UNIVERSITY OF BIRMINGHAM

Behaviour-based malware detection using neural networks

Supervisor: Mohan Sridharan

Author: Ángel Alberto Hamilton López

Student ID: 1972652

MSc Cyber Security

School of Computer Science, University of Birmingham

Birmingham, September the 3rd of 2019

Acknowledgements

To my parents, for giving me the opportunity of studying in another country and for their immense support.

To my friends and family for caring for me, supporting me and motivating me whenever I needed and whenever I didn’t.

To my supervisor Mohan Sridharan for his help and trust.

Abstract

The objective of this project was to research and evaluate the use of neural networks to detect malware based the behaviour of the software.

Typical anti-malware software relies mostly on signatures and other methods of static analysis, which is only effective against already known malware and is much less useful against polymorphic malware and first-day attacks. The common denominator of all malware is that it behaves maliciously so having a detection system based on behaviour would potentially identify any malware as it executes, regardless of it is known, unknown or polymorphic.

For this project we developed a tracing software for windows using Event Tracing for Windows. This software, given an executable file, executes it and generates a log files with certain system calls done by the executed program. These logs are then processed and fed into neural networks to train them into being able to distinguish logs from a malware program or a normal software.

Keywords: Neural Networks, Malware, Machine Learning, Event Tracing for Windows.

Contents

Table of figures

Chapter 1: Introduction

The aim of this project is to research the use neural networks and Event Tracing for Windows [1], as tools to create a functional behaviour-based malware detection software. The final product was not developed in the end, but the research done in this area has provided interesting insight in this area of research.

1.1 State of the art

Malware, short for malicious software, is defined as "Any software designed to cause damage to a single computer, server, or computer network" [2]. Under this definition we find a lot of different types of software like viruses, trojans or ransomware, each of which has different intentions and different strategies to attack the computer. The common denominator of all malware is that it tries to do something damaging to the machine it is running on. Traditionally, anti-malware software has relied on signature detection and heuristic methods, but with 4.4 new malware programs being created every second [3] it is impossible for anti-virus companies to keep up to date on all the new possible signatures and attack patterns. For this reason, a software able to identify a malicious activity within a system without requiring previous knowledge of the specific malware would be a valuable tool for any machine or network.

This problem of defining and detecting malicious behaviour has been tackled in multiple ways, specially using machine learning to approach the problem [4], [5] and [6]. In these cases, the results obtained using different methods were promising and that is why we decided to try our own approach at the problem using our own software solutions in each step of this process. As for the machine learning algorithms we are using, we chose Neural Networks [7] as these have not been tested in any of the related works we found. Neural networks are supervised machine-learning systems and as such are trained with a set of inputs and expected outputs and by trial an error the network slowly learns and adapts. After training, the network can be used to make predictions on new input values. Thanks to their properties, neural networks are used in very different tasks like natural language processing [8], face recognition [9] or self-driving cars [10].

1.2 Objectives

Being a research project, the main objective was not to develop a running software, but instead try out different possibilities to train a neural network able to distinguish malware from normal programs. Among our secondary objectives we have:

- Develop a software able to trace the system calls made by a given executable file during its execution.
- Use the tracer software to generate a dataset by analysing both malware and legitimate software.

- Evaluate different types of Neural Networks using our own dataset to test their usefulness in identifying software

We tried different approaches to the processing of information and the initial parameters of the neural networks in to see if this combination of tools (ETW and neural networks) is viable to make a functional system.

1.3 Overview of this document

In the second chapter we will talk about background information in more detail, as well as other related works that tackle this problem from different angles. In chapter 3 we detail the methodology followed during the project and explain the tracer program and the neural networks used in detail. Chapter 4 contains our thoughts on the project, the achievements and potential improvements to keep working in this line in the future. Finally, chapter 5 is the conclusion and summary of the project, after which we have the references and appendixes.

Chapter 2: Background

In this chapter we will describe background information that will help in order to understand the methodology used. The technologies used in this project are presented and explained in this chapter.

2.1 Malware

As we defined earlier, any software designed to cause damage to a single computer, server or computer network is considered to be malicious software, malware for short. Malware can be classified in different types based on a multitude of criteria. Some of the most famous malware types are:

- Trojan: A malware disguised as something else [11].
- Virus: A malware that inserts itself into other files. Virus have their own subtypes [11].
- Bacteria: A malware which tries to absorb a type of resource, like disk space [11].
- Spyware: A malware that aims to remain hidden while collecting data from the system.
- Ransomware: A malware that disables a system, usually using encryption, until a ransom is paid.

Malware can be analysed using dynamic analysis, running the malware and collect information, or using static analysis, which extract information without executing the malware [6]. Static analysis is more common and safe since it does not risks running the malware and if it is already known, it is easy to find similitudes in the binary files which expose it as a malicious file. In our case we are using dynamic analysis.

The malware we used to build our dataset was obtained from “theZoo” git repository [12]. This repository contains more than 200 different live malware of different types and targeted at different operative systems. We used a selection of those we were able to successfully execute during the sample-collection phase in our project.

2.2 VirtualBox

VirtualBox [13] is an open source virtualization software that allows its users to create virtual machines using a multitude of operative system. VirtualBox is still receiving frequent updates and offers multitude of features.

For this project, we used VirtualBox as a platform to create virtual machines in which to execute and analyse the malware. Virtual machines isolate the malware we execute in them, which greatly reduces the risks which come with executing malware. In addition to that, it makes it very convenient to create new virtual machines and to dispose of the ones already infected with malware.

2.3 Event Tracing for Windows

Event Tracing for Windows [1], also know as ETW, is a tool provided by Microsoft Dev Center that gives programmers the ability to work with trace events and trace sessions. These events provide information about the state of programs during their execution and can be used for debugging, performance analysis or other uses.

In this project we used ETW to track and log the operations realized by the software we area analysing. These logs are the dataset that we then fed into the neural networks. To implement ETW in our program we used the "Microsoft.Diagnostics.Tracing.TraceEvent" library [14] for C#.

2.4 Neural Networks

Neural networks [7] are a popular type of computing system inspired by biology and try to imitate the behaviour of the cells in the brain to learn without being given specific instructions. Neural networks are built by units called cells organized in layers. The neurons from a layer receive data from the layer right before and after doing some calculation, referred as "activation function" they output a value to the layer right after. Neurons can be connected to one or more of the neurons of the following layer by edges. These edges have a weight which adjust how much a neuron is affected by the input coming from that edge, a greater value meaning more influence in the output. Neural networks are usually divided in three parts:

- Input: An input layer collects the information given to the network and produces an output collected by the hidden layers.
- Hidden: A layer, o group of layers, that apply transformations to the information given by the input layer.

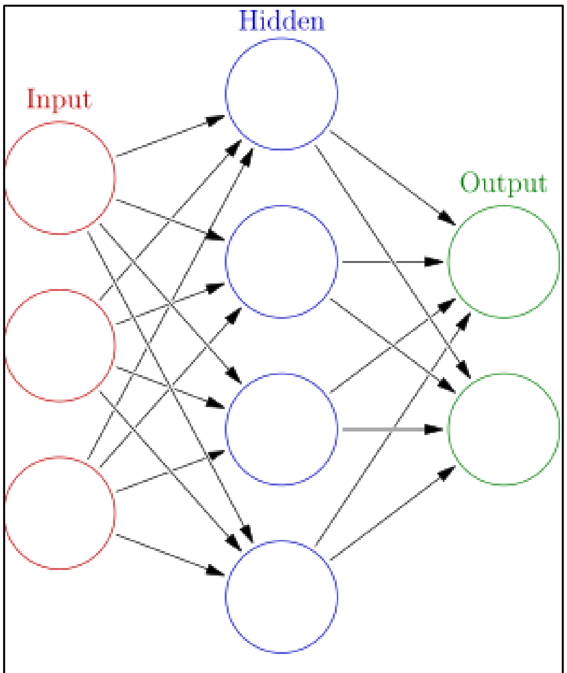


Figure 1. Structure of a neural network. [39]

- Output: This layer receives the data from the hidden layer and outputs a value that is then interpreted.

There exist multitude of designs for neural networks that are used for different purposes. Convolutional neural networks are used for face recognition [9], deep neural networks are used to drive cars [10] and recurrent neural networks are used for language modelling [15] among other examples.

For this project we were specially interested in recurrent neural networks (RNN) which are a type of neural network where the hidden layer has a loop that feeds on itself. This way, some information of the state of the network in each time T can be transferred to the network in T+1, influencing the output. This is property is interesting since we can use it to build an RNN that receives information sequentially, as it is generated in real time, without losing information between each timeframe. Within RNNs, the type we have chosen to focus on is Long Short-Term Memory networks.

2.4.1 Long Short-Term Memory Network

Long Short-Term Memory networks (LSTM) [16] are different from traditional neural networks in that they can be used to analyse a sequence of data rather than a single instance of it. This, combined with their ability to hold information for undefined amounts of time make them ideal for things like speech recognition [17] and other tasks that require to make decisions based not only on the actual input but on past iterations. LSTMs have a very well-defined structure, even though some variations exist, figure 2 shows the most common structure for a LSTM cell. Each cell of a LSTM layer is actually formed by multiple simpler cells:

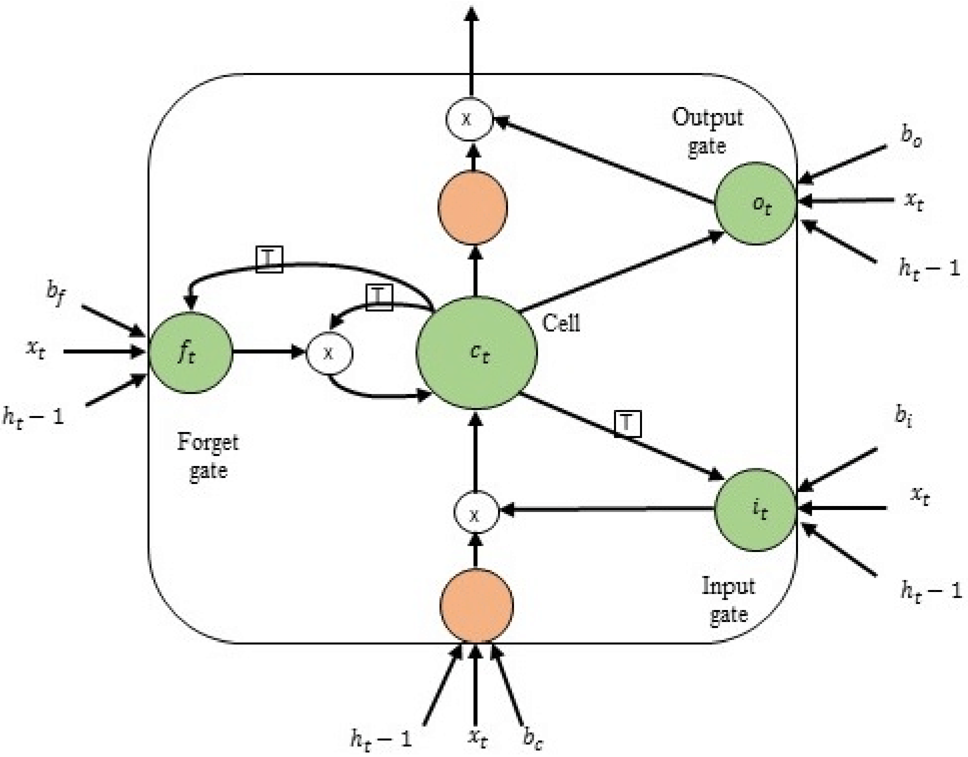


Figure 2. Diagram of the structure of a LSTM cell [40]

- Memory cell: Is the core of the LSTM cell. The information on it is updated each time step and stored for the next one.
- Forget gate: Receives the input information and the state of the memory cell. It decides how much of the information of the cell will be forgotten during this time step.
- Input gate: receives the old state of the memory cell and the input information. This is actually composed by two different operations, the first of them decides which values of the memory cell will be updated and then the other choses the new values.
- Output gate: This gate receives the old cell state and the input. It decides which parts of the updated cell state should be output.

For a much more detailed explanation on LSTMs see [18].

We decided to focus on LSTMs for this project because of their ability to remember pasts states. We can input the logs we have as a sequence and the network might be able to find relationships between log entries distant in time. Also, in a real application the information could be fed to the network in real time as it is produces which would save storage space and allow for a faster reaction.

2.5 Other Machine Learning algorithms

Even though our focus is neural networks, we also wanted to include other already tested Machine Learning algorithms to have something to test the networks against. The two algorithms that we decided to use for this are Decision Trees and Support-Vector Machines (SVMs), the two best performing in [6].

2.5.1 Decision Tree

Decision Trees [19] are an easy way to automate decision-making. A decision tree is made of nodes organized in layers. Each node in a layer is connected to exactly one of the nodes of the layer above and any number of nodes of the layer below. The highest layer, which only contains one node, is called the “root” of the tree and is where the algorithm to make a decision starts. Some nodes are not connected to the layer below. These nodes are called “leaf” nodes and the decision-making algorithm finishes once it reaches one of them. The algorithm to use a decision tree is as follows: Starting from the root, based on one of the attributes of the object being evaluated choose which node connected to the current one you will go to on the next layer. Repeat the process until you reach a leaf.

There exist algorithms able to construct a decision tree from a data set. We will use one of them to create a decision tree based on the same data set we are training the neural networks with.

2.5.2 Support-Vector Machine

Support-vector machines, also known as support-vector networks [20] are a supervised machine learning algorithm. Given a training dataset of elements each belonging to one of two categories, the algorithm will create a N-dimensional space to place the elements of the dataset in. Afterwards it will try to find a gap in the space that divides perfectly all the elements of both categories, as wide as possible. To predict to which group a new element belongs to, the SVM will place it in the space and check which side of the gap it falls on.

The basic SVMs use only two groups for classification, and the gap has to be a straight line. It is possible to use different functions to generate non-linear divisions and use multiple groups for classification.

2.6 Developing tools

2.6.1 C# and Visual Studio

C# is an object-oriented programming language with its roots in the C family of languages [21]. It includes support for component-oriented programming and several useful features like garbage collection and exception handling, similar to C++. It is one of the default languages used in the .NET Framework [22] and Visual Studio [23] making it very convenient for developers wanting to program apps for Windows. Visual Studio offers utilities for the language like debugging, syntax highlighting and default libraries. In addition to that it offers the possibility to download code using the NuGet packet manager.

We decided to use C# because it has a functioning ETW library. We also tried C++, but the library was missing a lot of the documentation and it didn't work as intended. Visual Studio was the default choice for an IDE since it includes a lot of quality of life features that make the task of programming in C# much easier. It also allows for a quick installation of the ETW and the Newtonsoft.Json [24] libraries which we needed for our tracer program.

2.6.2 Python and TensorFlow

Python [25] is an interpreted general-purpose scripting programming language. Python is still on development, currently at the version 3.7.4 and is one of the most popular programming languages used from a multitude of applications from web apps to education to data science. It uses indentation rather than punctuation to delimit block of codes, this combined with its dynamic typing and relatively simple syntax makes python an easy language to read and write. We installed Python with the Anaconda distribution [26], which also includes libraries like NumPy [27] and TensorFlow [28], specialized in machine learning and data processing. We chose PyCharm [29] as our ide because it includes a lot of utilities to program in Python like built-in python console, syntax highlighting and debugging. It also has a specialized version that includes anaconda, making it very convenient for this project.

TensorFlow is a very powerful open-source library that includes a lot of tools for machine learning and data science. It has functions to create and train multiple types of neural networks in a very easy way, which helps us avoid potential errors while implementing the algorithms. TensorFlow was our default option for a library to build the neural networks with due to its fame and ease of use, and since we wanted to use it we decided to go for Python as our programming language for that part of the project. There are some libraries for C# that try to adapt TensorFlow, but they do not work as intended and lack documentation. TensorFlow can also be configured to use Nvidia GPUs [30], increasing its speed.

2.6.3 Scikit-Learn

Scikit-learn [31] is a python module that contains multiple tools for data mining and data analysis. It is a project built by the community and is still in development. It includes a very easy to use implementation for both decision trees and SVMs, both of which can be created, trained and tested in three lines of code. This was perfect for our project and that is the reason we decided to use it.

Chapter 3: Methodology

In this chapter we are going to describe in detail how exactly we did our research. We will also describe all the software we made for it and discuss the most important implementation details. Figure 3 shows a diagram of our system, from extracting the data of executable files to training the networks and getting the models.

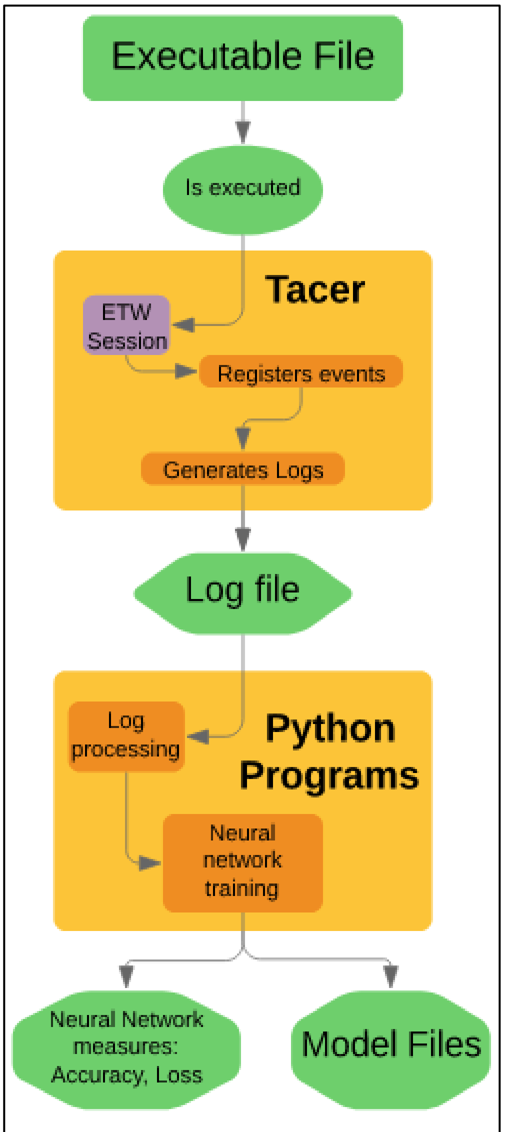
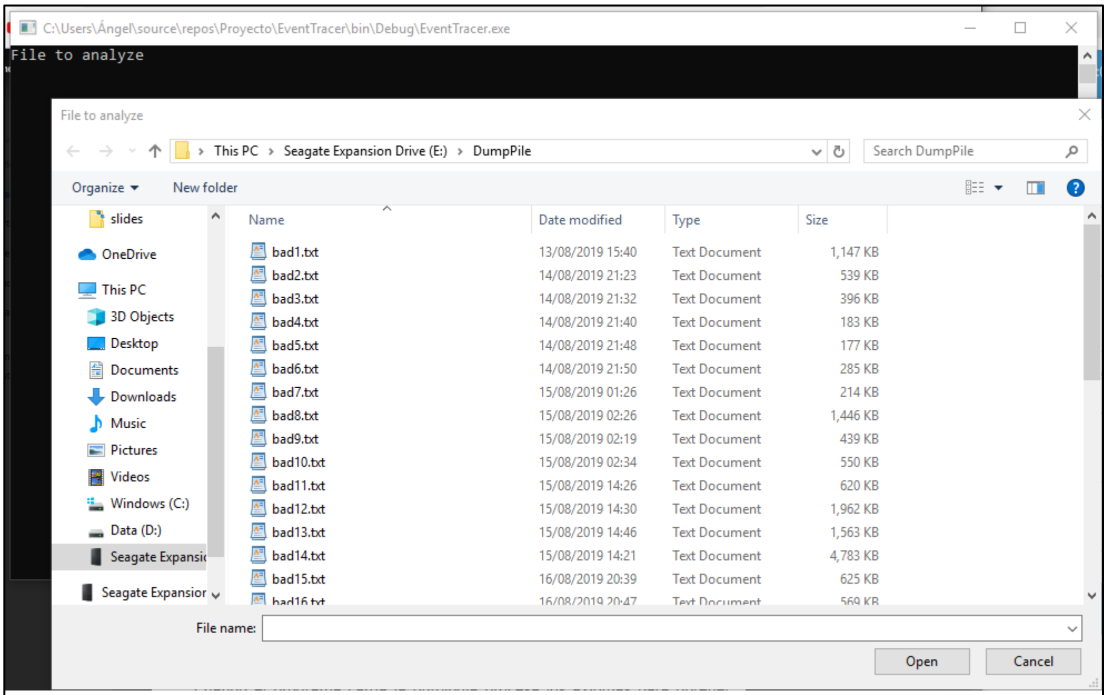


Figure 3. Diagram of the methodology

3.1 Tracer program

The Tracer was designed to get as much information as possible of a program during its execution. When executed, the program opens a console window and a file-search dialog that allows the user to easily look for the executable program they want to analyse. After choosing a file, the Tracer opens a second dialog window to look for the output file. This output file might have any extension, but it will write the output logs as text. Once both files have been chosen the Tracer will start an ETW kernel session and, after a delay of 2 seconds, it will execute the file we chose to analyse. It will track the process created by the file and all its child process, showing in the console whenever a new process is created or terminated. It will dump all the information obtained on the output file. When all the process tracked have been terminated the console will output a message so the user knows that I can close the tracker program. The log files are written in real time, making it possible to close the tracer program mid-execution and still get a log of all the activity until that moment. This is especially useful when analysing ransomware and other malware that will encrypt or disrupt the log file during its execution since we can extract a copy of the log file before it is corrupted.



```
File to analyze
```

Name	Date modified	Type	Size
bad1.txt	13/08/2019 15:40	Text Document	1,147 KB
bad2.txt	14/08/2019 21:23	Text Document	539 KB
bad3.txt	14/08/2019 21:32	Text Document	396 KB
bad4.txt	14/08/2019 21:40	Text Document	183 KB
bad5.txt	14/08/2019 21:48	Text Document	177 KB
bad6.txt	14/08/2019 21:50	Text Document	285 KB
bad7.txt	15/08/2019 01:26	Text Document	214 KB
bad8.txt	15/08/2019 02:26	Text Document	1,446 KB
bad9.txt	15/08/2019 02:19	Text Document	439 KB
bad10.txt	15/08/2019 02:34	Text Document	550 KB
bad11.txt	15/08/2019 14:26	Text Document	620 KB
bad12.txt	15/08/2019 14:30	Text Document	1,962 KB
bad13.txt	15/08/2019 14:46	Text Document	1,563 KB
bad14.txt	15/08/2019 14:21	Text Document	4,783 KB
bad15.txt	16/08/2019 20:39	Text Document	625 KB
bad16.txt	16/08/2019 20:47	Text Document	569 KB

Figure 4. First dialog window of the tracer

3.1.1 Implementation

To implement the Tracer, we used C# as our language and Visual Studio as our IDE. To start we created a new C# console program project on Visual Studio and then, using the NuGet package manager, we installed the packages *Microsoft.Diagnostics.Tracing.TraceEvent* and *Newtonsoft.Json*. The first one was necessary because it includes the ETW library we wanted to use and the second one helped us to output the logs on JSON format.

The main (and only) file of this program is EventTracer.cs can be found in our GitHub page [32] under at EventTracer/EventTracer.cs. We used only one namespace *EventTracer* and one class *Tracingfromfile* for the whole Tracer program. This was to keep the code as simple as possible since the Tracer program itself is very simple to follow and understand. All the code for this part was made by us, based on the tutorials found in the Perfview GitHub repository [14].

The Tracingfromfile class has six static variables:

- ***tracking***: A list with the PIDs of the process we are interested in tracking. Initially only contains the process created by the executable and then its children (if any) are added as they are created. PIDs are also deleted from the list as the process are terminated.

- ***KernelSession***: A variable required to hold the ETW session.
- ***baseProcess***: This variable of the Process class is used to execute the file to analyse.
- ***dumpfile*****:** Path to the output file.
- ***d*****:** XmlDoucment variable used as a middle step to generate the JSON output. Rather than creating a new variable every time we chose to have a single variable updated every time we need it.
- ***finder*****:** Necessary to open the dialog to find the files.

All of these are static class variables because they need to be accessed by different threads and the information must be shared between all of them.

The main function starts by checking if the Tracer is being executed with admin privileges because it needs them to successfully create a kernel event tracing session. If it does not have them, a message will show in the console terminal and the program will close after the user presses any key. If it does have them, then it will create and execute a new thread. This first thread opens an *OpenFileDialog* with the *finder* variable and ask the user for the file to analyse. The file chosen by the user is stored in a new variable *filePath* and checked to see if is exists. This double check is left from when the program required the user to manually input the file path and was left in as an extra security measure. It then writes the file path on console to show it to the user and opens a second dialog, using again *finder*. This time the file chosen is stored in *dumpfile*. The output file chosen is checked in case it is the same as the input file, which is something that can occur if the second window is closed right after it opens because, since we are using the same variable, when the second dialog opens it will default to the input file the user chose. If everything goes well the Tracer will clean the output file. Finally, this thread will create a new *Tracingfromfile* object using *filePath* as a parameter and execute its *begin* function.

The initialization function for the class *Tracingfromfile* takes a file path to the file the user wants to analyse as only parameter. It begins by initializing *tracking*, *d*, and *KernelSession* and afterwards it subscribes the ETW session to all event providers. Each event requires a handler and a type of event with a handler not assigned will do effectively nothing. This is way we chose to subscribe the session to all the events and then add handlers to the ones we are interested in processing. After a lot of experimentation through trial and error we ended up with the events we thought will give the most amount of useful information. There are some events we wanted to include, mainly main memory access and modification, but were unable to due to the sheer number of events produced each second. In some of our tests, after running the Tracer for 10 minutes we ended up with almost 4 gigabytes of data that was only generated during the first 5 milliseconds of execution. To make the Tracer work in a reasonable time frame we had to cut out some of those events. All the events we use are handled by the same function *general* apart from the *ProcessStart* and *ProcessStop* events. We will discuss those functions later in this chapter. The initialization function continues by setting up *baseProcess* to execute the file we gave to it as a parameter.

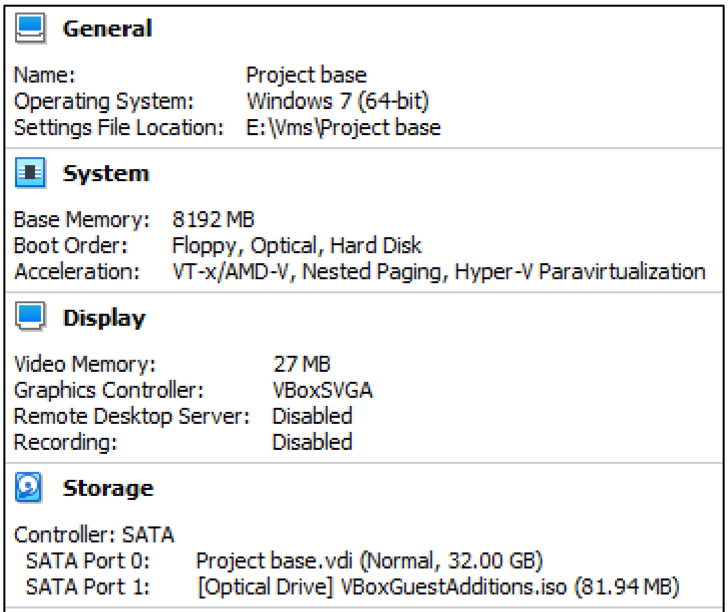
The *begin* function is executed after the initialization of the class instance by *main*. It starts by creating a new thread object which will simply wait for 2 seconds, start the input program and add the process ID generated by it to *tracking*. Afterwards it starts the thread and starts the ETW session. This way we make sure the session is running when we execute the file.

The *general* function is the function we use to handle all the events. It receives the data of the even as an input and check if the process ID of the event is one of the process we are tracking, ignoring it if it is not the case. All the *TraceEvent* instances have a function *Dump* which outputs all the data of the even in XML format. We use this function to load the information in *d* and then using the conversion method provided by the *Newtonsoft.Json* library we append it to the output file.

Both *processStarted* and *processStopped* functions handle the events of their same name. They have a functionality similar to *general* with some variations. *ProcessStarted* checks every new process that starts and if the parent process ID is equal to one of the process ID we are tracking then the PID of the new process is added to the *tracking* list and a log is produced. Whenever a process we are tracking is terminated, *processStopped* logs the event and then removes the PID form *tracking*. After removing the last PID from the list, it outputs a message to the user through the console.

3.1.2 Generating the dataset

The Tracer can be used to run and log normal executable files without any complications. However, tracing malware is more complicate since it requires to take security measures to avoid the malware from damaging our systems. At the same time, we want the malware to be executed freely so we can log all its activities. To both avoid being affected by it and at the same time allow it to run rampant in a system we decided to use virtual machines. Using VirtualBox we created a base virtual machine in which we disabled windows defender and installed the Tracer.



General

Name: Project base
Operating System: Windows 7 (64-bit)
Settings File Location: E:\Vms\Project base

System

Base Memory: 8192 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, Hyper-V Paravirtualization

Display

Video Memory: 27 MB
Graphics Controller: VBoxSVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage

Controller: SATA
SATA Port 0: Project base.vdi (Normal, 32.00 GB)
SATA Port 1: [Optical Drive] VBoxGuestAdditions.iso (81.94 MB)

Figure 5. Specs of the base virtual machine

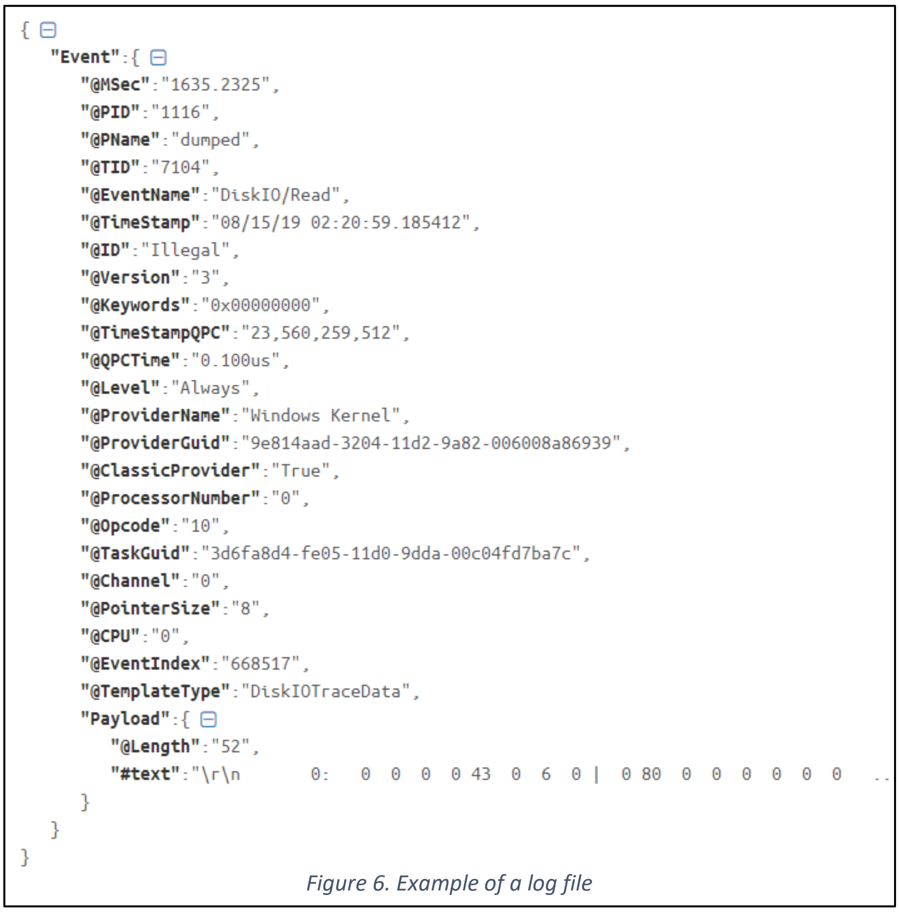
We downloaded the malware binaries from theZoo and loaded them into an USB storage unit. Then we proceeded to clone the base machine and boot it with the USB storage unit connected. In the virtual machine we executed a malware binary with the Tracer and generated an output file on the same USB storage. This trial an error, the different malwares in theZoo are made for different versions or operative systems so some of them didn't even work. In some cases it was required to connect the virtual machine to the internet and output the log in our personal One-Drive [33] directory so we could retrieve a copy before it was corrupted. After executing the malware and getting the log file, we deleted the virtual machine and created another clone to start the process again.

We created a total of 25 malware log files and 25 non-malware log files. The list of programs used for each one can be found in our GitHub.

3.1.3 Log files

The log files created were named "goodX.txt", if they came from a legit software, and "badX.txt" if they came from malware, "X" being a number used to distinguish between them. The logs are written in JSON format and use the .txt extension to make it easier for us to open them with basic text edition programs. Each file contains an arbitrary number of log events. Each log event has the structure shown in figure 6, with 25 fields with information about the file, 26 in some event logs where the parent PID is included.

All the log files obtained are in our GitHub, at NeuralNetworks/DumpPile. These are the log files as they were fed into the following parts of the project.



```
{ ⊟
   "Event":{ ⊟
      "@MSec":"1635.2325",
      "@PID":"1116",
      "@PName":"dumped",
      "@TID":"7104",
      "@EventName":"DiskIO/Read",
      "@TimeStamp":"08/15/19 02:20:59.185412",
      "@ID":"Illegal",
      "@Version":"3",
      "@Keywords":"0x00000000",
      "@TimeStampQPC":"23,560,259,512",
      "@QPCTime":"0.100us",
      "@Level":"Always",
      "@ProviderName":"Windows Kernel",
      "@ProviderGuid":"9e814aad-3204-11d2-9a82-006008a86939",
      "@ClassicProvider":"True",
      "@ProcessorNumber":"0",
      "@Opcode":"10",
      "@TaskGuid":"3d6fa8d4-fe05-11d0-9dda-00c04fd7ba7c",
      "@Channel":"0",
      "@PointerSize":"8",
      "@CPU":"0",
      "@EventIndex":"668517",
      "@TemplateType":"DiskIOTraceData",
      "Payload":{ ⊟
         "@Length":"52",
         "#text":"\r\n       0:   0  0  0  0 43  0  6  0 |  0 80  0  0  0  0  0  0   ..
      }
   }
}
```

Figure 6. Example of a log file

Even though we have 25 from each type, the malware log files are usually much smaller than the normal software. Individually, we have a total of 77874 “good” logs, 76.16% and 24380 “bad” logs, 23.84%. We did not want to cut any information out having a small dataset already and fearing that we might remove important information. We decided to leave the difference in. In a real case, the great majority of the software that is executed is going to be good software so having approximately three quarters of our dataset being good is a more realistic training environment.

3.2 Python Machine learning

For the Machine learning section of this project we separated the functionalities in different files. All files refences in this section can be found in the NeuralNetworks directory in our repository. To use them, the first thing the we did was to process the log into a more manageable format and remove information that we are not planning on using. Using the processed log then we experimented with DFFs and LSTMs using different combinations of parameters for the configuration, different log information and different shapes of networks.

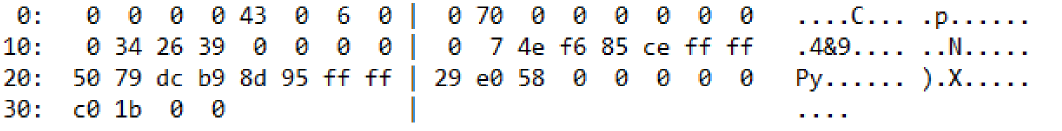
The general details of the implementation of the different files are explained in this chapter. Exactly what parameters we used and any other specific details will be described for each case during chapter 4 when we discuss the results.

3.2.1 Log processer

The main reason why we needed to further process the log files is because some of the fields are not numeric which makes it impossible for the neural networks to use them straight from the Tracer. In addition to that, this gives us the opportunity to filter the information. From all the fields that logs have, we narrowed down our selection to the following ones:

- The ID of the process. Initially we used the value provided by the log but we then realized that, due to how our data collection process, malware process usually had a lower PID than legitimate ones. This caused an artificial difference between them which will not happen in a real scenario, so we decided to get rid of that by using relative PIDs. The first PID to appear in any given log file is replaced by a “1”, the second one by a “2” and so on.
- The ID of the thread that runs the process. We also used the raw values on the beginning of our experimentation and decided to give them relative values the same way we did with PIDs due to the exact same reason.
- The Time stamp of the event, in milliseconds measured since the last time the system was turned on. This also had a big value difference between the malware and the normal software. Since the malware was executed in virtual machines that were turned on just before executing the Tracer, their time stamps were orders of magnitude lowers that the software executed on our machine which had been running for a log time. The logs of the events are written in chronological order and to give them relative values we take the first time stamp as 0 and we subtract the original value of every subsequent time stamp.

- The Event Name. This name distinguishes the operation that the event log refers to. This is very useful information because it states what the process was trying to do. It is a text field, so we used a function to get all the potential ENs from all log files and created a dictionary that assigned an integer to each EN. We found a total of 12 and if will probably need to be update whenever more logs with new ENs are added.
- The payload. This is the most interesting field and at the same time the most complicated to process. The payload value is a text containing a variable number of hexadecimal values followed by a representation of those values in ascii and it also contains separators, indexes and spaces to make it easier to read for the user (figure 7). To process this into a numeric value we can work with we tried using a hash function, but it would just lose information in the process and we wanted to keep as much information in as possible so this approach was discarded. Our final idea consisted of the following steps:
    - Use regular expressions to extract get a string with just the hexadecimal values one after the other. The regex used can be found in NeuralNetworks/LogProcesser.py. We end this step with one single very long hex value.
    - We turn the hex value into a decimal value. It is still very long to be processed so we need to split it into smaller numbers.
    - Using a *split* function, we transform the decimal value into a string and, starting from the most significant digit, we take 9 digits at a time and add them as an int to a list. If the last number would have less than 9 digits, we add 0 to the left until we have 9 digits.
    - We need one regular input size to be able to train the networks, so we need every processed log to have the same number of payload pieces. We decided to use a total of 60 pieces, each of them of 9 digits, which will allow to split every payload in our dataset. If the payload is not bit enough to fit all 60 pieces, we se the value of the unused pieces to 0. This number 60 will probably need to be changed in the future if we include in the dataset a payload that will not fit into the limit.



```
 0:   0  0  0  0 43  0  6  0 |  0 70  0  0  0  0  0  0   ....C... .p......
10:   0 34 26 39  0  0  0  0 |  0  7 4e f6 85 ce ff ff   .4&9.... ..N.....
20:  50 79 dc b9 8d 95 ff ff | 29 e0 58  0  0  0  0  0   Py...... ).X.....
30:  c0 1b  0  0             |                           ....
```

Figure 7. Payload text value

After the processing the log using the split method, we will have the same number of files, with the same names this time in NeuralNetworks/ProcessedLogs/Splitted and with the same number of log entries. Each log entry having the following structure:

- **PID**: A positive integer between 1 and the total number of different PIDs in the file.
- **TID**: A positive integer between 1 and the total number of different TIDs in the file.

- **TS**: A positive integer.
- **EN**: A integer between 1 and 12.
- **PL**: An array of 60 positive integers up to a value of 999,999,999 (under the 32bit limit)

3.2.2 Data Processing

Even after processing the logs we still need to get the information into a NumPy array. To do that we have NeuralNetworks/DataProcess.py, a library made by us with multiple functions to process to get the information into a NumPy array in many different ways, depending on what our needs are for the specific experiment. The file contains a lot of unused code and functions from past experiment. Even though the functions are different, all of them follow the same basic structure.

The functions require 3 parameters: the route to take the input files from, a number that indicates the first file to be used and a number that indicates the last. For example, with the parameters "ProcessedLogs\Splitted\", "1" and "20" the function will process the files from "NeuralNetworks\ProcessedLogs\Splitted\bad1.txt" to "bad20.txt" and form "good1.txt" to "good25.txt". The functions meant for LSTM also have a parameter that indicates how many logs should be put together in a single temporal sequence. For each "bad" and "good" file, the function reads and loads it into a json array. The array is then appended into another array *d* log by log. For each log in *d* an array *l* is loaded with binary value that indicates if that log comes from a "bad" file or a "good" file. For example, *l(100)* will be 1 if *d(100)* comes from a "bad" file, and 0 if it comes from a "good" file. After that, the function runs over *d* and for each log it creates an array with only the values, not the keys, and adds that as a NumPy array object to the array *npdata*. The array added will be just a list of 64 positive int values. We also add the binary from *l* to *nplabel*. *Npdta* and *nplabel* are kept between files so they will end up with all the logs of all the files we chose to process. Finally, *nplabel* and *npdata* are transformed into NumPy arrays and loaded into the output.

The output of the program is an array with two elements. *Output(0)* is a NumPy array with all the logs we are going to use for training and *Output(1)* is a NumPy array with binary values that indicate if each log in *Output(0)* is or is not malware.

3.2.3 Neural Networks

Even though the DFF and LSTM are in different files both of them are structured the same way, loosely based on one of the official TensorFlow tutorials [34]. First, we declare a couple of variables to easily alter the training and data parameters and after that we execute one of the data processing functions and get the output. We run the same data processing function a second time targeting different files to get a test dataset. The model is declared and compiled and finally we run the training. After the training we print on console how many "good" logs are and how many "bad" logs are, for debugging purposes, and then we run the tests using the testing dataset. After that we also show how many "good" and "bad" logs are in the testing dataset.

We ended up using cross-validation [35], a method in which the full dataset is divided in smaller blocks, 10 for a 10-fold cross-validation, and one of them is chosen to test while the others are the training dataset. This is repeated until all the blocks have been testing sets. For our implementation of this the code is similar to the normal algorithm, except that the training and testing code is in a loop that gets a new block every iteration. The blocks are not stored separately but generated as needed to save space.

All neural network used have the same output layer: a 2-neuron layer with a SoftMax activation function [36]. One of the neurons means 1 (malware) and the other means 0 (normal software) and the SoftMax function makes so that the total value of both neurons has to be 1. This makes it so the neural network does not output a binary value, but rather outputs how sure about the input being each category, so it can output 0.75 for 1 and 0.25 for 0 meaning that it is a 75% sure the log it received is malware.

3.2.3 Decision Tree and SVM

The Decision Tree algorithm and the SVM have their own files, which are almost identical to one another. First, they make the dataset by calling a function from DataProcess.py and create an algorithm instance, decision tree or SVM. From the training set we select randomly a percentage of them to be the test set and run the training of the algorithm. Finally, we run the testing and output the results on screen. The code implemented was based on the official Scikit learn tutorials [37] and [38].

Chapter 4: Discussion and Results

In this chapter we are going to talk about the results of our tests and compare them. Afterwards we will evaluate the results and discuss the project in general. It is important to note that we will talking about the last iterations of each of the algorithms used, which we consider to be the best ones.

We started the experiment using the hashing payload along with the split payload to compare results. The hashed payload yielded consistently worse results on every test so we scraped the idea entirely. The files are still in the repository, but we will not talk about them in the results. After some days of testing we concluded that the best combination of data was the one described on chapter 3.2.1 so all the algorithms described here are using an identical dataset.

To evaluate the effectiveness of each algorithm we used the following metrics: accuracy of the training set and accuracy of the testing set, as a general measure in both cases; true positive rate, as a measure of how much of the malware is actually detecting; and false positive rate, to check how many false alarms this method would raise. We also did for each technique a "Full file test" in which we used the algorithm to evaluate each log file and output if it is good or bad. The way in which it was done varies for each algorithm. This was made to simulate a more real scenario because while an algorithm might fail classifying individual logs it might be still be able to detect that a log file contains enough malicious activity to consider it bad.

4.1 DFF

To train the DFF we used a network of 4 layers with 8 neurons in the first layer, 64 on the second one with a TanH activation function [36], 32 on the third one with ReLU and the output layer has 2 neurons with a SoftMax function as discussed in 3.2.3. It was trained using the files from 6 to 25 ("good6" to "good25" and "bad6" to "bad25") and tested using 10-fold cross-validation.

- Accuracy of the training: 75.62%
- Mean accuracy of the testing: 74.89%

We ran a test in which we asked the DFF to predict the logs of the training dataset (files from 6 to 21) to get the true positive rate and the false positive rate. We did the same with files 1 to 5 to see how it behaved with files it had never seen.

- TPR for testing dataset: 28.16%
- FPR for testing dataset: 3.6%
- TPR for new dataset: 50.98%
- FPR for new dataset: 4.84%

For the full file test we sent all the logs of each file through the network. For each log we obtain the prediction that it is malware, a float number between 0 and 1, add it to an accumulator and divide the accumulator by 2. Then if the accumulator is over a certain value, we stop the process and consider this file as bad. This is trying to simulate a real-life scenario in which you would want to stop the malware during its execution. This way, the malware is classified as malware as soon as there are group of logs in a row with a high probability of being bad. The value of this threshold can be modified and depending on it, the full file experiment will yield vastly different results. The lower the threshold value, the more true and false positives. These are the metrics for a full file experiment using a threshold of 0.7:

- Accuracy: 84%
- TPR: 80%
- FPR: 16%

4.2 LSTM

The LSTM uses only 2 layers, one with 16 LSTM cells and the output layer explained in chapter 3.2.3. LSTM trains using series of data rather than individual logs, so we need to specify a temporal window size. For this experiment we decided to use a value of 75, which means that it will take the first 75 logs of each file and grout them into a temporal sequence, then the next 75 and continue. If there are less than 75 logs left, the left-over logs are ignored. We tested adding log full of 0s as padding, but it yielded worse results. Again using 10-fold cross-validation training with the files 6 to 25 this time:

- Accuracy of the training: 81.4%
- Mean accuracy of testing: 80.24%

Again, we ran a test to predict the logs of the training dataset to get the true positive rate and the false positive rate and on the unused files to see how the LSTM reacted:

- TPR for testing dataset: 9.8%
- FPR for testing dataset: 0.88%
- TPR for new dataset: 0%
- FPR for new dataset: 0.5%

We used the same full file test that we did for the DFF, but seen how heavily biased this network is towards non-malware the threshold is as low as 0.35:

- Accuracy: 52%
- TPR: 56%
- FPR: 52%

4.3 LSTM with moving temporal window

This neural network is exactly the same as the previous LSTM, what changes is the dataset. Instead of taking 75 logs at a time and then skipping to the next 75, we have a window of size 75 that moves one log at a time. Starting with logs 0 to 74, the next group will be 1 to 75, then 2 to 76. This causes the majority of the individual logs to be repeated many times, inside slightly different groups each time. We tried this approach because in a real life scenario a software could have a buffer of size N that as more files get added the old ones are deleted and it gets run through the network from time to time, creating a similar effect to what we have. It also has the added characteristic of artificially increasing the dataset size.

- Accuracy of the training: 81.79%
- Mean accuracy of testing: 80.81%

Same tests as with previous LSTM:

- TPR for testing dataset: 02.44%
- FPR for testing dataset: 0.72%
- TPR for new dataset: 0.19%
- FPR for new dataset: 0.87%

We used the same full file test, with threshold 0.35:

- Accuracy: 62%
- TPR: 52%
- FPR: 72%

4.4 Decision Tree

The decision tree algorithm was trained using a 60% of the total dataset and the other 40%, chosen randomly, was the testing set. Even though there was some variation due to the randomness of the training set and test set separation, the results were very consistent:

- Accuracy of the testing set: 82%
- True positive rate: 39%
- False positive rate: 9%

A decision tree will always have a 100% accuracy on the training set so we will disregard that metric in this case.

For the full file test, for each file we ran all its logs through the decision tree and did the average. If more than 50% of their logs were considered bad, then we classify the whole file as malware. The metrics for the full test file where:

- Accuracy: 86%
- True positive rate: 68%
- False positive rate: 0%

As stated earlier, the Decision tree will always have a 100% accuracy on the training set, and most of the log passed to the tree during the full file analysis were part of the testing set. These metrics are not very useful, but they reveal that the tree has a clear bias towards classifying logs as non-malware. This is reasonable, since the majority of the logs of our dataset are from normal software.

4.5 SVM

The SVM used 70% of the dataset as training and a random 30% as testing. There was some variation due to this randomness, but the results were consistent:

- Accuracy of the testing set: 79%
- True positive rate: 39%
- False positive rate: 13%

Similarly to the decision tree, a SVM is made to perfectly fit the training set, so the training set accuracy is 100%.

We made the full file test in the exact same way that for the decision tree. For each file we ran each log through the SVM, and if the majority of them were classified as bad, then the whole file is classified as bad

- Accuracy: 82%
- True positive rate: 64%
- False Positive rate: 0%

The SVM results are very similar to those of the decision tree, for the same reasons again. Most of the logs on the files were part of the training set, so the SVM will have a 100% accuracy on those, and it also has a bias towards classifying files as good.

4.6 Result Comparison

For DFF, LSTM and LSTM window, the TPR and FPR are the average of TPR and FPR with seen and unseen data.

	ACC TRAINING	**ACC TEST**	**TPR**	**FPR**	**ACC FULLFILE**	**TPR FULLFILE**	**FPR FULLFILE**
DFF	75.62%	74.89%	39.57%	4.22%	84%	80%	16%
LSTM	81.4%	80.24%	4.9%	0.69%	52%	56%	52%
LSTM WINDOW	81.79%	80.81%	1.315%	1.59%	62%	52%	72%
DT	-	82%	39%	9%	86%	68%	0%
SVM	-	79%	39%	13%	82%	64%	0%

After seeing the results, it is clear that the DFF is clearly the best over-all algorithm in this specific case, followed closely by SVM and DT. Both LSTMs have good accuracy but very low TPR and FPR, which means that they get that accuracy by labelling almost the entirety of the logs as non-malware. This is a direct consequence of choosing to have an asymmetric dataset.

4.7 Discussion

The results of this projects are interesting and show that using neural networks for behavioural analysis is possible, even when it may not be the best option. It also shows that the ETW library can obtain useful information, seen how all the algorithm had a certain degree of accuracy. This proof of concept shows that combining neural networks and ETW to create a software capable of using behavioural analysis to detect malware is possible and viable. What it does not show it how successful it will be.

The main problem with this project is that the dataset was extremely small, just 25 malware files and 25 other files is not nearly enough to get any meaningful conclusions about its effectiveness. Also, the difference between the size of the malware and the normal software did show, and all the algorithms had a bias towards classifying the inputs as non-malware. This is a good strategy to get a good accuracy, but in a real-life scenario we will rather have 100 false positives than a false negative. Repeating the training using recall as optimization parameter can be a good idea.

This project is lacking in scale, everything that we think can be improved requires more time, work, and data. Study other times of neural networks will be interesting and having much more time to fine-tune the parameters will be extremely useful. Also, as we already said, if someone tries to continue this work the first thing that they should do is generate more data with the Tracer. As a research project, its biggest shortcoming has been all the time we spent experimenting around and trying things that went nowhere, and someone with more experience could have done what we did in much less time.

Said that, we are still very proud of our results and how we achieved them. We managed to create all the software and data we used from ourselves. Even if it would have been more efficient to get some already made database of malware data and ran it through the neural networks, we decided to do everything ourselves. We didn’t just get two already made things and put them together, we built everything from scratch selecting the data, which data, how to collect it, how to process it… And that requires much more work and much more knowledge, some of which we had to learn as we worked.

Chapter 5: Conclusion

This project proposes a unique way of approaching the creation of anti-malware software by researching the possibility of using Event Tracer for Windows to obtain the runtime data and neural networks to process it. We have a program that tracks the execution of a software and logs important information of its activities during runtime. A python program processes these logs so then they can be used to train neural networks. We then chose to train deep feed forward neural networks and long short-term memory networks. We have shown how they perform using different metrics and compared them with already tested algorithms like decision trees and support vector machines. In the end, we concluded that experiments were successful and a good proof of concept, but it will require much more time and data to answer the question to if this approach is better than others.

Turning back to the objectives of our project, we can say that we have successfully met them. Our objectives what to try and experiment with the idea to see how far we can take this by implementing all the software and getting the data ourselves, and we managed to do exactly that. Even when the results were inconclusive, this research shows that doing thing this way is possible.

References

[1] Microsoft, “Event Tracing,” 31 5 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal. [Accessed 3 September 2019].

[2] Microsoft, “Defining Malware: FAQ,” 01 04 2009. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10). [Accessed 3 September 2019].

[3] AVTest, “AVTest Security Report 2018/2019,” 2019. [Online]. Available: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf. [Accessed 3 September 2019].

[4] K. Chumachenko, *Machine Learning methos for malware detection and classification,* kaakkois-suomen ammattikorkeakoulu, 2017.

[5] P. T. C. W. a. T. H. Konrad Rieck, “Automatic Analysis of Malware Behavior using machine learning,” *Journal of Computer Security,* 2011.

[6] A. Dhiman, *Malware detection and classification using machine learning techniques,* Department of Computer Science and Engineering, Indian Institute of Technology, 2018.

[7] S. Haykin, Neural networks: a comprehensive foundation., Prentice Hall, 19994.

[8] J. W. Ronan Collobert, “A unified architecture for natural language processing: deep neural networks with multitask learning,” no. ICML '08 Proceedings of the 25th international conference on Machine learning, pp. 160 - 167, 2008.

[9] S. Lawrence, “Face recognition: A convolutional neural-network approach.,” *IEEE transactions on neural networks,* vol. 8, no. 1, pp. 98 - 113, 1997.

[10] T. Tian, “Deeptest: Automated testing of deep-neural-network-driven autonomous cars.,” *Proceedings of the 40th international conference on software engineering.,* pp. 303 - 314, 2018.

[11] M. Kantarcioglu, *Introduction to malware,* The University of Texas at Dalas.

[12] Y. Nativ, “theZoo - A Live Malware Repository,” 2014. [Online]. Available: https://github.com/ytisf/theZoo. [Accessed 12 July 2019].

[13] VirtualBox, “VirtualBox home page,” VirtualBox, [Online]. Available: https://www.virtualbox.org/. [Accessed 22 June 2019].

[14] Microsoft, “Perfview,” [Online]. Available: https://github.com/Microsoft/perfview. [Accessed 20 July 2019].

[15] T. Mikolov, “Recurrent neural network based language model,” *Eleventh annual conference of the international speech communication association,* 2010.

[16] S. a. J. S. Hochreiter, “Long short-term memory.,” *Neural computation,* vol. 9, no. 8, pp. 1735 - 1780, 1997.

[17] A. S. F. B. Has¸im Sak, “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling,” Google, 2014.

[18] C. Olah, “Understanding LSTM Networks,” 27 August 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed 29 June 2019].

[19] J. R. Quinlan, “Induction of decision trees,” *Machine learning,* vol. 1, no. 1, pp. 81 - 106, 1986.

[20] C. Cortes and V. N. Vapnik, “Support-vector networks,” *Machine Learning,* vol. 20, no. 3, pp. 273 - 297, 1995.

[21] Microsoft, “A Tour of the C# Language,” 04 05 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. [Accessed 05 September 2019].

[22] Microsoft, “.NET,” [Online]. Available: https://dotnet.microsoft.com/. [Accessed 28 June 2019].

[23] Microsoft, “Visual Studio,” [Online]. Available: https://visualstudio.microsoft.com/. [Accessed 28 June 2019].

[24] J. Newton-King, “Json.NET,” [Online]. Available: https://www.newtonsoft.com/json. [Accessed 20 July 2019].

[25] Python, “Python,” [Online]. Available: https://www.python.org/. [Accessed 05 September 2019].

[26] Anaconda, “Anaconda Distribution,” [Online]. Available: https://www.anaconda.com/distribution/. [Accessed 1 August 2019].

[27] NumPy, “NumPy,” [Online]. Available: https://numpy.org/. [Accessed 01 August 2019].

[28] Tensorflow, “TensorFlow,” [Online]. Available: https://www.tensorflow.org/. [Accessed 1 August 2019].

[29] JetBrains, “PyCharm,” [Online]. Available: https://www.jetbrains.com/pycharm/. [Accessed 1 August 2019].

[30] TensorFlow, “GPU support,” [Online]. Available: https://www.tensorflow.org/install/gpu. [Accessed 22 August 2019].

[31] Scikit-learn, “Scikit-learn,” [Online]. Available: https://scikit-learn.org/stable/index.html. [Accessed 8 September 2019].

[32] Á. Hamilton, “TFM,” 13 July 2019. [Online]. Available: https://github.com/alu0100888102/TFM. [Accessed 13 July 2019].

[33] Microsoft, “One-Drive,” [Online]. Available: https://onedrive.live.com/about/en-gb/. [Accessed 06 September 2019].

[34] TensorFlow, “Train your first neural network: basic classification,” [Online]. Available: https://www.tensorflow.org/tutorials/keras/basic_classification. [Accessed 03 August 2019].

[35] Wikipedia, “Cross-validation,” [Online]. Available: https://en.wikipedia.org/wiki/Cross-validation_(statistics). [Accessed 5 September 2019].

[36] Wikipedia, “Activation Function,” [Online]. Available: https://en.wikipedia.org/wiki/Activation_function. [Accessed 20 August 2019].

[37] Scikit Learn, “Decision Trees,” [Online]. Available: https://scikit-learn.org/stable/modules/tree.html. [Accessed 08 September 2019].

[38] Scikit learn, “Support Vector Machines,” [Online]. Available: https://scikit-learn.org/stable/modules/svm.html. [Accessed 08 September 2019].

[39] Wikipedia, “Artificial neural network,” [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network. [Accessed 3 September 2019].

[40] ResearchGate, “Structure of a memory cell in long short-term memory (LSTM)-RNN.,” [Online]. Available: https://www.researchgate.net/figure/Structure-of-a-memory-cell-in-long-short-term-memory-LSTM-RNN_fig4_318453428. [Accessed 05 September 2019].