RADBOUD UNIVERSITY NIJMEGEN

FACULTY OF SCIENCE

Automated Malware Attribution

USING MACHINE LEARNING ON BINARIES

THESIS BSC COMPUTING SCIENCE

Author:
Bart Hofman
s1018982

Supervisor:
Erik POLL

Second reader:
Alex SERBAN

March 2021

Abstract

The classification (attribution or detection) of malware binaries has an urgent need to be automated. Due to the increase in the amount of malware and the sophistication of this malware, the need for automation is greater than ever. Several attempts have been made at creating machine learning models that can classify binaries. In this thesis a new technique is created that builds upon these previous attempts, learning from its mistakes and using its successes.

Finding a good dataset for this thesis, was a challenge. The dataset that was selected in the end, was created by Boot [11] for his master thesis. It consists of malware samples from 12 different APTs. Even though it was not the ideal dataset, it had sufficient data to perform the experiments needed for this thesis.

The feature selected to train a machine learning model on was the disassembly of a binary. This feature was chosen because of previous work using features that were comparable or contained similar information and that had succes using these features, similar to the partial disassembly used by Haddad Pajouh et al. [9] or the raw bytes used by Raff et al. [8]. The dissembled code was extracted from the binary by means of a custom disassembler plug that disassembled and reformatted the code to try and maximise the context of the disassembled code. The disassembled code is then tokenised and used as the input feature for the model. The model attributes the malware to the APT that created it.

The model used in the thesis is a neural network architecture consisting of 4 layers. It first has an embedding layer followed by a bidirectional LSTM and a dense layer that then connect to the output layer that returns the attribution to an APT expressed in its probability.

The performance of the model was good, having an 86% accuracy and an average F-measure of 0.83. However, the model's performance is not good enough for the model to serve as a sole attributer. The model could play an advisory role in attributing malware. Nevertheless, the model did show promise and can still be expanded upon greatly by unpacking the malware samples and altering the model's design.

The technique used in this thesis shows promise for binary analysis and using the disassembly of a binary showed promising results, however, there is still a lot to be improved upon.

Contents

1 Introduction

With the start of the decade, the amount of malware has increased dramatically [10]. Figure 1 depicts the rapid increase of malware attacks. This large increase in attacks brings new challenges with it, such as the need for fast detection and response, but also finding out where the malware originated from, to hold the perpetrators responsible. Not only is there more malware, but the sophistication of the malware has increased as well. Since 2010 cyber-warfare has become a reality and state-sponsored malware groups (also called APT, Chapter 1.2) and large hacking groups are becoming an ever-greater threat [10][14]. This increasing threat calls for a new approach to managing malware. When normally managing malware on users computers, it consists of 2 steps: malware classification (based on detection) and malware removal. However, in this thesis, we will focus on classification based on attribution to an APT. This means that the thesis will focus on determining who made the malware and not the detection of malware

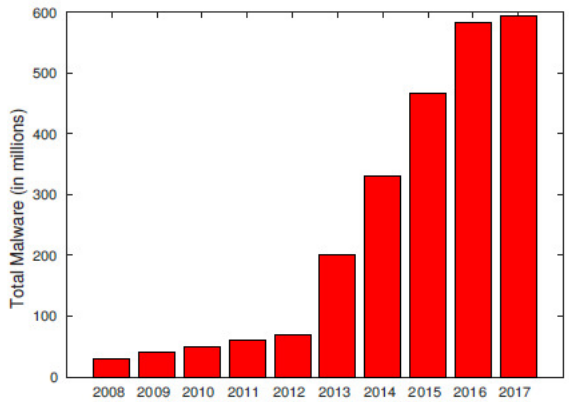


Figure 1: Amount of detected malware attacks since 2008 [10]

In literature, there are currently two methods to analysing malware, dynamic and static methods[1]. In the process of dynamic classification, the malware is executed in a controlled environment, that allows for the analysis of the behaviour and purpose or result of the malware. This method is also called sandboxing.

Static analysis of a binary does not involve executing the binary. The content of the file is analysed as such. Currently, the most used technique of static analysis is fingerprinting[1]. With fingerprinting, the malware binary is characterised by the information that is derived from the file such as compiler flags, the OS that it is written for or some ASCII printable text that is in the binary. This information is used to create a "fingerprint" of the malware. Fingerprints of different files can be compared in order to detect similar binaries e.g. similar malware. Due to the never-ending security game between the creators of malware on one side and the creator of anti-virus software on the other side, this technique has become less effective. Malware has become more obfuscated because of the use of polymorphism and packers to hide its true purpose and circumvent detection mechanisms like fingerprinting [2]. This has made it harder for deterministic techniques and humans themselves to classify these binaries.

To meet this challenge there has been a shift from methods like fingerprinting towards the use of machine learning to classify malware. There have been several attempts to create effective machine learning models to classify malware [8][5][16][12][9].

This thesis will explore the possibilities of a new static analysis technique, building on previous attempts and improving upon them. This new technique will be using the disassembly extracted from the malware binary by means of a disassembler. The data that results from this will be used to train an RNN (Chapter 1.2) network based around an LSTM (Chapter 1.2) to classify the malware based on its author, also called attribution. Attribution differs from detection on only 1 thing, namely: instead of having the binary choice of malware or non-malware, attribution has multiple classes that it can be, one class for every malware creator. Due to time constraints and the lack of a suitable dataset, training the model on detection will be left as future work.

The performance of the model will be evaluated on its F-measure and several other metrics, using a model that was created by another student [15] based on the MalConv model [8] as a benchmark. Furthermore, the model will be evaluated on its usability in real-life scenarios.

1.1 Readers Guide

Chapter 2 This chapter discusses previous attempts at detecting malware using machine learning and the important lessons learnt in this research.

Chapter 3 This chapter gives a quick overview of the strategy used in this thesis and why this strategy was chosen.

Chapter 4 This chapter goes over the needed tools for disassembling binaries and the tools used for creating a model.

Chapter 5 In this chapter, the methods for malware sample collection and modification are discussed, before finally discussing the architecture of the new model and the training and evaluation of this new model.

Chapter 6 This chapter discusses the results obtained from the new model and evaluates its performance.

Chapter 7 This chapter discusses the research that couldn't be done in this thesis and future improvements on this thesis.

Chapter 8 Finally, this chapter discusses the implications of this research and summarize the results.

1.2 Abbreviations used

APT Advanced Persistent Threat

API Application Programming Interface

Bi-LSTM Bidirectional Long Short Term Memory

CNN Convolutional Neural Network

ESN Echo State Network

GRU Gated Recurrent Unit, a form of LSTM with a type of ”forget” gate

LSTM Long Short Term Memory

RNN Recurrent Neural Network

PE header Portable Executable header

ELF header Extensible Linking Format header

Architecture Neural network layer design

Model A completed neural network, with all its layers combined

1.3 Basic concepts

1.3.1 What is a binary

A binary is a file built up of bytes that represents code in such a way that a computer can interpret it and execute it. It consists of several sections of which a few are of importance to this thesis.

header The header of a binary contains basic information about the environment for which the binary is made. For example, if its a 32 bit or a 64 bit executable. Windows executables use a PE and Linux executables use ELF (Section 1.2).

.text The *.text* section of a binary contains all its code.

There are many more sections in a binary however these two are of importance to this thesis. An important fact about how sections are placed within a binary is that they are positioned at an arbitrary offset within the binary.

1.3.2 Stripped binaries

A binary can be stripped. What this means is that the compiler removes all debug symbols from the binary when compiling it. This is done to improve performance but also makes it more difficult for humans to reverse engineer the binary, but has no affect on the actual disassembly process.

2 Prior work on malware detection and attribution

When researching the topic of classifying malware using machine learning, several papers stood out because of the techniques used in them or the issues they encountered. These papers gave useful information as to how to create an effective model and what to take into consideration when doing so.
The most important takeaways of each paper are listed below:

2.1 Raff et al. [8] described a new way of analysing malware files by using the raw bytes of the file. They faced several challenges while doing so, such as large sample sizes and the placements of sections within a binary (Chapter 1.3.1).

2.2 Safar et al. [12] tried several types of machine learning architectures to see which one would be most effective for classifying malware. They concluded that the CNN-LSTM hybrid network performs best. However, a standalone LSTM performed only slightly worse.

2.3 Vinayakumar et al.[7] exposed the benefits of an LSTM over an RNN when trying to classify malware due to the need for correlation over long distances in a sequence. They found that an LSTM has better performances over long sequences.

2.4 Pascanu et al.[5] introduces an interesting feature, based on API calls, that they used to train their model. They furthermore introduced several useful techniques for dealing with long sequence lengths, albeit of a different feature type.

2.5 Feng et al.[16] used a concept called Bi-LSTM to give a model the possibility to learn correlations both going backwards through the input sequence as well as going forwards. They furthermore introduce a variable-length RNN. However, the implementation of this on the model made in this thesis will be left for future work.

2.6 George et al.[3] research showed the importance of managing your dataset when classifying malware. Their model's performance was significantly affected by the underrepresentation of a class within the dataset.

2.7 Haddad Pajouh et al.[9] used an interesting feature selection, using the core instructions of the ARM instruction set at their feature. However, it had some potential weaknesses that could harm the effectiveness of their model.

This selection of papers is far from complete. However, the most important papers with regards to this thesis are listed above. These papers were selected because of the techniques and issues described within them. There are many more papers about malware classification using machine-learning. A lot of them have a recurrent factor, namely that the datasets used for the training of the model are often secret and unattainable for use [8][7][5][16][6].

2.1 MalConv

In 2017 Raff et al. [8] created a new model for classifying malware based on the raw bytes. The model processes an entire binary in one go without any preprocessing of the data. This brought some interesting and relevant challenges with it.

The first challenge they encountered was the random placement of elements within the binary. In a binary the location of sections and the header (PE or ELF) is random (1.2 and 1.3.1). Therefore they decided on a CNN because it has the capacity to find these randomly allocated sections. Furthermore, they use a fully connected layer so the network can process these sections better.

The second challenge they faced was the size of the sample they try to analyse. Malware can vary in size considerably. When analysing a binary directly and all at once, this would cause the memory footprint to be too large to handle and this would lead to drops in performance. Therefore they used a moving window of 500 bytes. This means that they first process the first 500 bytes, then the next 500 and so forth.

A limitation that comes with using the raw bytes when analysing executables is the fact that you are limited to a broad overview of what happens within the binary. If you want a more precise overview of what happens you would have to disassemble the binary. The reason for this is that for a model to be able to completely understand the binary, it would have to learn the context that it is operating in. This is formed by combining several sections from the binary.

Another model was trained, using the same structure as the original paper, but on a different dataset, by Mulder [15]. In his thesis, he trained the model on an attribution dataset [18]. This same dataset will be used to train the model made in this thesis for an accurate comparison.

2.2 Comparing performance of neural networks

The use of machine learning and specifically the use of RNNs when classifying malware was researched by Safar et al. [12]. They used a form of disassembly and applied it to the Microsoft 2015 BIG malware dataset [20] and used the data resulting from this as a dataset to train and compare the performance of several types of architectures. They concluded that for non-hybrid networks, the LSTM network performed best. However, on overall performance, the CNN-LSTM hybrid network performed better.

2.3 RNN vs LSTM

Vinayakumar et al. [7] created a model that could classify Android malware using the Android permissions as a bag of words. They clearly describe the advantages of an LSTM over a traditional RNN. An LSTM has the potential to make links across larger distances of sequences. Because the sequence size used for the training of the model made in this thesis relies on the size of the .text (Chapter 1.3.1) section of the original binary, the model will need to be able to handle links across larger distances in the sequence. This is necessary because the code won't be structured in execution order. So possible associations will have to be made over longer distances.

2.4 Dealing with long sequences

A totally different approach for sequence feature selection was presented by Pascanu et al. [5]. They put forward a way of detecting malware based on the API call sequence of the malware. From the paper itself, it was not clear if this was done in a static or dynamic matter. What was clear from the paper was that the sequence size that resulted from this form of feature extraction was too long for direct input. Their model was based around a standard RNN and an ESN (Chapter 1.2). The memory windows, how long they can 'remember' from earlier in the sequence, for these types of architecture is too small for large sequences. Therefore they applied two techniques called half-frame and max-pooling.

Half-frame is a technique that uses the intermediate states of an RNN as its output instead of its final state and Max-pooling is a downsampling algorithm that can be used as a way to give an abstraction of a previous layer.

A small improvement upon this paper was made by Athiwaratkuen et al. [6] by replacing the standard RNN with an LSTM and a GRU (Chapter 1.2), this improved the model significantly.

Due to the fact that the technique developed in this thesis has even longer sequences of more complex data, namely disassembly instead of API calls, than these paper, it might be interesting to see the effects of applying the techniques used in these papers to the model designed in this thesis.

2.5 Bi-LSTM

An interesting take on the normal LSTM model was done by Feng et al.[16]. They proposed a different approach for detecting malware in .apk files (android applications). The data generation they used has no relation to the one used in this thesis. However, a very powerful technique used by Feng et al. is the bidirectional LSTM or Bi-LSTM (Chapter 1.2).
A Bi-LSTM is particularly useful if you want the model to preserve information from both forward and backwards in the sequence chain. This feature is useful when analysing the disassembly that the model made in this thesis will use because of the fact that it isn't structured in execution order. Another concept introduced by Feng et al. is a variable-length RNN. However, due to time constraints, the implementation of this will be left as future work.

2.6 Unbalanced dataset

George et al.[3] found an interesting pitfall when classifying malware using machine learning. They concluded that their model performed worse due to the way their dataset was constructed. They had a dataset that was extremely unbalanced. This did not give their model a chance to learn the correlations properly. Therefore the performance of the model was harder to evaluate. The dataset used in this thesis is not a balanced dataset either. However, if this is taken into consideration during the evaluation, the effectiveness of the model can still be extrapolated.

2.7 Feature selection

With machine learning, feature selection is very important. Haddad Pajouh et al.[9] used a questionable feature as their input sequence for an RNN. The feature they selected was the core ARM instruction of each instruction in the binary. This gave them a representation of the binary in terms of mov, push, pop and so forth. However, what they failed to take into account is that not all instructions are created equal. The instruction mov is proven to be Turing complete. This means that an entire binary can consist of only the mov instruction.

Given this possibility, the technique created in this thesis uses the argument of each instruction as well, as to give context to the neural network on which it can learn equivalence between instructions in the hope to avoid such possible issues.

3 Strategy

The technique developed in this thesis is based on the idea that a program e.g. binary can be attributed to its creator based on the content of the binary and that a disassembled binary serves this purpose better than the raw binary. If a model has sufficient learning capacity, it would be able to distinguish two binaries from each other based on the patterns that match a specific creator. This idea originated from reading prior research that succeeded in detecting malware using ARM instruction and attributing malware using the raw binary (Section 2.7 and 2.1).

Both of these two features used, ARM instruction and the raw binary, are less complete then the full disassembly would be. This is because of the way a binary is structured. Raff et all [8] used the raw binary. All information that is in the disassembly, is also in the binary. However, it is randomly allocated and out of its proper context due to the way binaries are structured (Section 1.3.1). Disassembling the binary would put the information contained in the binary in proper context and may give a more useful feature. Haddad Pajouh et al. [9] also used the disassembly, seeing it as a good feature to use for training a model. However, they stripped the disassembly instructions of their corresponding arguments thus removing a possibly useful piece of information. Therefore, the idea is to use the full disassembly thus providing the model with the most complete picture of the internals of the binary.

Because of the success of the features used by Raff et al. [8] and Haddad Pajouh et al. [9], it would follow that a feature that is more complete would have more success.

To create a model that can use this feature and utilize it to attribute the malware binary to its creator, first the disassembly is extracted from a binary by a disassembler. Applying this to an entire dataset is challenging but can be done by creating a custom plugin for the disassembler used. Subsequently, the extracted disassembly should be structured in a way such that the model can learn the context of a specific instruction by its neighbouring instructions and its arguments.

The core architecture for the model in this thesis is the LSTM, specifically a bidirectional LSTM. The LSTM architecture has the potential to keep track of important parts of the disassembly sequence while learning to disregard the less important parts (Section 2.2). This is also an advantage considering the large disassembly sequence sizes that can occur.

After construction, each separate element of this technique can be combined to create an affective attirbution system for malware binaries.

4 Binary Ninja and TensorFlow

For a few tasks in this thesis, two third-party software stacks were used. The first is a binary analysis tool called 'Binary Ninja' that has a built-in disassembler that will be utilised. The second is 'TensorFlow', a machine learning framework that has many built-in systems that facilitates the creation of a neural network.

Both these tools are discussed in detail below.

4.1 Binary Ninja

As input sequence for the model developed in Section 5.5, the disassembly of the input binary is used. To get this data, a disassembler is needed that is easily scriptable to enable the creation of a plugin that can create sequences from the entire dataset and it has to be fast in order to do this within a reasonable time.

When considering these requirements, only two major candidates remain. The first candidate is IDA pro. IDA pro is a powerful binary analysis tool that has a built-in disassembler that is scriptable using both c++ and Python3.

The second candidate is Binary Ninja. Binary Ninja is a more lightweight binary analysis tool that also has a built-in disassembler that is scriptable using Python3.
Binary Ninja was chosen, mainly because it has a well-documented API [19] and a good support community. The second advantage of Binary Ninja over IDA pro is the relatively low price.

The university purchased a GUI licence for Binary Ninja. On a personal note, the GUI licence was a lot less practical to work with then a headless version. A headless version would have made the development of the plugin for binary ninja a lot easier.

4.2 Tenserflow-Keras

There are 3 major Python frameworks for machine learning, Pytorch, Keras and MXnet. The most used is Keras. An expanded version of Keras is maintained by Google under the software stack TensorFlow under the name TensorFlow-Keras.

TensorFlow [21] is a framework that facilitates building and training neural networks. It supports memory management for large datasets, GPU support, tokenisers, and many more features. Keras itself already supports a wide range of neural network architectures that can be easily fitted together into a complete model as well as the functions needed to train and evaluate them.

The reason TensorFlow was chosen over MXnet and Pytorch is the ease of use of TensorFlow, especially for relative novices in the field of machine learning. It is well documented and there is a large support community.

5 Process from raw binary to classification

To arrive from a binary to an attribution consists of many steps. Every step that the binary has to go through is discussed in depth in this chapter. Per step, the different options will be discussed and the final choices will be elaborated on. The entire process is visualised in Figure 2.

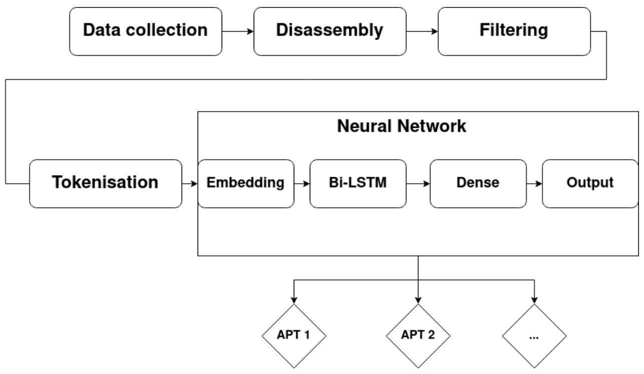


Figure 2: Graph of the steps taken during this thesis

5.1 Data collection

To train the model created in this thesis, training data is needed. This data comes in the form of a dataset. A dataset contains samples of the problem that the model has to classify, in our case malware binaries. As mentioned in Chapter 2, many of the datasets used in prior work are private datasets that were not available. An attribution dataset, a dataset with malware and their creator attached as label, that is suitable for the model to train on was available and contained usable samples.

5.1.1 APT Malware Dataset

This APT (Chapter 1.2) Malware dataset was constructed by Boot [11]. for his master thesis. The dataset consists of a total of 3594 samples from a total of 11 APTs. These APTs again originate from 5 countries. All of the malware samples were requested by Boot from Virus Total [23], a large online database of malware samples. Although this dataset consists solely of malware, not all malware comes in the shape of a binary. Therefore, the non-binary malware files have to be removed. This is done by the disassembler plugin, and will be be explained in Section 5.2. After eliminating the non-binary files a dataset of 2911 binaries remained. However, the final dataset contained 2321 samples. Why the additional 590 flies were removed will be explained in Section 5.3.

5.2 Disassembler

The model requires an disassembly sequence that consists of the disassembled binary. This chapter will go over the creation of the plugin that is used to extract this disassembly sequence from the binary.

5.2.1 Previous attempts

The writing of the plugin took multiple attempts, each improving upon its previous attempt until a stable plugin, that consistently worked, was created. The first attempt was based on the idea that a chronological disassembly structure could be created by using the entry function as a starting point. It would iterate over the disassembly until a call or a jump instruction was found and then recursively jump into those functions to disassemble them and add the instructions to the disassembly structure.

The flaw in this attempt became apparent when the plugin tried to disassemble a stripped binary (Chapter 1.3.1) and it was not able to find the entry point of the binary, thus resulting in an empty disassembly structure.

After this attempt, the second iteration of the plugin took care of this by checking the function table for the entry point. However, the plugin still used the call and jump instructions to create a chronological map.

Although this iteration performed better than the first attempt, there was still a fatal flaw. The plugin did not take into account any conditional jumps or jumps and calls that were obfuscated in such a way they would not be found. Therefore, the chronological map would not be valid and the concept of the chronological maps was abandoned and instead the choice for the use of an enumeration of all functions based on the function table was made.

5.2.2 Proccess management

The disassembly of a single file takes between 5 and 20 seconds. For the entire dataset, this would take approximately 10 hours. Therefore, there was a need for a more speedy process. Multi-threading is the process of having multiple concurrent threads working on the same problem. In this case, the disassembling of binaries. Binary Ninjas disassembler didn't allow multiple async disassemblers to run within the same instance of Binary Ninja. Because of this, an out of the box solution was needed. Multi-threading consists of a few basic components:

Resource When working with multiple threads there is often a shared resource or shared queue of items that need to be processed by the threads. In our case, this is a queue of binary files to be disassembled. This queue is represented in the form of a file, that has the name of a binary file on each line. Popping an item of the queue is done by taking the last line of the file, loading in that file, and deleting the line from the queue file.

Workers The workers are the threads that do the task that is needed, they grab an item from the queue and perform the task on that item and repeat this process until the queue is empty. In this case, the workers are Binary Ninja instances that disassemble a binary.

Master The master is a worker that takes back control of the program after the queue is empty and all threads are finished. In this case, one of the Binary Ninja instances is the master.

Using these components, a multi-threading system can be built without the Binary Ninja instances knowing of each other's existence or any direct communication between them.

However, one problem remains. When two Binary Ninja instances want to grab the next file from the queue at the exact same time, there could be a case of duplicate work. To avoid this mutex locks can be used. However, because the threads are not running in the same environment, they can't have a shared mutex lock. This is solved by creating a '.lock' file. If the '.lock' file exists, it means another thread is currently using the file that the Binary Ninja instance is trying to access. Until the '.lock' file is deleted, the thread will wait.

To keep a overview of all the processes running, a logging system was implemented that registered when a file was processed, and what instance was working or had worked on that file.

Using this system, a successful setup was made of 4 concurrent Binary Ninja instances each running 1 thread. This sped up the processing of the dataset significantly.

5.2.3 Disassembling and Reformatting

The process of disassembling and reformatting the binary can be represented in six core steps.

1. First, the binary is loaded in by the plugin. Binary Ninja then identifies if the file is a binary and if not, discards the file.

2. Binary Ninja then disassembled the file using its internal disassembler.

3. The Binary Ninja internal disassembler hands control back to the plugin which then does the following steps in preparation of the reformatting.

   (a) First, the symbol table is rewritten. The symbol table is a lookup table where function names are mapped to their address. This table contains all the names and addresses of the functions used. From this table, all function names can be extracted and it can then be inverted to function as a lookup table where each address is mapped to a function name.

   (b) Secondly, the functions that were found by the symbol table are checked for their length and based on this, they are selected for in-lining. What in-lining is or why it is done, will be explained later on in the process, where the functions will be inlined.

   Now that all preparations are done, the plugin goes on to the next step.

4. For the reformatting, all functions will have their instructions enumerated. This is done in the following manner:

   (a) All instructions and arguments are enumerated

   (b) When a call or jump is found, the address to which is called or jumped is looked up in the symbol lookup table constructed before and the address is replaced with the function name at that address.

   (c) If a call is made to a function that is selected for inlining, the function will be in-lined right after the call or jump to that function. When inlining a function, the disassembly of this function is taken and is injected into the function that calls it just after the call. This is done with smaller functions to put their disassembly in the correct context where it is used. Thus providing the model more context for identifying what APT created the binary. This is illustrated in Figure 3. Even though the exact effectiveness of inlining hasn't been tested. It can logically only add more information to the model.

(d) Lastly, a check is done over all the disassembly if any memory addresses are left. These will then be replaced by a tag that corresponds to the read-write-execute privileges of the memory address.

5. When this process is done for every function, a sanity check is performed that checks if all functions have been enumerated at least once. This is to avoid missing a function that is selected for in-lining but never called directly.
6. Now that the reformatting of the disassembly is done, it is written to a file with the same name as the binary but with the extension ".decomp".

The result of this process on a "Hello_world" program can be found in Appendix 10.1.

This disassembling of the binaries provides a clear overview of what code is present in the binary. Because the data is now organised, the problem that Raff et al. [8] faced with randomly located sections, is no longer an issue. Furthermore, because the entire disassembly is enumerated, instructions and arguments, the instructions have more context for the model to learn on, which gives the possibility for the model to understand equivalence, unlike the approach used by Haddad Pajouh et al. [9]

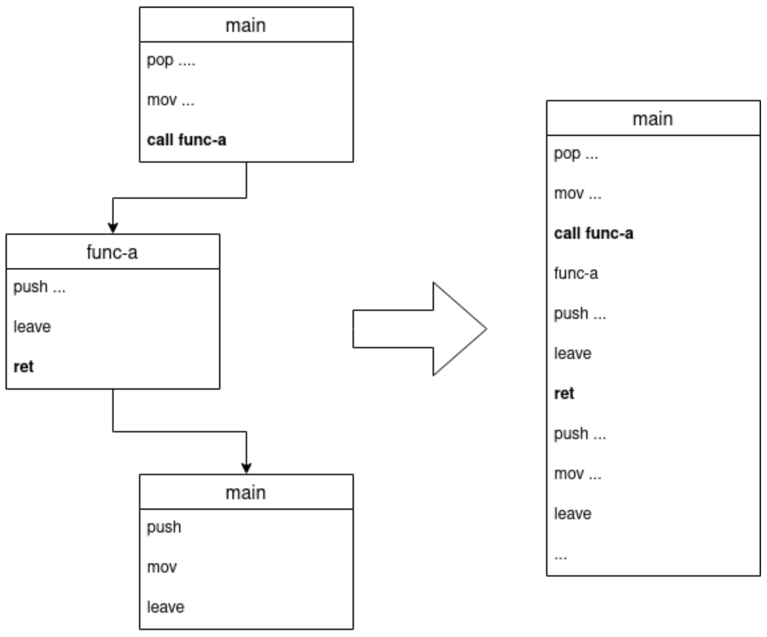


Figure 3: Example of inlining a function

5.3 Filtering

Now that the binaries have been disassembled, another problem has to be confronted, namely the size of the disassembled binares. From Figure 4 it is visible that the spread of malware sizes differs between APTs. Because of this, the dataset is unbalanced. The larger the size of the disassembly, the more data it has for the model to train on. If an APT has small samples, the model won't have as much data to find a pattern in as for an APT that has large samples. Therefore the model can't train as wel on that APT as it can on the APTs with larger samples. With having such an unbalanced dataset comes

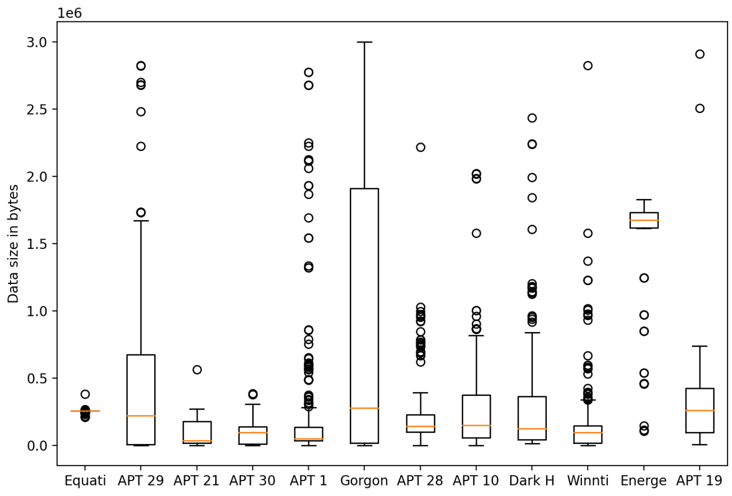


Figure 4: Data size spread per APT

another problem. The largest malware disassembly was 30+ MB, while the smallest was less than 1KB (Figure 4). Because in the next section (Section 5.4) the disassembly sequences are padded to all have a uniform size, the difference between these two would have to be filled up with zeros. This would then increase the overall size of the dataset. If all disassembled malware would stay in the dataset and all other samples would be padded to fit the size of the largest data file. The size of the dataset would then be 49 GB. The dataset would not only be impractical to work with, it would also mainly consist of padding rather than actual data. To solve this, large samples can be excluded from the dataset. Therefore the size limit of a data files was set to a maximum size of 0.5 MB. The largest downside of this filtering was the loss of an entire class. The APT called "Energetic Bear" has its mean at 1.6 MB (Figure 4) and would be largely excluded from the dataset after the filtering for size. Because of this, it was decided that the entire APT "Energetic Bear" would be excluded from the dataset. This filtering brought the size of the dataset down to 658 MB after tokenisation (Section 5.4). A dataset of this size is easy to manage and faster to load into memory.

5.4 Tokenisation

A machine learning model does not understand language and text as a human does, it needs to be able to perform computations on its input. Because of this, the input of the model has to be numerical. This is were a tokenizer comes in. A tokenizer replaces the instructions and arguments in the disassembly from the binary with a continuous numerical representation. A model can then use these numerical representations to create an internal fingerprint for an APT without having to know the meaning of the text.

A tokeniser needs to create the mapping from instructions and arguments to their numerical representation. The process of creating this map is called fitting. When fitting the tokenizer, it is important that it only uses the training data to fit the tokenizer. This way, you can make sure that the model is completely unaware of the test data.

The tokeniser used in this thesis was limited to a maximum of 100 words, this means that only the 100 most frequently used instructions and arguments get a unique numerical representation. All other instructions and arguments get represented by an "OOV", or out of vocabulary token. This is done to focus the model on the core differences of the binary and not get stuck on potentially useless data.

5.4.1 Datastorage

Disassembling (Section 5.2), filtering (Section 5.3) and tokenising (Section 5.4) the entire dataset is a time consuming process. Ideally this only has to be done once. Therefore, after the data processes and is ready for use, it is saved to JSON files for easy retrieval by a generator, to be later described in Section 5.5.2.

5.5 Neural Network

5.5.1 Model design

After researching several model designs from prior work (Chapter 2), the choice was made to use a bidirectional LSTM as the core of the model created in this thesis. The reason for this choice was mainly because the disassembly is not in execution order. The bidirectional LSTM analyses its input sequence from front to back, but also back to front. This gives the model a better chance of learning the differences between APTs in the bigger picture. However, before the input data reaches the bidirectional LSTM it first goes through an embedding layer. An embedding layer simplifies the language used inside the neural network. It compresses large sequences into smaller sequences based on objects that are found in close proximity and represent this as one object. After the embedding layer and the bidirectional LSTM, the model has two dense layers. The first dense layer is structured such that the model can effectively learn correlations through the output of the bidirectional LSTM. The final layer of the model has as many nodes as there are APTs to be learned on and is its output layer.

A lot of the measures to deal with long sequences, as described by Pascanu et al. [5] were not implemented into the network due to a lack of time and will be left as future work.

The next few sections will be discussing the training of the model and all requirements that come with this. When speaking of the model in these chapters, it is regarding this model.

5.5.2 Input pipeline

For a model to train, it needs training data. There are two methods of delivering this data to the model. The first method would be to load the entire dataset into memory at once and feed it to the model as one big data blob. However, due to the large sequence size and the large model size that comes with this, the memory footprint would be too large. Therefore, a second method exists that utilises generators.

A generator is a function that, each time you call it, returns the next object from a given list. The dataset that is used to train the model, is saved in JSON files, as described before in Section 5.4.1. Therefore, the generator only has to open and read the next JSON file in the directory every time it needs to provide a new sample.This way, the minimal amount of data is in memory at any time.

5.5.3 TensorFlow dataset

Besides using generators, there is another thing that can be done to optimise data management. TensorFlow has a useful built-in tool for creating a lightweight object that utilises a generator and can create circular shuffled datasets with this. A circular shuffled dataset is a self repeating dataset that comes in a randomly shuffled order each time, this is done to improve the training process. Tensorflow-datasets are optimised to work with the rest of the TensorFlow software stack including Keras and GPU accelerated learning. This comes in useful because of the large network size, the TensorFlow-dataset optimises the memory usage of the GPU so this doesn't have to be reimplemented per model.

5.5.4 Training the model

Now that the training data and the model are ready, the model can be trained. Initially, the model was overfitted on a smaller dataset with only two classes to tweak the model's parameters like layer sizes and activation functions. When the model performed to satisfaction, the model was trained on the entire dataset. When training a model, you don't know in advance how many epochs, where an epoch is the time it takes for the model to go over the entire dataset once, it takes for the model to reach its optimal performance. Therefore, a mechanism called early-stopping is implemented. Early-stopping checks performance increases per epoch and decides to stop the training when a delta change hasn't been reached. This was after 26 epochs for the model trained on the APT dataset.

5.5.5 Model evaluation

A machine learning model can be evaluated in many ways. The most common metric for performance is the accuracy metric. However, because of the unbalanced dataset that is used in this thesis, this metric would be unreliable. The reason for this is that the accuracy score is based on the proportion of correctly predicted items. Take the following example:
If a dataset contains 90% of sample A and 5% of sample B and 5% of sample C and the model would identify 90% of A correctly and can't distinguish B and C from one another, the model would still have a 90% accuracy even though it is not an effective model for its task. This would also be the case with our dataset if we were to use the accuracy as sole metric for performance as is visible from Figure 5.

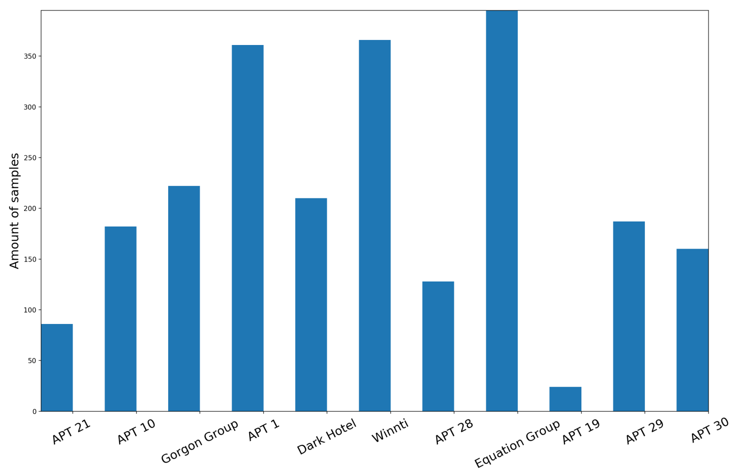


Figure 5: Dataset sample count per APT

To still evaluate the model and get a measure of its performance, the following set of metrics were used.

Recall What proportion of malware that was classified to be the correct APT in proportion to all that APT's samples.

Precision What proportion of malware that was classified to be of a given APT was actually made by this APT. This gives a good indication of if the model can distinguish between classes.

Accuracy The proportion of correctly predicted items.

F-measure The F-measure, also called the F1-score, is the harmonic mean of both the recall and the precision. The F-measure is a good metric when a combination of both recall and precision is needed.

Due to the goal of this model, namely attributing a given piece of malware, both the recall and precision of the model are of importance. In a real-life scenario, if the model would incorrectly predict the creator given a piece of malware and this results in wrongfully placing the blame on an innocent party. It could cause a diplomatic incident.

6 Discussion of results

6.1 Sample size matters

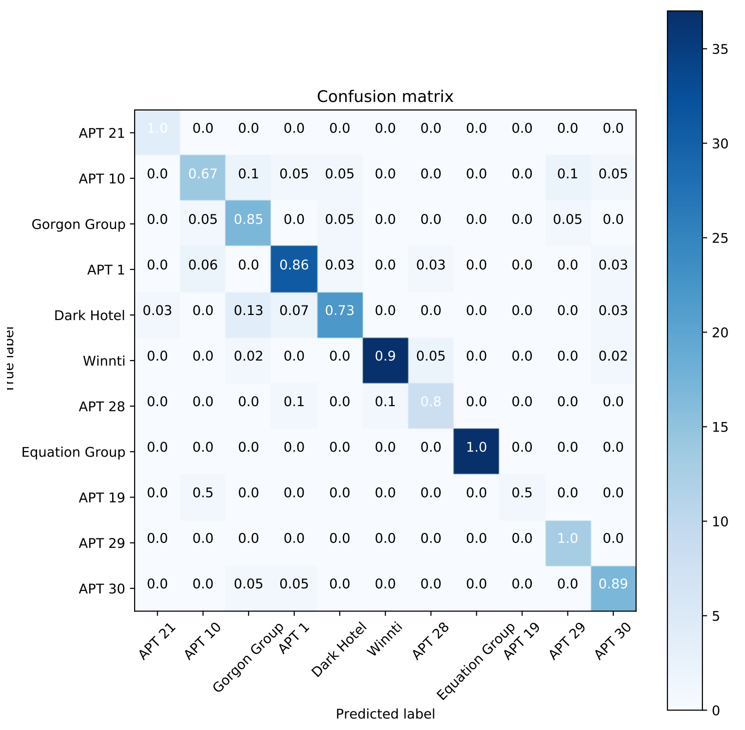
From Figure 4 it was clear that one APT its samples, namely ”Energetic Bear”, were nearly all the same size. This was the reason it was removed from the dataset used to train the model. However, this near-constant size creates the possibility to manually attribute malware coming from this APT, thus negating the need for sophisticated techniques.

6.2 Model performance

As described in Chapter 5.5.5 the model will be evaluated using four main metrics, namely recall, precision, accuracy and F-measure. The most important metric will be the F-measure. This is because of the importance of both recall and precision and F-measure being the harmonic mean of them gives an impression of the overall performance.

All of the metrics that are used can be calculated using a confusion matrix of the correct APT and the predicted APT given a malware sample as seen in Figure 6.

Figure 6: Confusion matrix for all APTs



	APT 21	APT 10	Gorgon Group	APT 1	Dark Hotel	Winnti	APT 28	Equation Group	APT 19	APT 29	APT 30
APT 21	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
APT 10	0.0	0.67	0.1	0.05	0.05	0.0	0.0	0.0	0.0	0.1	0.05
Gorgon Group	0.0	0.05	0.85	0.0	0.05	0.0	0.0	0.0	0.0	0.05	0.0
APT 1	0.0	0.06	0.0	0.86	0.03	0.0	0.03	0.0	0.0	0.0	0.03
Dark Hotel	0.03	0.0	0.13	0.07	0.73	0.0	0.0	0.0	0.0	0.0	0.03
Winnti	0.0	0.0	0.02	0.0	0.0	0.9	0.05	0.0	0.0	0.0	0.02
APT 28	0.0	0.0	0.0	0.1	0.0	0.1	0.8	0.0	0.0	0.0	0.0
Equation Group	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
APT 19	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0
APT 29	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
APT 30	0.0	0.0	0.05	0.05	0.0	0.0	0.0	0.0	0.0	0.0	0.89

Table 1: Evaluation scores per APT

Evaluation metric	APT 21	APT 10	Gorgon Group	APT 1	Dark Hotel	Winnti
Recall	1.00	0.67	0.85	0.86	0.73	0.90
Precision	0.80	0.78	0.68	0.86	0.88	0.97
F-measure	0.89	0.72	0.76	0.86	0.80	0.94
Train samples	82	161	202	325	180	325
Test samples	4	21	20	36	30	41

Evaluation metric	APT 28	Equation Group	APT 19	APT 29	APT 30
Recall	0.80	1.00	0.50	1.00	0.89
Precision	0.73	1.00	1.00	0.81	0.81
F-measure	0.76	1.00	0.67	0.90	0.85
Train samples	118	358	22	174	141
Test samples	10	37	2	13	19

Table 2: Evaluation score average from all APTs

Metric	Value
Recall	0.84
Precision	0.85
F-measure	0.83
Accuracy	86%

6.3 Context of the results

6.3.1 Individual APTs evaluation

To determine how the model performed, a good interpretation of the metrics, with regards to malware attribution, is needed. What do recall and precision mean in practice with regards to malware attribution? When looking at malware, if an APT has a recall of 0.6, then 40% of that APTs malware were attributed to another APT. A precision of 0.6 means that 40% of the malware attributed to this APT were not made by that APT. Using this, the results from Table 1 can be properly interpreted.

Overall, the network performs good, having an F-measure of 0.83 and an accuracy of 86% (Figure 6 and Table 2). However, there are some interesting cases to look at.

1. The first APT 10 has 10% of its samples (2 samples) attributed the Gorgon group and 10% of its samples attributed to APT 29 (Figure 6 and Table 1). the Gorgon Group and APT 29 are both affiliated with different countries than APT 10 (Table 5). APT 29 does originate from a neighbouring country of APT 10. This might indicate resource sharing within regions or more likely resource sharing between historic allies. However, since the amount of data that is misattributed to these 2 APTs is 4 samples in total and 3 more samples were misattributed to 3 other APTs, no real conclusions can be drawn from this. It is more likely that these samples were misclassified based on some noise in the data.

2. The next interesting case that comes forward is APT 1. The model performs well on samples from this APT. Only 5 of its 36 were misattributed. Given the size of the training data (Table 1), the model managed to get a relatively good grip on this APT. The 5 samples that were not attributed correctly were most likely misattributed due to noise in the samples.

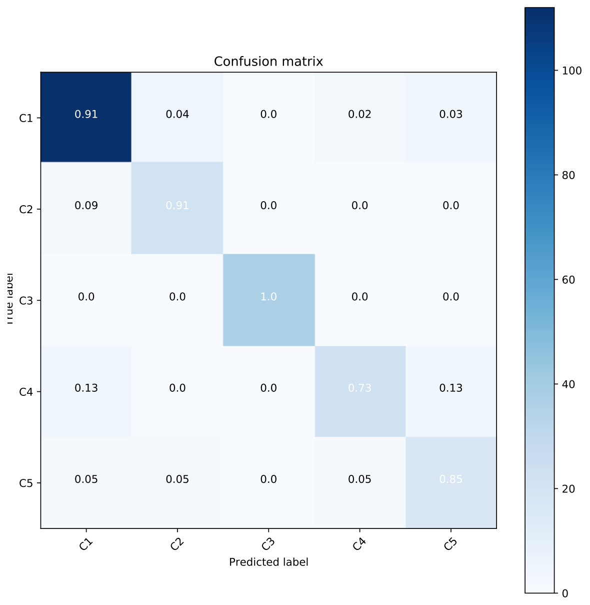


Figure 7: Confusion matrix of APTs per country

Table 3: Evaluation scores per country

Evaluation metric	C1	C2	C3	C4	C5
Recall	0.91	0.91	1.00	0.73	0.85
Precision	0.94	0.78	1.00	0.88	0.68
F-measure	0.93	0.84	1.00	0.80	0.76

3. When looking at Dark Hotel, the first thing that comes forward is the fact that 13% of its samples (4 samples) were attributed to the Gorgon Group. Even though these are only a few samples, it is interesting that they were all misattributed to one specific APT. This could indicate that these APTs share resources or reuse each other's malware.

4. The model performed really well on the Equation Group (Figure 6). It managed to attribute all of its samples correctly. This could indicate that this APT has a specific style or type of malware that it uses that is not used by any other APT.

5. One APT that didn't perform well at all was APT 19 (Figure 6). This can be attributed to a lack of data. APT 19 only had 22 train samples and 2 test samples (Table 2).

The amount of data used to train and test the model is not enough to make any conclusions on what caused possible misattribution if it was noise or any other correlation between APTs. Nevertheless, the model can attribute several classes relatively consistently. There are some factors which can explain this.

1. The first factor that might explain the performance is the fact that the malware samples are not all written in the same programming language. Each programming language compiles differently. Therefore, if an APT is more prone to have samples that are of a specific programming language and this language is not used by other APTs, then this can be a clear identifier for this APT.

Table 4: Evaluation scores based on associated country

Evaluation metric	Value
Recall	0.88
Precision	0.85
F-Measure	0.86
Accuracy	90%

Table 5: Countries with their affiliated APTs

Country	Affiliated APTs
C1	APT 1, APT 10, APT 19, APT 21, APT 30, Winnti
C2	APT 28, APT 29
C3	Equation group
C4	Dark Hotel
C5	Gorgon Group

2. The second factor that comes into play is the coding styles of each group. Even though after compiling, variable names and other personalizing features are stripped, programming styles are still visible. These programming styles can be used as an identifying feature of a group or individual person. This was also corroborated by Caliskan Islam et al.[4]. Therefore APTs can be identified by their respective programming styles.

Nevertheless, there are also some factors that have not yet been mentioned that could have had a negative effect on performance.

1. As mentioned in Chapter 1, packers are used to obfuscate malware. Packers have a relatively small amount of disassembly. Therefore, there is less data to identify the APT with. A way to identify if this could be an actual performance hindrance would be to have a dataset consisting of confirmed non packed malware samples and compare its performance to the performance on the APT dataset (Section 5.1). A way to combat packers is by combining the approach used in this thesis with a dynamic analysis approach that will be further elaborated on in future work (Chapter 7)

2. Another possible performance hindrance is polymorphism. If the malware was altered by a polymorphic engine, the model might not be able to link it to similar malware due to its altered structure. To confirm that this is an actual problem, a test has to be done with a non-altered dataset and a dataset that has been altered by a polymorphic engine. This type of obfuscation can most likely not be solved by using the disassembly as a feature, however using API calls could be useful in combating this, like is done by Pascanu et al.[5].

Resource sharing and malware reuse are all speculative. What can be concluded from this model is that it can not yet be relied upon as a sole attributer. What it can be used for, is an advisory role. Its overall performance as can be seen from Table 2 is relatively good given the dataset it was trained on. However, due to the sensitive nature of malware attribution, the model doesn't perform good enough to be seen as a reliable attributer. When asking the model to attribute a piece of malware, it would be more useful to look at the top 3 highest candidates that the model puts forward instead of only looking at the highest. The model then gives a human a prime candiate and what other candidates there are.

Table 6: C1 APTs compared

APT	Recall	Precision	F-Measure
c1-1	0.86	0.86	0.86
c1-2	0.67	0.78	0.72
c1-3	0.50	1.00	0.67
c1-4	1.00	0.80	0.89
c1-5	0.89	0.81	0.85
c1-6	0.90	0.97	0.94

The dataset that was used to train and test the model was unbalanced (Section 5.5.5). Because of this, some APTs had more test and train samples than others. For the APTs that have relatively few test samples, the results are less easy to interpret because the probability that the model got lucky is higher.

6.3.2 Country evaluation

From Figure 7 it is visible that most of the APTs in the APT dataset originate from C1. The APTs affiliated with C1 have some variation in performance (Table 6) taking into account that some of these had a very limited amount of test samples, this could be attributed to potential resource sharing between these APTs, but this conjecture isn't supported by the other results (Figure 6). Nevertheless, due to the secrecy surrounding these APTs, it is unlikely this hypothesis can be tested.

The overall F-measure of the model when looking at the country data is 0.86 (Section 6.3.1). However, when looking at the performance hindrances, another hindrance might play a role, namely false flag attacks. A false flag attack is an attack meant to misdirect the victim towards another country or organisation [17]. The possible inclusion of false flag attacks in the dataset could have a negative effect on the performance of the model. Confirming that this is an actual performance hindrance is problematic due to the fact that there is no exact data available.

6.4 Strategy Evaluation

Overall the strategy discussed in Chapter 3 performed well. Getting all the parts working took quite some time. However, once everything was working it was stable and a good system to work with. The speed-boost that the disassembler got by using multithreading (Section 5.2) helped a lot in speeding up the process.

6.5 Benchmark with previous work

As stated before, accuracy is not a good metric for looking at the performance of a model trained on an unbalanced dataset. However, Mulders [15] used different performance metrics, the only overlapping metric was the accuracy, therefore it is used for comparing the performances of the model developed in this thesis and the model developed by Mulders.

In previously done experiments on the APT dataset by Mulders [15] the accuracy for the prediction per class was 85%. From the results of the new model, it is visible that the accuracy is only slightly better at 86% (Table 2). However, the model used by Mulders wasn't limited to only binaries and thus had a larger range of samples at its disposal. This could be a good argument to support the method developed by Raff et al. [8].

When looking at the per country performance, there is a similar increase in 1% with regards to accuracy, from 89% to 90% (Table 4). From these results, the effect of using disassembly seems marginal. However, there are still a lot of improvements that can be made upon the model as will be discussed in the next Chapter (Chapter 7).

7 Future work

7.1 Packers

This new technique has not been tested to its full extent because of a technique used by malware authors called packing. Packing is the practice of compressing the malware executable code and encrypting it. The only thing the disassembler plugin will find is the decompressing and decrypting code. The decompressing an decrypting code is not useless for all cases. For example, it can still be useful when trying to identify the APT that wrote the malware because different APTs use different methods of packing. A promising way of getting around packers is by doing a memory dump of the program after its first system call. This is usually a good indication that the malware is unpacked and can then be dumped and decompiled.

7.2 Dataflow graphs

This thesis focused mainly on the code of the malware samples, however, a good addition to the code would be data flow graphs. Data flow graphs were used by C. Meijer at al.[13] to find crypto algorithms in binaries. Data flow graphs can also be useful to identify malware authors because they are affected by the way the program is written.

7.3 Network

The LSTM architecture was not the best performing architecture in the experiments performed by Safa et al. [12]. The CNN-LSTM hybrid network performed best. Using a CNN-LSTM based model could improve performance. However, the model in its current state is very minimalistic. Using only one bidirectional LSTM layer. Adding more LSTM layers might improve performance also. The reason the CNN-LSTM wasn't implemented initially was due to my personal inexperience with machine learning and the framework TensorFlow.

7.4 Max-pooling and Half-frame

Pascanu et al. used two interesting techniques that can be useful when analysing long sequences. They used half-frame and max-pooling.

Half-frame is a technique that uses the intermediate states of an RNN as its output instead of its final state and max-pooling is a downsampling algorithm that can be used as a way to give an abstraction of a previous layer.

Applying these techniques could speed up the model and increase its performance.

7.5 Variable length RNN

Feng et al. [16] used a variable-length RNN. Applying this to the model developed in this thesis increases the ability of the network to process larger malware samples. In Section 5.3 many samples had to be removed from the dataset because of their size. Using a variable RNN would enable the model to also process these samples without having to pad the other samples to their size and thus keeping the dataset small.

7.6 Inlining

During this process of disassembling and reformatting (Section 5.2), the functions are inlined. This was done based on the assumption it would give the model more context to learn on. However, to confirm this assumption, a study should be performed comparing the performance of disassembly that has inlined functions to the performance of a model that does not have inlined functions.

7.7 Performance on a raw binary

In this thesis, the network is used to process a disassembled binary. However, it would be interesting to see the performance of a similar network structure but directly trained on the raw binary as done by Raff et al. [8]. To get an accurate comparison of the models' performances, both should first go through several of the future work steps noted above (Section 7.3 and 7.4)

7.8 Malware detection

The technique for malware analysis in this thesis is applied to malware attribution. However, it would be interesting to see the effectiveness of this same technique applied to malware detection. An important factor to keep in mind when doing so is that the malware should be unpacked first, how this is done is described in Section 7.1. This is important to make sure the model trains on malware and not on packers.

8 Conclusion

8.1 Data acquisition

During this thesis, there was a recurring problem with acquiring the datasets used by other researchers due to those datasets being privately owned by companies. This was solved by obtaining access to one open-source dataset that was used in another thesis [15].

8.2 Strategy evaluation

8.2.1 Disassembly plugin evaluation

The process of extracting the disassembly was a process that needed a lot of attempts before it was suitable for use. The effectiveness of using disassembly as a feature isn't yet fully clear. Therefore the same model used in this thesis should be trained on raw binaries as well, as described in Section 7.7. What can be concluded is that the plugin as it is structured now is a fast way of disassembling a large dataset and the time it takes to disassemble one sample for real-life attribution is negligible.

8.2.2 Model evaluation

From the results discussed in Section 6 it can be concluded that the model has potential. Its high F-measure of 0.83 and an accuracy of 86% show that the model can learn to attribute APTs. However, the model doesn't perform well enough to be used as a sole attributer. The model's performance could significantly improve given a better dataset.

When comparing the technique developed in this thesis, it performs slightly better than the technique developed by Raff et al. [8] and adapted by Mulders [15]. However, the technique developed in this thesis has some drawbacks. The technique is limited by malware in the form of a binary. Because the technique is based on the disassembly of that binary. The technique developed by Raff et al. doesn't have this drawback. However, the technique developed in this thesis does have more possibilities to expand on its analysis of the binary by unpacking packed binaries and creating a more complicated network structure as described in Chapter 7.

8.3 Possible real life applications

Currently, the performance of the model is not good enough to be used as a sole attributer. However, Chapter 6 shows that the model performs well enough to be used in an advisory capacity.

The technique as developed in this thesis would be suitable for creating an application, however, a few issues have to be addressed first.

1. The plugin needs to be rewritten for the Binary Ninja headless version such that it can be integrated into a bigger piece of software.
2. The input pipeline to the model would have to be reformed to handle single files.

8.4 Personal lessons learned

During this thesis, I came across a lot of problems that I had not foreseen. If I or any other student would have to continue this research or use aspects of it, here are the most important takeaways of the process.

8.4.1 Expectactions

Being a cyber-security student and more akin to low lever cyber-security, it is a big change to look at machine learning. When I started with the subject I had no idea what things would work and what would not. Therefore I decided to use something I did know, namely binary analysis and disassembly in particular. When reading the prior works like the work from Raff et al. [8] and Haddad Pajouh et al. [9] I came across features and techniques used for malware classification using machine learning that, in my experience with malware analysis, should not be valid features. After discussing with friends and security experts like C. Meijer, I concluded for myself that these features were indeed ambiguous. However, looking at the results that the models trained on these features achieved, I was surprised by how well they performed. Another conclusion that I came to when reasoning about these features, is that the features that should not work for detecting malware, can work for attributing malware.

8.4.2 Disassembler

When I started with the disassembly phase of the thesis. I had the idea to integrate the disassembler in the input pipeline of the actual model. However, the university bought the GUI licence for Binary Ninja. Having to use the GUI over the headless version was a huge time waste and I would advise anyone using the same strategy to buy the headless version. It would make the disassembly process more streamlined and would avoid the detour that had to be made to multithread and speedup this process.

8.4.3 Time lapse

The most time in this research I spent on developing and getting the model working. Creating the Binary Ninja plugin was not a quick process, however, it was a familiar subject to me and I knew how everything interacted. When I started creating the model, I had no prior experience with machine learning or TensorFlow in general. This inexperience caused me to make some stupid mistakes and was the cause of a lot of double work. My advice to future researchers that don't have experience with machine learning would be to get familiar with the existing frameworks before you start properly experimenting with them.

8.4.4 General remarks

Even though this thesis wasn't easy for me to complete, I am convinced that machine-learning will play a key role in future cyber-security and malware analysis.

Lastly, the disassembled APT dataset and model used with all associated code can be found in a public repository on Github [22].

9 Bibliography

References

[1] N. Idika and A. Mathur. *A survey of malware detection techniques*. Tech. rep. Purdue University, 2007.

[2] Y. Ilsun and Y. Kangbin. “Malware Obfuscation Techniques: A Brief Survey”. In: *Proceedings - 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010* (2010), pp. 297–300.

[3] L. Deng G. E. Dahl J. W. Stokes and D. Yu. “Large-scale malware classification using random projections and neural networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 3422–3426.

[4] A. Caliskan Islam, F. Yamaguchi, E. Dauber, R. E. Harang, K. Rieck, R. Greenstadt and A. Narayanan. “When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries”. In: *CoRR* abs/1512.08546 (2015).

[5] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu and A. Thomas. “Malware classification with recurrent networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2015), pp. 1916–1920.

[6] B. Athiwaratkun and J. W. Stokes. “Malware classification with LSTM and GRU language models and a character-level CNN”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2017), pp. 2482–2486.

[7] R. Vinayakumar, K. P. Soman and P. Poornachandran. “Deep android malware detection and classification”. In: *International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (2017), pp. 1677–1683.

[8] E. Raff, J. Baker, J. Sylvester, R. Brandom, B. Catanzaro and C. Nicholas. “Malware Detection by Eating a Whole EXE”. In: *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[9] H. Haddad Pajouh, A. Dehghantanha, R. Khayami, K Kwang and R. Choo. “A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting”. In: *Future Generation Computer Systems* 85 (2018), pp. 88–96.

[10] A. Sahu, P. Parwar and D. Agrawal. “An Analysis to Detect Malware using Machine Learning”. In: *ijsart Volume 5* (2019).

[11] C. Boot. “Applying Supervised Learning on Malware Authorship Attribution”. Master Thesis. Radboud University, 2019.

[12] H. Safa, M. Nassar and W. A. Rahal Al Orabi. “Benchmarking Convolutional and Recurrent Neural Networks for Malware Classification”. In: *2019 15th International Wireless Communications Mobile Computing Conference (IWCMC)* (2019), pp. 561–566.

[13] V. Moonsamy C. Meijer and J. Wetzels. “Where’s Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code”. In: *Not yet published, arXiv:2009.04274* (2020).

[14] K. van Teeffelen. “Universiteit Maastricht over de hack: we konden niet anders dan betalen”. In: *Trouw* (2020).

[15] W. Mulder. “Where does this malware comefrom?” Bachelor Thesis. Radboud University, 2020.

[16] R. Feng, J. Q. Lim, S. Chen, S. Lin and Y. Liu. “SeqMobile: A Sequence Based Efficient Android Malware Detection System Using RNN on Mobile Devices”. In: *Not yet published, arXiv:2011.05218* (2020).

[17] F. Skopik and T. Pahi. “Under false flag: using technical artifacts for cyber attack attribution”. In: *Cybersecurity, Springeropen* (2020).

[18] *APT dataset.* URL: `https://github.com/cyber-research/APTMalware`.

[19] *Binary Ninja Api Documentation.* URL: `https://api.binary.ninja/`.

[20] *Microsoft 2015 BIG dataset for malware classficiation.* URL: `https://www.kaggle.com/c/malware-classification/data`.

[21] *Tensorflow.* URL: `https://www.tensorflow.org/`.

[22] *Thesis repository containing the dataset and code.* URL: `https://github.com/BluePython339/BSCThesis`.

[23] *Virus total.* URL: `https://virustotal.com`.

10 Appendix

10.1 Hello world disassembled

The output of a simple hello world program in the language C disassembled by the plugin described in this thesis.

```
sub_401000
push 0xffffffff
push sub_401d6b
mov eax [ fs : 0x0 ]
push eax
mov [ fs : 0x0 ] esp
push ecx
mov eax [ rw-data ]
test eax eax
jne r-xdata
mov ecx [ esp + 0x4 ]
mov [ fs : 0x0 ] ecx
add esp 0x10
retn
push 0x620
call ??2@YAPAXI@Z
add esp 0x4
mov [ esp ] eax
test eax eax
mov [ esp + 0xc ] 0x0
je r-xdata
xor eax eax
mov [ rw-data ] eax
mov ecx eax
call sub_401070
mov [ rw-data ] eax
mov ecx [ esp + 0x4 ]
mov [ fs : 0x0 ] ecx
add esp 0x10
retn
sub_401070
mov edx ecx
push esi
push edi
mov ecx 0x41
lea esi [ edx + 0x8 ]
xor eax eax
mov edi esi
mov [ edx + 0x4 ] 0x0
[ edi ]
mov ecx 0x41
lea edi [ esi + 0x104 ]
[ edi ]
mov ecx 0x41
lea edi [ esi + 0x208 ]
[ edi ]
```

```
sub_4016a0
push ebx
push ebp
push esi
push edi
mov esi ecx
call sub_4019f0
mov edi [ esi + 0x4 ]
push rw-data
push rw-data
push edi
call [ FindResourceA@IAT ]
mov ebx eax
push ebx
push edi
call [ LoadResource@IAT ]
push eax
call [ LockResource@IAT ]
push ebx
push 0x0
mov ebp eax
call [ SizeofResource@IAT ]
mov edi eax
test edi edi
jle r-xdata
pop edi
pop esi
pop ebp
pop ebx
retn 0x4
mov eax [ esp + 0x14 ]
push edi
push ebp
push eax
mov ecx esi
call sub_401620
push 0x0
push 0x80
push 0x2
push 0x0
push 0x0
add esi 0x51c
push 0x40000000
push esi
mov ebx eax
call [ CreateFileA@IAT ]
```

```
mov ecx 0x41
lea edi [ edx + 0x51c ]
mov [ edx ] r--data
[ edi ]
mov ecx 0x41
lea edi [ edx + 0x314 ]
[ edi ]
mov ecx 0x41
lea edi [ edx + 0x418 ]
[ edi ]
pop edi
mov eax edx
pop esi
retn
sub_401100
mov eax [ esp + 0x4 ]
mov edx [ esp + 0x8 ]
mov [ ecx + 0x4 ] eax
lea eax [ ecx + 0x314 ]
push esi
mov esi 0x104
sub edx eax
test esi esi
jne r-xdata
mov [ eax ] cl
inc eax
dec esi
jne r-xdata
mov [ eax ] 0x0
pop esi
retn 0x8
mov [ eax - 0x1 ] 0x0
pop esi
retn 0x8
mov cl [ edx + eax ]
test cl cl
je r-xdata
mov [ eax - 0x1 ] 0x0
pop esi
retn 0x8
sub_401180
push ecx
push ebx
push ebp
push esi
push edi
mov edi ecx
call sub_4018b0
mov ebx [ edi + 0x4 ]
push rw-data
push rw-data
push ebx
```

```
lea ecx [ esp + 0x14 ]
push 0x0
push ecx
mov esi eax
push edi
push ebx
push esi
call [ WriteFile@IAT ]
push esi
call [ CloseHandle@IAT ]
sub_401730
push ebp
mov ebp esp
push 0xffffffff
push sub_401d80
mov eax [ fs : 0x0 ]
push eax
mov [ fs : 0x0 ] esp
sub esp 0x350
push ebx
push esi
push edi
xor edx edx
mov ecx 0xe
xor eax eax
lea edi [ ebp - 0x4c ]
mov [ ebp - 0x50 ] edx
[ edi ]
mov ecx 0x40
lea edi [ ebp - 0x257 ]
mov [ ebp - 0x258 ] dl
mov [ ebp - 0x35c ] dl
[ edi ]
stosw [ edi ]
stosb [ edi ]
mov ecx 0x40
xor eax eax
lea edi [ ebp - 0x35b ]
mov [ ebp - 0x154 ] dl
[ edi ]
stosw [ edi ]
stosb [ edi ]
mov ecx 0x40
xor eax eax
lea edi [ ebp - 0x153 ]
mov [ ebp - 0x10 ] esp
[ edi ]
stosw [ edi ]
stosb [ edi ]
lea eax [ ebp - 0x14 ]
push r--data
push eax
```

```
call [ FindResourceA@IAT ]
mov esi eax
test esi esi
je r-xdata
pop edi
pop esi
pop ebp
pop ebx
pop ecx
retn
push esi
push ebx
call [ LoadResource@IAT ]
test eax eax
je r-xdata
push eax
call [ LockResource@IAT ]
mov ebp eax
test ebp ebp
je r-xdata
push esi
push 0x0
call [ SizeofResource@IAT ]
mov ebx eax
test ebx ebx
jle r-xdata
push 0x0
push 0x80
push 0x2
push 0x0
push 0x0
add edi 0x10c
push 0x40000000
push edi
call [ CreateFileA@IAT ]
mov esi eax
lea eax [ esp + 0x10 ]
push 0x0
push eax
push ebx
push ebp
push esi
call [ WriteFile@IAT ]
push esi
call [ CloseHandle@IAT ]
sub_401210
sub esp 0xc
push ebx
mov ebx [ CreateDirectoryA@IAT ]
push ebp
mov ebp [ GetLastError@IAT ]
push esi
```

```
mov [ ebp - 0x4 ] edx
mov [ ebp - 0x14 ] 0x6f
call _CxxThrowException
sub_4018b0
sub esp 0x208
push ebp
push esi
mov ebp ecx
push edi
mov ecx 0x40
xor eax eax
lea edi [ esp + 0xd ]
mov [ esp + 0xc ] 0x0
[ edi ]
stosw [ edi ]
stosb [ edi ]
lea eax [ esp + 0xc ]
push 0x104
push eax
push 0x0
call [ GetModuleFileNameA@IAT ]
lea eax [ ebp + 0x8 ]
lea edx [ esp + 0xc ]
mov esi 0x104
sub edx eax
test esi esi
jne r-xdata
mov [ eax ] cl
inc eax
dec esi
jne r-xdata
mov [ eax ] 0x0
lea eax [ ebp + 0x210 ]
lea edx [ esp + 0xc ]
mov esi 0x104
sub edx eax
dec eax
mov cl [ edx + eax ]
test cl cl
je r-xdata
jmp r-xdata
test esi esi
jne r-xdata
mov [ eax ] cl
inc eax
dec esi
jne r-xdata
mov [ eax ] 0x0
lea edi [ esp + 0xc ]
or ecx 0xffffffff
xor eax eax
[ edi ]
```

```
push edi
mov [ esp + 0x10 ] rw-data
mov [ esp + 0x14 ] rw-data
mov [ esp + 0x18 ] rw-data
xor esi esi
lea edi [ esp + 0x10 ]
mov edx [ esp + esi * 4 + 0x10 ]
mov eax [ esp + 0x20 ]
mov esi 0x104
sub edx eax
call ebp
cmp eax 0xb7
je r-xdata
test esi esi
jne r-xdata
mov [ eax ] cl
inc eax
dec esi
jne r-xdata
inc esi
add edi 0x4
cmp esi 0x3
jb r-xdata
pop edi
pop esi
pop ebp
mov [ eax ] 0x0
pop ebx
add esp 0xc
retn 0x4
dec eax
mov cl [ edx + eax ]
test cl cl
je r-xdata
dec eax
pop edi
pop esi
pop ebp
mov [ eax ] 0x0
pop ebx
add esp 0xc
retn 0x4
mov eax [ edi ]
push 0x0
push eax
call ebx
cmp eax 0x1
je r-xdata
pop edi
pop esi
pop ebp
pop ebx
```

```
not ecx
dec ecx
mov al 0x5c
mov edx ecx
mov cl [ esp + edx + 0xa ]
sub edx 0x2
cmp cl al
je r-xdata
dec eax
mov cl [ edx + eax ]
test cl cl
je r-xdata
jmp r-xdata
mov ecx 0x40
xor eax eax
lea edi [ esp + 0x111 ]
mov [ esp + 0x110 ] 0x0
[ edi ]
stosw [ edi ]
lea esi [ esp + edx + 0xd ]
lea ecx [ esp + 0x110 ]
stosb [ edi ]
mov edx esi
mov edi 0x104
lea eax [ esp + 0x110 ]
sub edx ecx
mov cl [ esp + edx + 0xb ]
dec edx
cmp cl al
jne r-xdata
test edi edi
jne r-xdata
mov [ eax ] cl
inc eax
dec edi
jne r-xdata
lea edx [ esp + 0x110 ]
push rw-data
push edx
mov [ eax ] 0x0
call [ strstr@IAT ]
add esp 0x8
mov [ eax + 0x4 ] 0x0
lea eax [ esp + 0x110 ]
lea ecx [ esp + 0xc ]
push eax
push ecx
push rw-data
add ebp 0x10c
push 0x104
push ebp
mov [ esi ] 0x0
```

```
add esp 0xc
retn 0x4
sub_4012b0
sub esp 0x694
xor eax eax
push ebx
mov [ esp + 0x5 ] eax
push ebp
mov [ esp + 0xd ] eax
push esi
mov [ esp + 0x15 ] al
push edi
lea eax [ ecx + 0x10c ]
xor ebx ebx
push 0x2e
push eax
mov [ esp + 0x28 ] ecx
mov [ esp + 0x18 ] bl
call [ strrchr@IAT ]
mov edx eax
lea eax [ esp + 0x18 ]
add esp 0x8
mov esi 0xa
lea ecx [ esp + 0x10 ]
sub edx eax
cmp esi ebx
jne r-xdata
mov [ ecx ] al
inc ecx
dec esi
jne r-xdata
mov [ ecx ] bl
mov ecx 0x40
xor eax eax
lea edi [ esp + 0x5a1 ]
mov [ esp + 0x5a0 ] bl
lea edx [ esp + 0x5a0 ]
[ edi ]
lea ecx [ esp + 0x10 ]
stosw [ edi ]
push ecx
push rw-data
push 0x104
push edx
stosb [ edi ]
call sub_401a30
mov ecx 0x40
xor eax eax
lea edi [ esp + 0x4ad ]
mov [ esp + 0x4ac ] bl
[ edi ]
stosw [ edi ]
```

```
call sub_401a30
add esp 0x14
pop edi
pop esi
pop ebp
add esp 0x208
retn
dec eax
mov cl [ edx + eax ]
test cl cl
je r-xdata
jmp r-xdata
sub_4019f0
push esi
mov esi ecx
push edi
lea edi [ esi + 0x418 ]
push edi
call sub_401210
lea eax [ esi + 0x314 ]
add esi 0x51c
push eax
push edi
push rw-data
push 0x104
push esi
call sub_401a30
add esp 0x14
pop edi
pop esi
retn
sub_401a30
mov eax [ esp + 0x8 ]
cmp eax 0x7fffffff
jbe r-xdata
push ebx
xor ebx ebx
push esi
push edi
test eax eax
jne r-xdata
mov eax 0x80070057
retn
mov ecx [ esp + 0x18 ]
mov edi [ esp + 0x10 ]
lea esi [ eax - 0x1 ]
lea eax [ esp + 0x1c ]
push eax
push ecx
push esi
push edi
call [ _vsnprintf@IAT ]
```

```
mov esi [ RegOpenKeyExA@IAT ]
add esp 0x10
stosb [ edi ]
lea eax [ esp + 0x24 ]
lea ecx [ esp + 0x10 ]
push eax
push 0xf003f
push ebx
push ecx
push 0x80000000
call esi
mov ecx [ esp + 0x24 ]
mov ebp [ RegQueryValueExA@IAT ]
lea edx [ esp + 0x2c ]
lea eax [ esp + 0x49c ]
push edx
push eax
push ebx
push ebx
push ebx
push ecx
mov [ esp + 0x44 ] 0x104
call ebp
mov edx [ esp + 0x24 ]
push edx
call [ RegCloseKey@IAT ]
mov ecx 0x40
xor eax eax
lea edi [ esp + 0x8d ]
mov [ esp + 0x8c ] bl
[ edi ]
stosw [ edi ]
stosb [ edi ]
mov ecx 0x40
xor eax eax
lea edi [ esp + 0x191 ]
mov [ esp + 0x190 ] bl
[ edi ]
stosw [ edi ]
stosb [ edi ]
lea eax [ esp + 0x49c ]
mov edi 0x104
push eax
push rw-data
lea ecx [ esp + 0x198 ]
push edi
push ecx
call sub_401a30
add esp 0x10
lea edx [ esp + 0x1c ]
lea eax [ esp + 0x190 ]
push edx
```

```
add esp 0x10
test eax eax
jl r-xdata
mov ebx 0x80070057
pop edi
mov eax ebx
pop esi
pop ebx
retn
mov [ esi + edi ] 0x0
mov ebx 0x8007007a
cmp eax esi
ja r-xdata
jne r-xdata
pop edi
mov eax ebx
pop esi
pop ebx
retn
mov [ esi + edi ] 0x0
pop edi
mov eax ebx
pop esi
pop ebx
retn
sub_401aa0
push esi
push rw-data
push 0x0
push 0x0
call [ CreateMutexA@IAT ]
mov esi eax
test esi esi
je r-xdata
push ebx
push edi
call sub_401000
mov ecx [ esp + 0x10 ]
push rw-data
push ecx
mov ecx eax
mov [ rw-data ] eax
call sub_401100
mov edi [ _beginthread@IAT ]
push 0x0
push 0x0
push sub_401b40
call edi
push 0x0
push 0x0
push sub_401b60
mov esi eax
```

```
push 0xf003f
push ebx
push eax
push 0x80000000
call esi
lea ecx [ esp + 0x34 ]
mov [ esp + 0x34 ] edi
push ecx
mov eax [ esp + 0x20 ]
lea edx [ esp + 0x90 ]
push edx
push ebx
push ebx
push ebx
push eax
call ebp
test eax eax
je r-xdata
dec ecx
mov al [ edx + ecx ]
cmp al bl
je r-xdata
jmp r-xdata
mov ecx [ esp + 0x1c ]
push ecx
call [ RegCloseKey@IAT ]
mov ecx 0x10
xor eax eax
lea edi [ esp + 0x4c ]
mov [ esp + 0x294 ] bl
[ edi ]
mov ecx 0x40
lea edi [ esp + 0x295 ]
[ edi ]
stosw [ edi ]
xor edx edx
push rw-data
mov [ esp + 0x40 ] edx
mov [ esp + 0x4c ] 0x44
stosb [ edi ]
lea eax [ esp + 0x90 ]
mov [ esp + 0x44 ] edx
push eax
mov [ esp + 0x40 ] ebx
mov [ esp + 0x4c ] edx
call [ strstr@IAT ]
add esp 0x8
cmp eax ebx
je r-xdata
lea ecx [ esp + 0x10 ]
push rw-data
push ecx
```

```
call edi
mov ebx [ WaitForSingleObject@IAT ]
add esp 0x18
mov edi eax
push 0xffffffff
push esi
call ebx
push 0xffffffff
push edi
call ebx
mov ecx [ rw-data ]
call sub_401730
call [ GetLastError@IAT ]
cmp eax 0xb7
jne r-xdata
push esi
call [ CloseHandle@IAT ]
or eax 0xffffffff
pop esi
retn 0x10
_start
push ebp
mov ebp esp
push 0xffffffff
push r--data
push _except_handler3
mov eax [ fs : 0x0 ]
push eax
mov [ fs : 0x0 ] esp
sub esp 0x68
push ebx
push esi
push edi
mov [ ebp - 0x18 ] esp
xor ebx ebx
mov [ ebp - 0x4 ] ebx
push 0x2
call [ __set_app_type@IAT ]
pop ecx
or [ rw-data ] 0xffffffff
or [ rw-data ] 0xffffffff
call [ __p__fmode@IAT ]
mov ecx [ rw-data ]
mov [ eax ] ecx
call [ __p__commode@IAT ]
mov ecx [ rw-data ]
mov [ eax ] ecx
mov eax [ _adjust_fdiv@IAT ]
mov eax [ eax ]
mov [ rw-data ] eax
call sub_401d47
cmp [ rw-data ] ebx
```

```
call [ strstr@IAT ]
add esp 0x8
test eax eax
je r-xdata
mov ecx [ esp + 0x20 ]
lea edx [ esp + 0x8c ]
lea eax [ ecx + 0x10c ]
push eax
push edx
push rw-data
lea eax [ esp + 0x2a0 ]
push 0x104
push eax
call sub_401a30
mov ecx 0x40
xor eax eax
lea edi [ esp + 0x3ad ]
mov [ esp + 0x3ac ] bl
[ edi ]
add esp 0x14
lea ecx [ esp + 0x398 ]
stosw [ edi ]
push 0x104
lea edx [ esp + 0x298 ]
push ecx
push edx
stosb [ edi ]
call [ ExpandEnvironmentStringsA@IAT ]
mov edx [ esp + 0x20 ]
lea eax [ esp + 0x38 ]
lea ecx [ esp + 0x48 ]
push eax
add edx 0x418
push ecx
push edx
push ebx
push ebx
push ebx
push ebx
lea eax [ esp + 0x3b4 ]
push ebx
push eax
push ebx
call [ CreateProcessA@IAT ]
mov [ eax ] bl
lea edx [ esp + 0x28 ]
push edx
push 0xf003f
push ebx
push rw-data
push 0x80000000
call esi
```

```
jne r-xdata
call sub_401d32
push rw-data
push rw-data
call _initterm
mov eax [ rw-data ]
mov [ ebp - 0x6c ] eax
lea eax [ ebp - 0x6c ]
push eax
push [ rw-data ]
lea eax [ ebp - 0x64 ]
push eax
lea eax [ ebp - 0x70 ]
push eax
lea eax [ ebp - 0x60 ]
push eax
call [ __getmainargs@IAT ]
push rw-data
push rw-data
call _initterm
add esp 0x24
mov eax [ _acmdln@IAT ]
mov esi [ eax ]
mov [ ebp - 0x74 ] esi
cmp [ esi ] 0x22
jne r-xdata
push sub_401d44
call [ __setusermatherr@IAT ]
pop ecx
cmp [ esi ] 0x20
jbe r-xdata
inc esi
mov [ ebp - 0x74 ] esi
mov al [ esi ]
cmp al bl
je r-xdata
mov al [ esi ]
cmp al bl
je r-xdata
inc esi
mov [ ebp - 0x74 ] esi
jmp r-xdata
cmp [ esi ] 0x22
jne r-xdata
cmp al 0x22
jne r-xdata
mov [ ebp - 0x30 ] ebx
lea eax [ ebp - 0x5c ]
push eax
call [ GetStartupInfoA@IAT ]
test [ ebp - 0x30 ] 0x1
je r-xdata
```

```
test eax eax
jne r-xdata
pop edi
pop esi
pop ebp
pop ebx
add esp 0x694
retn
mov edx [ esp + 0x28 ]
lea eax [ esp + 0x30 ]
lea ecx [ esp + 0x8c ]
push eax
push ecx
push ebx
push ebx
push ebx
push edx
mov [ esp + 0x48 ] edi
call ebp
test eax eax
je r-xdata
mov eax [ esp + 0x28 ]
push eax
call [ RegCloseKey@IAT ]
push ebx
push rw-data
push rw-data
push ebx
call [ MessageBoxA@IAT ]
pop edi
pop esi
pop ebp
pop ebx
add esp 0x694
retn
sub_4015c0
sub esp 0x54
mov edx ecx
push edi
mov ecx 0x10
xor eax eax
lea edi [ esp + 0x18 ]
mov [ esp + 0x4 ] eax
[ edi ]
mov [ esp + 0x8 ] eax
lea ecx [ esp + 0x4 ]
mov [ esp + 0xc ] eax
push ecx
mov [ esp + 0x14 ] eax
lea eax [ esp + 0x18 ]
lea ecx [ edx + 0x418 ]
push eax
```

```
cmp al 0x20
jbe r-xdata
inc esi
mov [ ebp - 0x74 ] esi
push 0xa
pop eax
movzx eax [ ebp - 0x2c ]
jmp r-xdata
push eax
push esi
push ebx
push ebx
call [ GetModuleHandleA@IAT ]
push eax
call sub_401aa0
mov [ ebp - 0x68 ] eax
push eax
call [ exit@IAT ]
sub_4010e0
push esi
mov esi ecx
call sub_401150
test [ esp + 0x8 ] 0x1
je r-xdata
mov eax esi
pop esi
retn 0x4
push esi
call ??3@YAXPAX@Z
add esp 0x4
sub_401150
mov [ ecx ] r--data
mov ecx [ rw-data ]
test ecx ecx
je r-xdata
mov [ rw-data ] 0x0
retn
mov eax [ ecx ]
push 0x1
call [ eax ]
sub_401b2c
push esi
mov esi [ CloseHandle@IAT ]
call esi
push edi
call esi
pop edi
pop ebx
xor eax eax
pop esi
retn 0x10
sub_401b40
```

```
push ecx
push 0x0
push 0x0
push 0x0
push 0x0
push 0x0
add edx 0x51c
push 0x0
push edx
mov [ esp + 0x3c ] 0x44
call [ CreateProcessA@IAT ]
pop edi
add esp 0x54
retn
sub_401620
sub esp 0x20
mov ecx 0x7
xor eax eax
push ebp
push esi
push edi
mov esi rw-data
lea edi [ esp + 0xc ]
[ edi ] [ esi ]
movsw [ edi ] [ esi ]
movsb [ edi ] [ esi ]
lea edi [ esp + 0xc ]
or ecx 0xffffffff
[ edi ]
not ecx
dec ecx
xor esi esi
mov ebp ecx
mov ecx [ esp + 0x38 ]
xor edi edi
test ecx ecx
jbe r-xdata
mov eax [ esp + 0x34 ]
pop edi
pop esi
pop ebp
add esp 0x20
retn 0xc
push ebx
mov ebx [ esp + 0x38 ]
mov eax edi
xor edx edx
div ebp
mov al [ esp + edx + 0x10 ]
mov dl [ esi + ebx ]
add al dl
mov dl [ esp + 0x34 ]
```

```
mov ecx [ rw-data ]
call sub_401180
mov ecx [ rw-data ]
jmp sub_4012b0
sub_401b60
mov ecx [ rw-data ]
push 0xc
call sub_4016a0
mov ecx [ rw-data ]
jmp sub_4015c0
??3@YAXPAX@Z
jmp [ operator delete@IAT ]
??2@YAPAXI@Z
jmp [ operator new@IAT ]
__CxxFrameHandler
jmp [ __CxxFrameHandler@IAT ]
sub_401ba0
push esi
mov esi ecx
call ??1type_info@@UAE@XZ
test [ esp + 0x8 ] 0x1
je r-xdata
mov eax esi
pop esi
retn 0x4
push esi
call ??3@YAXPAX@Z
pop ecx
_CxxThrowException
jmp [ _CxxThrowException@IAT ]
??1type_info@@UAE@XZ
jmp [ type_info::~type_info@IAT ]
_XcptFilter
jmp [ _XcptFilter@IAT ]
_initterm
jmp [ _initterm@IAT ]
sub_401d32
push 0x30000
push 0x10000
call _controlfp
pop ecx
pop ecx
retn
sub_401d44
xor eax eax
retn
sub_401d47
retn
_except_handler3
jmp [ _except_handler3@IAT ]
_controlfp
jmp [ _controlfp@IAT ]
```

xor al dl	sub_401d60
mov [esi + ebx] al	mov eax [ebp - 0x10]
inc esi	push eax
inc edi	call ??3@YAXPAX@Z
cmp esi ecx	pop ecx
jb r-xdata	retn
xor edi edi	sub_401d6b
test esi 0x3ff	mov eax r--data
jne r-xdata	jmp __CxxFrameHandler
mov eax ebx	sub_401d80
pop ebx	mov eax r--data
pop edi	jmp __CxxFrameHandler
pop esi	
pop ebp	
add esp 0x20	
retn 0xc	