hybrid-Analysis (see Section 2.2), along with 185 (non-malicious) samples collected from the executable files provided with a fresh installation of Windows 7[1]. In order to use these samples for classification (a supervised learning task), they need to be correctly labeled as evasive or non-evasive. For this labeling to be as accurate as possible, manual analysis would be preferable. However, manual analysis is a time demanding task, so the samples were first run in the analysis environment (see Section 3.2) in order to remove samples that failed to generate enough data to be utilized by the machine learning algorithm.

After running the samples, a total of 94 malicious samples remained to be analyzed manually. Since the samples to analyze had already been analyzed by Hybrid-Analysis, indicators of what actions the malware took could be found in their report. More indicators were also made available by running the samples in another sandbox, a version of Cuckoo that had been configured to be stealthy. With these indicators in mind, a framework was established.

The manual analysis was performed in a preconfigured virtual machine running Windows 10 that SecureLink provided in order to ensure the safety of the host system. To ensure that malware deploying evasive techniques does not expose its malicious behaviour, changes to the system were made in order to make it look as much as a virtual machine and/or sandbox as possible. The changes were made with techniques employed by Lundsgård and Nedström [14] in mind, see below.

- Changed the username to Malware
- Changed the computer name to Sandbox
- Added registry keys used by VirtualBox, Qemu, Sandboxie, Wine
- Changed the number of processors to 1
- Changed the amount of RAM to 1GB
- Created several instances of small processes renamed to look like processes run by virtual machines and sandboxes
- Added files commonly found in virtual machines and sandboxes
- Changed the screen resolution to 800x600px
- Run common analysis tools
- All samples were executed from the folder C:/Cuckoo/sample/sample.exe

[1]The samples were collected by a recursive search of the entire hard drive of the analysis environment for .exe-files. Only the samples that successfully executed in the analysis environment were included.

With these changes made, any file examined that did not exhibit malicious behavior, was considered evasive[2]. The indicators received when analyzing the samples using Hybrid-Analysis and the Cuckoo sandbox, were used as guidance as to what malicious behavior could be expected of each sample. The following tools were used during the analysis.

Process monitor Monitor API calls and file accesses.

RegShot Used to track changes made to the registry.

Wireshark Monitor network traffic.

API Monitor Monitor API calls.

In some cases the result was obvious. For example when analyzing one of the samples, it was previously assumed to be evasive due to the indicator "Checks Registry for VMWare specific artifacts" provided by Hybrid-Analysis. To verify this behavior, the file was launched, while monitored by process monitor. As expected, process monitor revealed a call checking the registry for a key, specific to Virtualbox. Shortly thereafter, the file terminated execution. Also, the analysis machine was searched for indications of infection, and as expected, they were nowhere to be found.

Other cases presented a much harder challenge. For example, one of the samples was showing clear indications of evasive behavior when analyzed in the sandboxes. However, when running on the analysis machine, a clear indication of malicious behavior could be noted, it registered itself for automatic run on system start. Oddly, it also performed the evasive checks flagged by Hybrid-Analysis. After more thorough inspection, it was shown that the malware did indeed exhibit evasive behavior. In case the evasive check failed (VM detected), it only registered itself for autostart. However, if it passed, it also started a service exhibiting malicious behavior. This was probably done in an attempt to avoid detection in a sandbox, where the action of registering for autostart might be considered to little of a threat on its own.

After labeling all samples, 24 of the samples were handpicked to be part of the test set, 12 evasive, and 12 non-evasive. The 12 evasive samples were chosen based on the evasive techniques used, in order to include several different evasive techniques, even such that are not present in the training data. These samples are presented in Table 3.1. The 12 non-evasive malware chosen for the test set were chosen based on their type, in order to include different types of malware. These samples are presented in Table 3.2.

[2]With reservation for files abusing stalling, as they will still exhibit their malicious behavior eventually.

Sample (MD5)	Evasive method
20ca...145e	Checks the registry for Wine, Virtualbox, Process Monitor, debuggers.
3873...b5f4	Delays execution a long time.
f09c...6c1a	Read registry for VMware; changing values in the registry results in the execution behaving differently.
f434...c13f	Stalls by using a very high number of calls to GetSystemProcess.
0cac...4cc0	Checks devices, registry keys used in VMware and Virtualbox.
752d...3991	Uses WMI queries to check the video controller.
fbf1...bd5b	Reads registry for known VM keys. Reads processes.
9030...02fd	Read registry for VMware, queries information about the hard drive.
75ef...64ca	Reads registry for known VM keys, Reads the Windows installation date.
b29a...1d38	Reads registry for known VM keys, Looks for analysis tools installed on the hard drive.
61b4...0688	Reads files and registry for known VMs.
c851...6d0f	Confirmed as evasive by analyst at SecureLink, techniques used unknown.

Table 3.1: Evasive samples included in the test set

Sample (MD5)	Description
20e9...1424	Adware
336b...6889	Adware
1e59...fc3c	Backdoor
6e9b...cc06	Trojan
5ab7...ee90	Keylogger
3c33...2156	Backdoor
9b2e...8ccd	Backdoor
7140...76d9	Network trojan
1c30...3318	Non-malicious
84c8...b549	Ransomware
246c...faf2	Backdoor
9c47...6f18	Ransomware

Table 3.2: Non-evasive samples included in the test set

3.2 Analysis environment

The goal when designing the analysing environment was to create an environment capable of automatically performing dynamic analysis on executable files, providing output suitable for use in a machine learning algorithm. To aid in this challenge, previous work in the field was examined as a source of inspiration. Few articles concerning the classification of evasive malware using machine learning were found, and those found focused on other methods than the one proposed in this thesis (e.g. analyzing differences in behavior between different analysis environments [2, 9]). However, vast amounts of research regarding the classification of malicious behavior using machine learning was found [3, 7, 10, 11, 20, 26, 30]. This was considered an equally good source of inspiration, as the only major difference between the two fields is the data used to train the model. Two previous approaches, achieving great results, focus on analyzing Opcodes [26], and System calls/API-calls [7, 11, 30]. SecureLink requested a sandbox consisting of open source components. Several options were considered.

PANDA Open source analysis platform for dynamically reverse engineering applications, that allows replays of the program execution and is extensible through lots of plugins.

Cuckoo Open source malware analysis system. Highly customizable.

DRAKVUF An agentless malware analysis system preconfigured to be stealthy. Hooks system calls by putting a breakpoint instruction (0xcc) at the beginning of internal kernel functions.

PANDA was first considered as the top choice, but after trying it out it was found that during its ongoing migration from its first version to its second, a plugin for monitoring system calls did not exist yet. PANDA was hence discarded as an option[3]. Both Cuckoo and DRAKVUF were viable options. However, due to recommendation from SecureLink, DRAKVUF was chosen for its stealthy properties.

3.2.1 Setting up the environment

Before installing DRAKVUF, a virtual machine running Ubuntu 16.04 was set up on VMWare. The reason for hosting the analysis machine on VMWare was to utilize VMWares snapshot functionality[4], and to ensure no malware could infect the company network. DRAKVUF was then installed on the virtual machine following the installation instructions on their website, creating the analysis domain as follows (An example of DRAKVUF in action can be seen in Figure 3.1).

[3]However, a plugin for monitoring system calls was available at the time of writing.

[4]DRAKVUF's installation process proved to be quite error-prone, taking snapshots at regular intervals provided a way to go back in case some step went wrong.

Setup process

1. A QCOW2-image (see Section 2.3.1) with 20GB of storage was created (base image).
2. A new domain (DomU) was created, with the QCOW2-image mounted as storage, and a Windows 7 image mounted to a CD-drive.
3. Windows 7 was installed on the DomU.
4. With the DomU booted, the following was changed;
   - Windows Firewall was turned off to avoid interference with analysis,
   - Network was configured to use the IP-address of Dom0 (actually the virtual interface in Dom0 connected to the DomU) as standard gateway, in order to be able to download samples from Dom0 .
5. The state of the domain was saved to a checkpoint-file using Xens built-in functionality (see Section 2.3), and the domain was shut down.

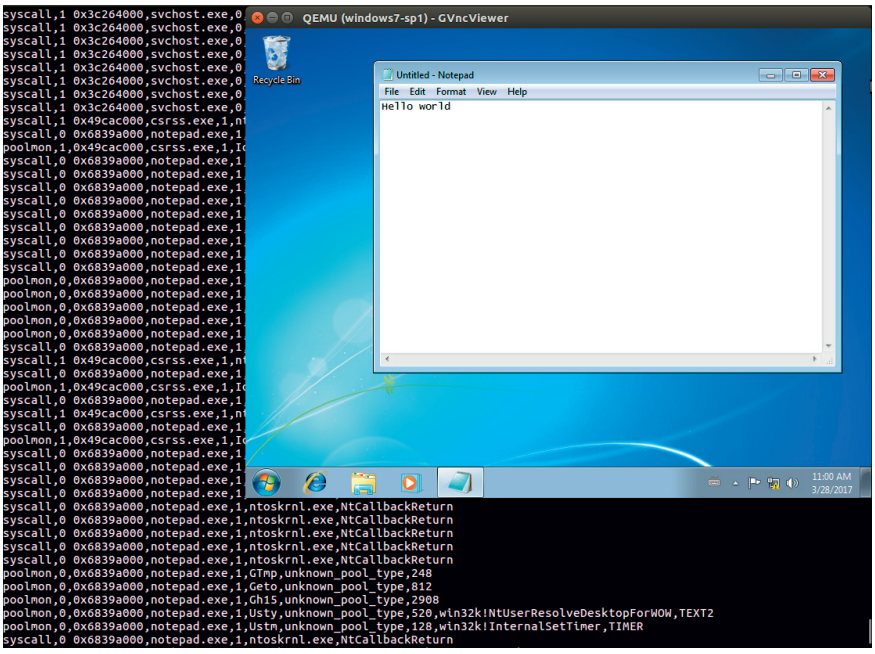

Figure 3.1: Example output from DRAKVUF while running notepad in analysis environment.

3.2.2 Analysis process

When the analysis environment was set up, all samples were analyzed as described below and as illustrated in Figure 3.2. The maximum time for analysis of each sample was limited to 90 seconds.

Analysis process

1. A snapshot of the base image is created.
2. A new domain is created by restoring the previously saved state, using the snapshot as storage.
3. DRAKVUF script is executed in domain 0, injecting and executing the following script in the analysis domain.

```
# Use powershell to fetch malware sample
# from Malware server
powershell (new-object System.Net.WebClient)
    .Downloadfile(
        'http://$MALSERV_IP/$MALWARE_NAME',
        'C:\Users\\\$USER\Desktop\mw.exe'
    )

# Execute the fetched malware

start C:\Users\$USER\Desktop\mw.exe
```

4. When executed, the script fetches a malware sample from the malware server hosted on domain 0 and executes the malware sample.
5. While the malware is running, each time an internal kernel function is called, DRAKVUF (domain 0) gains control of execution, and writes the system call performed to a log file, then returns control to the analysis domain.
6. When a time limit has been reached, DRAKVUF exits, regardless of if the malware sample has finished executing.
7. The analysis domain is destroyed and the snapshot created in step 1 is deleted.

3.2.3 Automatization and processing of logs

In order to avoid having to start the analysis of each sample manually, an automated analysis environment had to be created. In order to minimize development time, this was done in a very simple way, resulting in a system consisting of only three components.

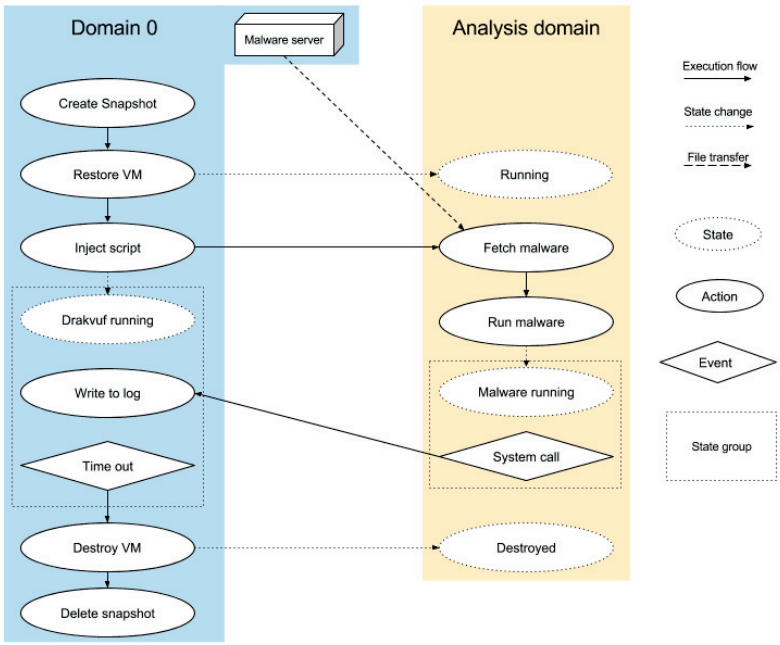

Figure 3.2: Dynamic analysis process

Folder watch script A script watching a folder for added files. Each time a new file was added, it was added in a queue of files to be analyzed. Implemented using the *inotifywait*-command included in the *inotify-tools* package part of the standard Ubuntu package repository.

Analysis script A script taking a filename as argument. When executed, starts analysis of the file according to the process presented in Section 3.2.2.

Http-Server A minimalistic http-Server used to serve files in a folder. Implemented using the http-server package provided by the node package manager (npm).

When the analysis script is started, it moves the file to be analyzed to the folder hosted by the http-server. It then performs the analysis piping the output to a file. In order to only keep the relevant logs produced by DRAKVUF, this file is processed using a text analysis tool. Any row not related to a system call produced by the analyzed file (actually, by the process it spawns), will be removed. The remaining rows are then written to the final log file.

3.3 Machine learning

In order to train a machine learning algorithm on the output from the analysis system, it needs to be transformed into the chosen n-dimensional feature space. Two different feature representations were considered as baseline. The first approach, inspired by Hansen/Larsen [7], contains features representing the number of occurrences of system calls. The second approach, aims at capturing sequential information possibly missed by the first approach. The name of the first 1000 system calls are used as features (encoded as integers). The logs from the analysis system were transformed to features according to the following procedures:

Representation 1 (number of system calls)

1. All logs in the training/test-set are parsed, recording the name of all system calls made in a set.
2. A dictionary is created, mapping names of system calls to an integer in the range [0,<Number of System Calls> -1].
3. For each log, the number of system calls is counted, divided by the total number of system calls in the log (to normalize), and added to index i (corresponding to the integer assigned to the system call in the dictionary) in the logs corresponding feature representation (vector).

Representation 2 (sequence of system calls)

1. All logs in the training/test-set are parsed, recording the name of all system calls made in a set.
2. A dictionary is created, mapping names of system calls to an integer in the range [0, <Number of System Calls -1>].
3. For each log, feature number i equals the system call made at position i in the log, translated to an integer using the dictionary from step 2. System calls not present in the dictionary are recorded as 0, and if fewer than N system calls are made, the feature representation is padded with zeros.

Using both these feature representations, two machine learning models were created. These models were tested using 100-fold cross-validation with a train/test-split of 70% training data and 30% test data, and a *RandomForestClassifier*. The test results can be seen in Table 3.3. As can be seen, representation 1 resulted in a significantly higher performance than representation 2. Representation 1 was chosen as the baseline representation, which will henceforth be referred to as the original features.

In order to improve the performance, the next step was to implement feature

	Representation 1	Representation 2
#Features	151	1000
Recall	0,51±0,28	0,14±0,18
Precision	0,67±0,31	0,30±0,39
Accuracy	0,83±0,08	0,75±0,07
f1-score	0,58±0,29	0,19±0,25

Table 3.3: Result of 100-fold cross validation on baseline models

engineering. The feature engineering was divided into feature extraction (i.e. constructing new features from the original features), and feature selection (i.e. searching all features for the subset of features providing the best performance).

In a survey by Sondhi [28], different feature extraction (feature construction in their survey) methods were presented. However, the methods presented were considered too time consuming to implement. Instead, a very straightforward technique based on brute-force was implemented;

Feature extraction

1. Consider all pairs of features x, y.
2. For each pair, create a new feature representing the ratio $\frac{x}{y}$.

In the proposed algorithm, the feature space is limited to the combination of feature pairs using the division operator. This resulted in a new feature set consisting of 23409 features. More and possibly better features could be constructed considering more operators, or n-grams ($n > 2$) of features. However, the feature space grows exponentially in the size of the n-grams, and choosing an $n > 2$ resulted in a feature space too large for feature selection to perform desirably.

Before deciding which method to use for feature selection, a list of desirable properties of the method was established. First, in order to reduce the risk of programming errors and bugs, the method should be easy to implement in Scikit-Learn. Also, it should be stable in the sense that it tends to select the same features if run multiple times. Finally, it should be able to produce reasonable results, even when the input is a very large set of features. In order to fulfill the first property, two methods implemented in Scikit-Learn were considered, *SelectFromModel* and *RandomizedLogisticRegression* (RLR). Apart from these, a naïve method, relying only on the correlation between features and the class label as scoring was considered as well. A brief introduction to each selection methods follows below.

RandomizedLogisticRegression Randomly sample the training data and fit a L1-penalized LogisticRegression model, where the penalty of a random subset of coefficients have been scaled. This randomization is performed several times, and the features selected more often (By the LogisticRegression model) are chosen as the final features.

Model	f1-score
RandomizedLogisticRegression	0.711
SelectFromModel (Using RandomForest)	0.763

Table 3.4: Results after feature selection

SelectFromModel A model is trained on the training data. Then, the features considered most important (implementation of importance score varies between models) are chosen.

Naïve Correlation For each feature, calculate the correlation to the class labeling. Keep only the features with a correlation over a given tolerance.

While the naïve correlation approach results in a stable selection (it always chooses the same features), it showed other weaknesses. Consider the case where two features each show little correlation to the class label. Together they give enough information to distinguish between the classes. In such a case, the model would drop both features, risking a performance drop.

The remaining two models were both considered viable, and were tested in order to see which one provided the best performance.

In order to get optimal results from both models, two methods for hyperparameter tuning were considered, *grid search* and *randomized search*. While grid search performs an exhaustive search over a parameter space, this requires domain knowledge in order to choose the appropriate parameter space, as exploring a too large space is very time consuming. Also, grid search can easily miss out on an optimal solution unless the parameter space is chosen with very high granularity. Resorting to randomized search instead, allows exploring a larger parameter space without consuming too much time, since the number of iterations are predefined (however choosing too few iterations will obviously lead to the risk of choosing suboptimal parameters). With this in consideration, randomized search was chosen as the desired method.

A randomized search was performed, using 100 iterations, optimizing the hyperparameters of the selection models, in combination with a RandomForestClassifier. The results from the optimization can be seen in Table 3.4.

To prepare the final classifier, feature selection was performed using SelectFromModel with the hyperparameters configured according to the results from the optimization. The model was then trained on all samples in the training set. Finally, the model was used to predict the labels (and the certainty of the labeling) of the samples in the test set. An overview of the final model and the results when labeling the test set can be seen in Chapter 4.

Chapter 4

Test results

This chapter presents the machine learning model developed as part of the implementation process (see Chapter 3). Also, the results when using this machine learning model to label executable samples is presented.

The model developed in this thesis is a *RandomForestClassifier* configured as presented in Table 4.1. Any hyperparameter not presented uses the default value assigned by Scikit-Learn. It is trained on a set of 19 evasive samples and 58 non-evasive samples. The input to the classifier (feature vectors) contains the number of system calls of each type, made by each sample (151 features in total). The input data is transformed by first extracting new features, considering each combination of features x,y, creating a new feature with the value x/y (e.g. if feature x and y represent the number of calls to NtClose and NtOpen, respectively. The new feature represents the number of calls to NtClose divided by the number of calls to NtOpen). Then, the most important features are selected, using *SelectFromModel* in combination with a *RandomForestClassifier*, configured according to Table 4.1 (the RandomForestClassifier used by SelectFromModel was configured in the same way as the one used for classification in the final model).

Parameter	**Value**
SelectFromModel	
Classifier	RandomForestClassifier
RandomForestClassifier	
Number of estimators	20
Min samples leaf	1
Min samples split	3

Table 4.1: Configuration of parameters for machine learning models

After performing feature selection using SelectFromModel, the resulting model consisted of a total of 94 features. A subset of the features selected can be seen in Table 4.2, only the 10 features deemed most important by the selection algorithm are shown.

Feature	**Importance**
DelayExecution/WaitForWorkViaWorkerFactory	0.11833
DelayExecution/ReleaseKeyedEvent	0.10613
CreateSection/DelayExecution	0.070176
DelayExecution/ResumeThread	0.06277
SetInformationFile/WaitForMultipleObjects32	0.05338
QueryTimerResolution/TraceControl	0.05186
Close/QueryDirectoryFile	0.04418
DelayExecution/QueryDebugFilterState	0.03740
ReadFile/EnumerateValueKey	0.03693
DelayExecution/Close	0.029237

Table 4.2: Chosen features, and their importance assigned by SelectFromModel using RandomForestClassifier

Also, the correlation between features and the class label was measured, and the 10 features showing most correlation to the class label are shown in Table 4.3.

Feature	**Correlation**
CreateSection/DelayExecution	-0.632915
QuerySymbolicLinkObject/DelayExecution	-0.632481
AllocateVirtualMemory/YieldExecution	-0.455466
ProtectVirtualMemory/AlpcDeleteSecurityContext	-0.376022
QuerySystemInformation/WriteVirtualMemory	-0.342793
QueryTimerResolution/TraceControl	0.509237
DelayExecution/AllocateLocallyUniqueId	0.589966
DelayExecution/QueryDebugFilterState	0.600701
QueryTimerResolution/ClearEvent	0.627196
DelayExecution/LockFile	0.636480

Table 4.3: Correlation between features and Evasive, only showing top/bottom 5

The generalization error of the model was measured, by using it to predict the label of 24 samples (12 of each class). The results of this labeling can be seen in Table 4.4, where a class label of 1 or 0, indicates a sample known to be evasive or non-evasive respectively. The last column represents the probability of being evasive, as assigned by the model, where a probability >0.5 results in the sample being labeled as evasive. Incorrectly labeled samples are highlighted in the table.

Hashsum (md5)	Class	Evasive
0cac...4cc0	1	0.45
1c30...3318	0	0.0
1e59...fc3c	0	0.0
20ca...145e	1	0.9
20e9...1424	0	0.0
246c...faf2	0	0.225
336b...6889	0	0.0
3873...b5f4	1	1.0
3c33...2156	0	0.3625
5ab7...ee90	0	0.7
61b4...0688	1	0.05
6e9b...cc06	0	0.0
7140...76d9	0	0.5
752d...3991	1	0.85
75ef...64ca	1	0.3825
84c8...b549	0	0.5833
9030...02fd	1	0.65
9b2e...8ccd	0	0.475
9c47...6f18	0	0.725
b29a...1d38	1	0.55
c851...6d0f	1	0.8
f09c...6c1a	1	0.7
f434...c13f	1	0.7
fbf1...bd5b	1	0.65

Table 4.4: List of actual class of samples (1=evasive) and the probability of being evasive assigned to them by the distinguisher.

Chapter 5
Discussion

In this chapter:

Analysis of results Presents an analysis of the results.

What do features actually represent? Provides an analysis of the resulting feature representation.

Evaluation of the analysis environment Contains an evaluation of the constructed analysis environment.

5.1 Analysis of results

Looking at the test set, three samples were incorrectly labeled as evasive, and two were incorrectly labeled as non-evasive. See Table 4.4 for the full result when running the test set in the distinguisher. More thorough manual analysis was performed in order to see why these samples failed to be labeled correctly. Two of the non-evasive samples were fairly similar. They were both variations of ransomware that during the stage of gathering samples had been seen encrypting files on the virtual machine used. Upon closer inspection it was seen that it spawned several processes that handled the encryption, while the original process delayed execution for a long time. By going back to the original output logs from DRAKVUF, it was seen that the number of calls to NtDelayExecution was high. This, in combination with a low number of system calls represented by the denominators in the features with high correlation to evasion led to the conclusion that it had been taken for a malware utilizing stalling methods. For the third false positive, it was more difficult to find out what had gone wrong. The system call ratios did not stand out when compared to the features with the highest/lowest correlation to evasion. Running the malware again showed that it created different windows. An assumption was made that the training set contained more evasive than non-evasive samples that created windows. Comparing the output from the sandboxes once again, this assumption was seen to be true. Hence it can also be concluded that the distinguisher besides encapsulating evasive behaviour, is also biased towards labeling samples creating windows as evasive.

It can also be seen that two of the samples were falsely labeled as non-evasive. This is the more dangerous case. False positives can be sent to an analyst to be manually analyzed before a final verdict is given. In an automated system, false negatives would be missed and could end up infecting more systems. Looking at the two false negatives, some similarities between the two could be seen. When run in the stealthy Cuckoo sandbox, it showed that it looked for VM-specific registry keys and files, and when it did not find them it modified several files and installed itself for autostart. When run in the badly configured Windows system, it did some of the checks but did not modify any files or install itself for autostart. Obviously the sample was displaying evasive behaviour. Upon looking at the output log it could be seen that the file size was small. Most of the log constituted of system calls related to the startup routine. From this it could be deduced that due to the quick check-and-exit by the malware, it was able to escape detection.

A major concern, raising questions regarding the validity of the results, is the limited data set used in this thesis. While gathering samples it turned out that using manual analysis to determine whether evasive techniques were used or not was a bigger problem than expected. A lot of samples were discarded after the result of manual analysis turned out to be inconclusive. This does of course raise questions regarding the technique used to gather samples. Surely the demands on accuracy when labeling samples could be lowered, resulting in a larger sample set, for example by labeling data completely based on the output of already exist-

ing sandbox-solutions. This method was tested at first, however, it was discarded due to lack of proper output. The output from the sandboxes tested did not include enough information to accurately label samples, which was revealed when validating some of the labeled samples using manual analysis. Thus, a decision had to be made, more samples, or a more accurate labeling. It was decided that a properly labeled sample set was desirable, as it would surely result in a better model. Thus, manual analysis was chosen as the preferred method for labeling, resulting in a smaller sample set.

5.2 What do the features actually represent?

By looking at the top features that were present in both the list of highest correlation to evasive, along with the highest feature importance, some conclusions can be drawn. Most noticeable is the frequent occurrence of NtDelayExecution/X, which was listed three times in the top five of each of the lists. This is not a surprise due to the amount of evasive malware gathered that utilized stalling techniques which often depend on calls to NtDelayExecution.

Looking at the other end of the list which shows negative correlation to evasive, it can be seen that NtCreateSection/NtDelayExecution is located at the top spot. This feature is also present at the top tier of features ranked high in importance. The presence of NtCreateSection is surprising. Evasive techniques relying on enumerating active processes often do so by first calling CreateToolhelp32-Snapshot and then iterating through the active processes. CreateToolhelp32Snapshot in turn calls NtCreateSection. Since this feature has high negative correlation to evasion, there must be some other reason why it is featured here. One reason could be that CreateProcess also calls NtCreateSection. Evasive samples would have a tendency to shut down before spawning new processes, while non-evasive malware would not.

5.3 Evaluation of the analysis environment

In hindsight, the choice of using DRAKVUF as basis for the analysis environment was probably not the best one. The installation process turned out to be error-prone, up to the point where it was decided to host the analysis environment on VMWare, in order to be able to revert to previous snapshots made in case some step in the installation process went wrong. DRAKVUF also lacks automated functionality out-of the box, requiring a custom implementation of features required, such as the ability to queue multiple samples for analysis, and writing output to an appropriate location. To summarize, setting up the analysis environment was a very time-consuming process, which could have been avoided by choosing a more full-scale solution, such as Cuckoo, as the basis for the analysis environment.

When setting up the environment, it was decided to prevent any network

traffic between the environment and the outside environment. This was decided in order to avoid the risk of exposing the company network to potential malware escaping the analysis environment. However, this was probably a bit overcautious as the network traffic was configured to be sent through a VPN specifically used when analyzing malicious files. This may have affected the results negatively, as some malware rely on network traffic to work, resulting e.g. in a stalling loop, awaiting an available connection. This could easily be mistaken for evasive behavior, but should not necessarily be labeled as such.

Another issue that arose was the fact that DRAKVUF was still in beta stage at the time of writing. This can be seen in its lack of support for e.g. logging arguments passed to system calls, and attaching timestamps to logs. Support for such features would allow more complex features to be constructed, possibly resulting in better prediction performance. For example, with the current implementation, reading any key from the registry will result in the same log output, even though reading some registry keys (specifically those used in evasive behavior) should be considered more indicative of evasive behavior. Also, the addition of timestamps would provide a means for the predictor to notice stalling loops performed without system calls.

When analyzing samples with DRAKVUF, the output consisted of every system call made in the system, including lots of unwanted information related to unrelated processes. The relevant system calls were extracted using text analysis tools, including only the logs containing the name of the sample analyzed as the calling process. Due to this, only logs related to the main process were gathered. It is not uncommon for malware to contain more than one process. For example, in downloaders, the main process is simply responsible for downloading a malicious payload, executing it in the infected environment. This clearly leads to complications when trying to model the behavior of malware, as a lot of information is missing. Attempts were made to come up with a way of logging all system calls made as a result of a malware executing. However, lack of support for this in DRAKVUF would require this to be implemented manually using only the available logs, and the problem was deemed too time-consuming to deal with. However, it is likely that evasive malware perform their environment checks in the main process, in order to avoid revealing the behavior of any additional processes (or services).

Apart from logging system calls, DRAKVUF also provides plugins, logging other events. Most notably, information regarding files accessed and files deleted from the system (including the file in binary format) can be provided. By utilizing this information, malware using file checks to implement evasive behavior could easily be spotted, provided a similar evasive sample is present in the training set. Also, the access to deleted files could help when analyzing malware that use temporary files to store information and/or execute code. However, this was excluded in order to limit the complexity of the machine learning model.

Something that should probably have been taken more into consideration when designing the analysis system is the desired behavior of evasive malware. By configuring the environment in order to avoid as many known detection tech-

niques as possible, any malware utilizing these detection techniques would launch its malicious behaviour as if run on a regular computer. This would allow a more complete view of the evasive samples. Specifically, evasive malware utilizing multiple evasive techniques would expose all of them to the system, providing more useful information to the machine learning algorithm.

Chapter 6

Conclusions & Future work

In this chapter, conclusions are drawn based on the results presented in Chapter 4 and knowledge gained during the course of this project. Also, in Section 6.1, suggestions for future work in the area is presented.

Even though the test set was small, the results of the labeling can be used to draw conclusions regarding the model's accuracy based on statistics. First, assume the model behaves like a binomial distribution, i.e. predictions are statistically independent and have an equal probability of success. Then, a confidence interval for the probability of this distribution can be used as a confidence interval for the model's predictive accuracy as well. Applying this logic to the model's accuracy using a 95% confidence interval result in an accuracy in the range [0.5329, 0.9023]. This means the distinguisher can be said to have an accuracy over 53% statistical significance*. Unfortunately, performing the same calculations for the recall (or precision), results in the interval [0.4281, 0.9451], overlapping 0.5. It can thus not be said with statistical significance* that the model has a recall over 0.5, i.e. it cannot be shown that it will identify more evasive samples than simply guessing would. However, it should be noted that this is just an analysis regarding what can be said about the results with statistical significance. The model still showed promise, with a 75% accuracy, recall, and precision.

With time being an issue, the analysis environment should preferably be implemented using a full-scale solution such as Cuckoo, in order to avoid having to manually implement some required functionality such as automatic processing of executable files. However, in order to get a system as accurate as possible, DRAKVUF would be the better choice as it allows for a more stealthy analysis system, due to the fact that it does not introduce any artifacts (e.g. monitoring programs, common registry keys, etc.) to the analysis machine. Thus, such a system would result in a smaller risk of being detected by the analyzed malware.

Something to take into consideration when preparing the output data from the analysis system, is the sequence of system calls commonly occurring at the startup of a process. The false negatives in the test set were most likely labeled erroneously as they only required a few system calls for the evasive behavior, occurring early in the execution process. This in turn caused the resulting number of system calls to reflect the behavior of the process startup rather than the behavior of the entire process. This could be solved by identifying and removing the beginning of the log (reflecting the process startup), possibly increasing the model's performance.

Some evasive techniques turned out to have a bigger impact on the feature representation. This mainly concerns any techniques involving stalling, due to the fact that a large amount of system calls are generally utilized when implementing such behavior.

One of the samples in the test set, labeled correctly as evasive by the developed distinguisher, got a non-malicious threat-score from Cuckoo. This indicates that the model could be used to improve the detection rate of existing sandboxes.

In this thesis, it has been shown that it is possible to detect evasive behavior in malware using dynamic analysis and machine learning. However, tests on larger sample sets would be needed to verify the system's scalability, in order for it to be used in a real sandbox-solution.

6.1 Future work

A first step for improving the proposed solution would be an improved sample set, utilizing samples labeled by professional malware analysts, in order to get as much relevant information as possible to start with. Given such a set, not only would the results be easier to verify, it would also allow more complex machine learning models to be used, most notably Deep Learning (this would of course require a significantly larger sample set).

Another interesting improvement would be to upgrade the analysis environment, following the recommendations from Section 5.3. This, in combination with an improved feature set utilizing the increased amount of information available, would allow for a significant performance boost.

Last, a model only capable of distinguishing between evasive and non-evasive behavior is quite unremarkable by itself. However, by implementing a sandbox, using this distinguisher as part of a pipelined solution, would be an interesting area of investigation.

Appendices

Appendix A

Extra tables

Evasive technique	Description
Check for memory artifacts	Search for strings in memory containing VMware, vbox, etc.
Sleep for a long time	Delays execution to prevent being run in a sandbox.
Scan running processes	Look for processes common in virtual environments, or the lack of processes normally found in a regular environment.
Check user name and/or computer name	Look for names such as "Sandbox" or "Malware".
Look for registry entries	Virtual machines often have registry keys specific for them.
Check the screen resolution	Regular users normally run their machine in a higher resolution than a sandbox.
Check the execution path	Running the malware as e.g. C:/malware.exe is suspicious.
Check the number of cores used	These days, running on a single-core system is not common.
Check the amount of RAM	A very small RAM size could imply it is running in a sandbox.
Check the size of the hard drive	A very small hard drive size is uncommmon in a normal workstation.
Check the MAC address	Common virtual machines often use MAC addresses starting the same way.
Look at browsing history	Regular usage would show traces in browsing history.
Check its own filename	Abort if longer than a certain length (hashsum) or if matching names like sample.exe.
Check connected devices	The devices connected to a sandbox may differ from a regular user.

Table A.1: Evasive techniques

```
arch = 'x86_64'
name = "windows7-sp1"
seclabel='drakvuf:vm_r:drakvuf_domU_t'
maxmem = 3000
memory = 3000
vcpus = 4
maxcpus = 4
builder = "hvm"
boot = "cd"
hap = 1
acpi = 1
on_poweroff = "destroy"
on_reboot = "destroy"
on_crash = "destroy"
vnc=1
vnclisten="0.0.0.0"
usb = 1
usbdevice = "tablet"
altp2mhvm = 1
shadow_memory = 32
audio=1
soundhw='hda'
vif = [
'type=ioemu,model=e1000,bridge=xenbr0,mac=00:06:5B:BA:7C:01'
]
disk = [
'tap:qcow2:~/snapshot.qcow2,hda,w',
'tap:tapdisk:aio:~/Windows7-64.iso,hdc:cdrom,r'
]
```

Listing 1: Xen config file used for analysis domain

Appendix B

Glossary

Accuracy $\frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives}$

API Monitor Tool for monitoring API calls made by applications. Shows a call stack of native function calls made by its parent Windows API call.

F1-score $\frac{2*(True\ Positives)}{2*(True\ Positives) + False\ Positives + False\ Negatives}$

Full virtualization Virtualization mode of an OS hosted on a hypervisor. The guest OS is unaware that it is in a virtualized environment, and operates directly on (virtualized) hardware, provided by the host.

Hardware assisted virtualization A type of Full Virtualization, where the processor has special instructions to aid the virtualization of hardware. This allows the guest OS to execute privileged instructions directly on the processor, without affecting the host.

Overfitting The process of fitting a model too much to the training set, ex. by using a too complex model. Will not generalize well to new data.

Paravirtualization Virtualization mode of an OS hosted on a hypervisor. Hardware interaction for the guest OS is done through calls to the hypervisor.

Precision $\frac{True\ Positives}{True\ Positives + False\ Positives}$

Process Monitor Allows the monitoring of processes, registry, and files.

Recall $\frac{True\ Positives}{True\ Positives + False\ Negatives}$

RegShot Used to observe changes made to the Windows registry, by comparing two snapshots taken at separate occasions.

Sandbox Automated solution for malware analysis. Usually run inside a virtual environment.

Scikit-learn Python library providing implementations of machine learning algorithms and utility functions.

Statistical significance* Means that a result from testing or experimenting is not likely to occur randomly or by chance. The * indicates a threshold of 5%, i.e. less than 5% probability that the result occurred by chance.

Supervised learning By labeling the training data, the task is to teach the machine learning model to predict the label of new input data.

System Service Dispatcher An internal dispatch table, used by the Windows kernel to translate system calls to the correct kernel function call.

Type-1 hypervisor Hypervisor running directly on the hardware.

Type-2 hypervisor Installs on top of the host operating system.

Underfitting The process of fitting a model too loosely on the training data, ex. by choosing a too simple model. Yields poor accuracy on new data.

Unsupervised learning All training data is unlabeled, and the model tries to learn the structure of the data.

Virtual Machine Emulated computer system.

Wireshark Used to analyze network traffic.

Zero-day threat A threat that exploits an undisclosed computer security vulnerability (it has been known for zero days).

Bibliography

[1] Avast. *Malware*. URL: https://www.avast.com/c-malware (visited on 05/02/2017).

[2] Davide Balzarotti et al. "Efficient Detection of Split Personalities in Malware." In: *NDSS*. The Internet Society, 2010. URL: http://dblp.uni-trier.de/db/conf/ndss/ndss2010.html#BalzarottiCKKKV10 (visited on 05/31/2017).

[3] George E. Dahl et al. "Large-scale malware classification using random projections and neural networks." In: *ICASSP*. IEEE, 2013, pp. 3422–3426. URL: http://dblp.uni-trier.de/db/conf/icassp/icassp2013.html#DahlSDY13 (visited on 05/31/2017).

[4] Aurélien Géron. *Hands on Machine Learning with scikit-learn and Tensorflow*. O'Reilly Media, 2017. ISBN: 9781491962299.

[5] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection". In: *J. Mach. Learn. Res.* 3 (Mar. 2003), pp. 1157–1182. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=944919.944968 (visited on 05/31/2017).

[6] Mark Hall et al. "The WEKA Data Mining Software: An Update". In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278. URL: http://doi.acm.org/10.1145/1656274.1656278 (visited on 05/31/2017).

[7] Steven Strandlund Hansen and Thor Mark Tampus Larsen. "Dynamic Malware Analysis: Detection and Family Classification using Machine Learning". MA thesis. Aalborg University, 2015. URL: http://projekter.aau.dk/projekter/en/studentthesis/dynamic-malware-analysis-detection-and-family-classification-using-machine-learning(11077420-bf4d-4362-80af-81b2d2f5c1b6).html (visited on 05/31/2017).

[8] IBM. *What is big data?* URL: https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html (visited on 05/18/2017).

[9] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "Barecloud: Bare-metal Analysis-based Evasive Malware Detection". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 287–301. ISBN: 978-1-931971-15-7. URL: http://dl.acm.org/citation.cfm?id=2671225.2671244 (visited on 05/31/2017).

[10] Bojan Kolosnjaji et al. "Adaptive Semantics-Aware Malware Classification". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*. DIMVA 2016. San Sebastián, Spain: Springer-Verlag New York, Inc., 2016, pp. 419–439. ISBN: 978-3-319-40666-4. DOI: 10.1007/978-3-319-40667-1_21. URL: http://dx.doi.org/10.1007/978-3-319-40667-1_21 (visited on 05/31/2017).

[11] Bojan Kolosnjaji et al. "Deep Learning for Classification of Malware System Call Sequences". In: *29th Australasian Joint Conference on Artificial Intelligence (AI)*. Dec. 2016. URL: https://www.sec.in.tum.de/assets/Uploads/deeplearning.pdf (visited on 05/31/2017).

[12] Lastline. *The First Choice in Advanced Malware Protection*. URL: https://go.lastline.com/rs/373-AVL-445/images/Lastline_Enterprise_WP.pdf) (visited on 04/13/2017).

[13] Tamas K. Lengyel et al. "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACSAC '14. New Orleans, Louisiana, USA: ACM, 2014, pp. 386–395. ISBN: 978-1-4503-3005-3. DOI: 10.1145/2664243.2664252. URL: http://doi.acm.org/10.1145/2664243.2664252 (visited on 05/31/2017).

[14] Gustav Lundsgård and Victor Nedström. "Bypassing modern sandbox technologies". MA thesis. Lund University, 2016. URL: https://lup.lub.lu.se/student-papers/search/publication/8880576 (visited on 05/31/2017).

[15] Malwarebytes. *State of Malware Report*. 2017. URL: https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf (visited on 05/02/2017).

[16] Mark McLoughlin. *The QCOW2 Image Format*. URL: https://people.gnome.org/~markmc/qcow-image-format.html (visited on 05/15/2017).

[17] Microsoft. *Microsoft Developer Network*. URL: https://msdn.microsoft.com/ (visited on 04/13/2017).

[18] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring Multiple Execution Paths for Malware Analysis". In: *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. SP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 231–245. ISBN: 0-7695-2848-1. DOI: 10.1109/SP.2007.17. URL: http://dx.doi.org/10.1109/SP.2007.17 (visited on 05/31/2017).

[19] Cisco Security Research & Operations. *What Is the Difference: Viruses, Worms, Trojans, and Bots?* URL: http://www.cisco.com/c/en/us/about/security-center/virus-differences.html (visited on 05/02/2017).

[20] Razvan Pascanu et al. "Malware classification with recurrent networks." In: *ICASSP*. IEEE, 2015, pp. 1916–1920. ISBN: 978-1-4673-6997-8. URL: http://dblp.uni-trier.de/db/conf/icassp/icassp2015.html#PascanuSSMT15 (visited on 05/31/2017).

[21] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2078195 (visited on 05/31/2017).

[22] Sebastian Raschka. *Python Machine Learning*. Birmingham, UK: Packt Publishing, 2015. ISBN: 1783555130.

[23] Symantec Security Response. *VENOM vulnerability could expose virtual machines on unpatched host systems*. 2015. URL: https://www.symantec.com/connect/blogs/venom-vulnerability-could-expose-virtual-machines-unpatched-host-systems (visited on 04/24/2017).

[24] Thomas Roccia. *The First Choice in Advanced Malware Protection*. URL: http://unprotect.tdgt.org/index.php/Sandbox_Evasion (visited on 04/13/2017).

[25] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. 6th. Microsoft Press, 2012. ISBN: 0735648735, 9780735648739.

[26] Igor Santos et al. “Opcode Sequences As Representation of Executables for Data-mining-based Unknown Malware Detection”. In: *Inf. Sci.* 231 (May 2013), pp. 64–82. ISSN: 0020-0255. DOI: 10.1016/j.ins.2011.08.020. URL: http://dx.doi.org/10.1016/j.ins.2011.08.020 (visited on 05/31/2017).

[27] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. San Francisco, CA, USA: No Starch Press, 2012. ISBN: 1593272901, 9781593272906.

[28] Parikshit Sondhi. “Feature construction methods: a survey”. In: *sifaka. cs. uiuc. edu* 69 (2009), pp. 70–71. URL: https://pdfs.semanticscholar.org/1faf/80be961715c763bfab82d577e2a86ae65a9a.pdf (visited on 06/01/2017).

[29] Symantec. *Internet Security Threat Report 22*. 2017. URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf (visited on 05/02/2017).

[30] Han Xiao and Thomas Stibor. “A Supervised Topic Transition Model for Detecting Malicious System Call Sequences”. In: *Proceedings of the 2011 Workshop on Knowledge Discovery, Modeling and Simulation*. KDMS ’11. San Diego, California, USA: ACM, 2011, pp. 23–30. ISBN: 978-1-4503-0836-6. DOI: 10.1145/2023568.2023577. URL: http://doi.acm.org/10.1145/2023568.2023577 (visited on 05/31/2017).

Printed by Tryckeriet i E-huset, Lund 2017

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2017-584

http://www.eit.lth.se