Fall 2017

Detecting Encrypted Malware Using Hidden Markov Models

Dhiviya Dhanasekar
San Jose State University

Detecting Encrypted Malware Using Hidden Markov Models

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Dhiviya Dhanasekar

December 2017

The Designated Project Committee Approves the Project Titled

Detecting Encrypted Malware Using Hidden Markov Models

by

Dhiviya Dhanasekar

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2017

Dr. Mark Stamp	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science
Mr. Fabio Di Troia	Department of Computer Science

ABSTRACT

Detecting Encrypted Malware Using Hidden Markov Models

by Dhiviya Dhanasekar

Encrypted code is often present in some types of advanced malware, while such code virtually never appears in legitimate applications. Hence, the presence of encrypted code within an executable file could serve as a strong heuristic for detecting malware. In this research, we consider the feasibility of detecting encrypted code using hidden Markov models.

ACKNOWLEDGMENTS

I extend my gratitude to my project advisor, Dr. Mark Stamp for his guidance and feedback throughout the course of the project. I would like to thank my committee members, Dr. Katerina Potika and Mr. Fabio Di Troia for providing me with their valuable advice.

I would also like to thank my family and friends for their unfailing support and continuous encouragement during my years of study. In particular, I have to thank my husband, Sujandharan for everything he did to support my ambitions to attend graduate school while I work.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Literature Survey 3

2.1 Malware and Viruses 3

2.2 Types of Viruses 3

2.3 Evolution of Viruses 4

2.3.1 Early Viruses 4

2.3.2 Encrypted Viruses 4

2.3.3 Polymorphic Viruses 7

2.3.4 Metamorphic Viruses 7

2.4 Virus Encryption 8

2.4.1 Simple Substitution Cipher 9

2.5 Common Virus Detection Techniques 9

2.5.1 Signature Based Detection 10

2.5.2 Anomaly Based Detection 11

2.5.3 Emulation Based Detection 11

2.6 Machine Learning Techniques for Malware Detection 11

2.6.1 Data Mining Methods and Artificial Neural Networks . . . 13

2.6.2 Hidden Markov Models 13

2.7 Re-creating HMM Applications 15

2.7.1 Implementing HMM 15

2.7.2 HMM and English Alphabet 15
2.7.3 HMM and Encrypted English Alphabet 16
2.8 Discussion 18

3 Implementation and Results 20
3.1 Experimental Setup 20
3.1.1 Generating HMM Inputs 20
3.1.2 HMM Implementation 21
3.1.3 Generating Training Models 23
3.1.4 Generating Test Files 23
3.1.5 Scoring with Test Files 23
3.1.6 Generating Scatter Plots 24
3.1.7 Measuring Accuracy of Models 25
3.2 Language and Performance Experiments 25
3.3 Experimenting with Maximum Number of Iterations 26
3.4 Experimenting with T 26
3.5 Testing with Encrypted Opcode Files and Plaintext Files 27
3.5.1 Experimenting with N, M 27
3.6 Testing with Partially Encrypted Opcode Files 27
3.6.1 Experimenting with Various Window Sizes 28
3.6.2 Experimenting with Various Window Sizes and Window Sliding Lengths 34
3.7 Testing with Encrypted Byte Files and Plaintext Files 36
3.8 Discussion 38

4 Conclusion and Future Work 40

LIST OF REFERENCES 42

APPENDIX

A Appendix A: Scatter Plots of Scores for Different Window Lengths . . 45

B Appendix B: AUC Values 55

LIST OF TABLES

1 Strengths and weaknesses of common virus detection techniques . 12

2 HMM notation 14

3 Transpose of B matrix for English letters and space 17

4 Preview of the most frequently occurring opcodes and their frequencies, ordered in the descending order of the frequencies. . . . 22

5 AUC values for different combinations of window sizes and window sliding lengths. 37

B.6 AUC values for different window sizes 55

B.7 AUC values, ROC threshold, TPR and FPR values for N, M combinations 58

LIST OF FIGURES

1 Encrypted virus and its replication. 5

2 Example of an encrypted virus code before execution. 6

3 The encrypted virus during execution. 6

4 The virus code after decryption. 6

5 Structure of a polymorphic virus. 8

6 Various generations of a metamorphic virus. 9

7 Simple substitution cipher example. 10

8 Machine learning implementation. 12

9 Hidden Markov model. 14

10 B matrix for 'a' shows that space was replaced with 'a'. 18

11 B matrix for 'f' shows that 'e' was replaced with 'f'. 19

12 Scatter plot for scores of encrypted files and corresponding plaintext files generated using $N = 2$, $M = 30$ training model. 28

13 ROC curve for encrypted files and plaintext files generated using $N = 2$, $M = 30$ training model. 29

14 AUC values for all window sizes. 30

15 ROC curves for different window lengths with $N = 2$, $M = 30$. . . 31

16 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 800. 32

17 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1300. 32

18 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1850. 33

19 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2200. 34

20 Splitting a file into windows of fixed sliding lengths and variable sliding lengths. 35

21 AUC curves for different window sizes and sliding lengths 36

22 Scatter plot for scores of encrypted byte files and corresponding plaintext byte files generated using $N = 2$, $M = 246$ training model. 39

23 Scatter plot for scores of encrypted byte files and corresponding plaintext byte files generated using $N = 3$, $M = 246$ training model. 39

A.24 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 100. 45

A.25 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 150. 45

A.26 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 200. 46

A.27 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 300. 46

A.28 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 400. 47

A.29 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 500. 47

A.30 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 700. 48

A.31 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 800. 48

A.32 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 900. 49

A.33 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1000. 49

A.34 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1200. 50

A.35 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1300. 50

A.36 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1500. 51

A.37 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1700. 51

A.38 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1850. 52

A.39 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2000. 52

A.40 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2100. 53

A.41 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2200. 53

A.42 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2400. 54

A.43 Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2500. 54

B.44 AUCs for different window lengths using sliding length = 50. . . . 56

B.45 AUCs for different window lengths using sliding length = 100. . . 56

B.46 AUCs for different window lengths using sliding length = 150. . . 57

B.47 AUCs for different window lengths using sliding length = 200. . . 57

CHAPTER 1

Introduction

Encrypted and polymorphic viruses are often encoded in files as encrypted sections. Hence encrypted code in a file often points to the presence of encrypted and polymorphic malware in the file. The aim of this research is to detect such encrypted sections in files.

Although disassembling files is indispensable for proper analysis of malware, disassemblers often produce incorrect results with a x86/x64 architecture [1]. Hence instead of using disassembling techniques, in this research, we will use hidden Markov models (HMMs) to detect encrypted malware. The scope of this project includes using HMM scores over short sequences of the file with malware content and applying transformation techniques to detect the encrypted sections in the file.

As part of this project, we completed the literature review needed for the research topic, developed the code required to generate encrypted input files for the algorithm, implemented a HMM model and tested it on the alphabets of the English language to partition them into vowels and consonants. We also completed the code to generate a row stochastic, alphabet bigram matrix for the English Language, with which our HMM Model was able to detect the keys for the encrypted English text we used as inputs to the HMM. In order to get closer to our goal of detecting encrypted code, we generated opcodes for several executables and encrypted the opcodes using a simple substitution cipher. We input these encrypted opcodes to the HMM program to obtain preliminary results. We then experimented with the HMM algorithm parameters to produce various training models. We scored each of these models using a scoring algorithm, with both encrypted and plaintext opcodes from test files. We generated the Area Under the Curve (AUC) values using Receiver Operating Characteristic (ROC) curves for each of these models to detect encrypted and plaintext test files. We repeated the experiments done using opcodes with bytes. We also encrypted sections of test files and scored them using various window lengths and window overlapping lengths to detect encrypted and plaintext opcode sections.

This thesis is organized into four chapters. Chapter 2 gives a summary of the literature review necessary to understand the origins of the project and presents the preliminary results obtained during the first half of the project by re-ceating existing HMM applications. Details about the experimental set-up, experiments conducted using various HMM parameters, scoring algorithm used and the results when tested with fully encrypted files and partially encrypted files are specified in Chapter 3. Finally, Chapter 4 summarizes possible extensions and future work for this project.

CHAPTER 2

Literature Survey

In this chapter, we introduce and define the terms that will be used in the rest of this thesis paper. We start with the definition of malwares and viruses. We specify the difference between them. We then explain the types of viruses and the ways in which viruses can be categorized. As part of the evolution section, we explain how virus writers created each of these virus categories as advancements in anti-virus softwares were made. Since our project relates to encrypted and polymorphic viruses, we provide more detailed literature review for these topics. Finally, we end this chapter with virus detection techniques that are popular currently and machine learning techniques that can be used for detecting viruses, while providing special emphasis on hidden Markov models, which we we use in the project.

2.1 Malware and Viruses

Malware or malicious software, is any software that has a malicious intent. It is generally used to disrupt computer or mobile operations, gather sensitive information, gain access to private computer systems, or display unwanted advertising [2]. Malwares are generally classified as viruses, worms, trojans, trapdoors, rabbits or spyware.

A virus is a self-replicating computer program that operates without the consent of the user. It spreads by attaching a copy of itself to some part of a program file [3]. It relies on someone or something to propagate from one system to another [4]. Although not all malware are viruses, the terms malware and virus have been used interchangeably in this thesis paper.

2.2 Types of Viruses

Viruses can be classified in two ways [5]. The first classification is based on what they infect. The categories of viruses based on this classification include viruses that affect the boot sector, files, macros, memory, applications, emails, data, compilers, library routines, debuggers, and even anti-virus softwares [4]. The second classification is based on how they infect. Using this classification, viruses can be categorized as encrypted, metamorphic, and polymorphic viruses. The next section traces the evolution of these viruses and explains how and why virus writers came up with these viruses.

2.3 Evolution of Viruses

From early viruses that merely replicated to the current zero-day viruses, virus writers have constantly been innovating to counter the advancements brought about by anti-virus softwares in detecting viruses. This section summarizes the history of how various types of viruses were created and thus helps us understand the nature of common types of viruses, some of which were mentioned in the previous section.

2.3.1 Early Viruses

The Brain virus, considered as the first virus, was released in January 1986 [4, 6]. It targeted the boot sector by changing the disk label of affected sectors to "©Brain" [7]. It would scan the boot sector each time the disk was read and if the boot sector was not affected, it would infect it. As a result, it was difficult to remove this virus completely. The brain virus was a simple virus without any malicious intent, whose aim was just to replicate and copy itself. Each time it copies itself, it creates an exact copy of its self and hence has a sequence of bytes known as a signature, that is characteristic of this virus [3]. Anti-virus software writers hence created the signature based detection method for such viruses.

2.3.2 Encrypted Viruses

To counter explicit signature detection in anti-virus softwares, virus writers created the encrypted virus. In such viruses, the signature would be hidden by using a simple encryption such as Simple Substitution using XOR. However, encrypted viruses would always include the same code to decrypt the encryption. Hence even if the key for encryption was modified for each replication, when the decryption code block was subjected to signature based detection, encrypted viruses could be detected.

Figure 1 shows an encrypted virus with an encrypted body, decryption routine and a key that decrypts the body [3]. It also shows that the decryption routine remains the same each time the virus is replicated, although the encrypted body varies for each replication. Figure 2 shows an example of the code used for an encrypted virus, before it starts executing. It shows the virus decryption routine and the encrypted virus body, the two key characteristics of an encrypted virus [3]. Figure 3 shows the virus from Figure 2, when it has executed the first five instructions in the code and it has completed decrypting the first byte of the virus body [3]. Figure 4 shows the virus from Figure 2, after it has been decrypted [3].

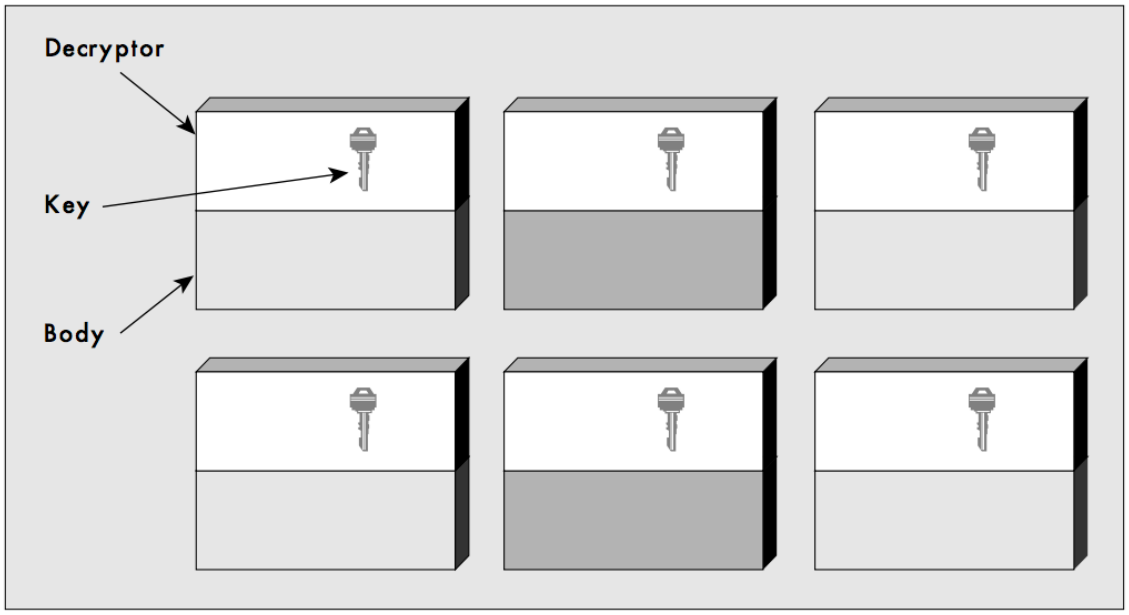
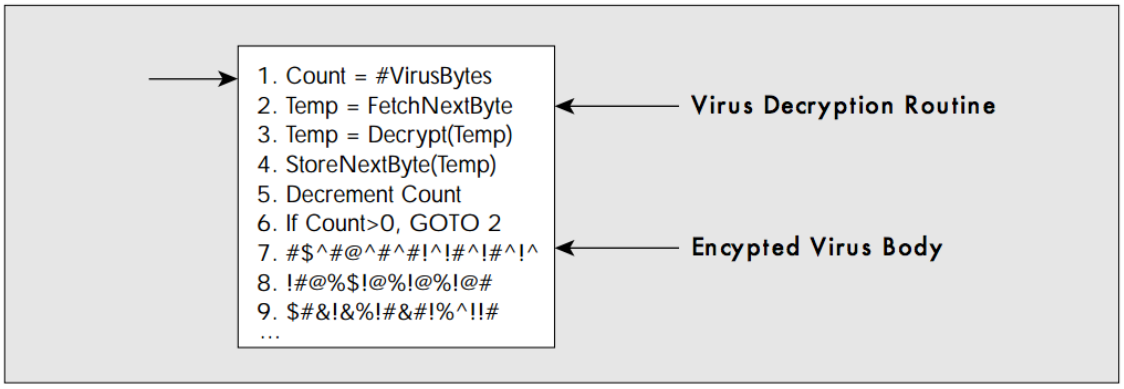
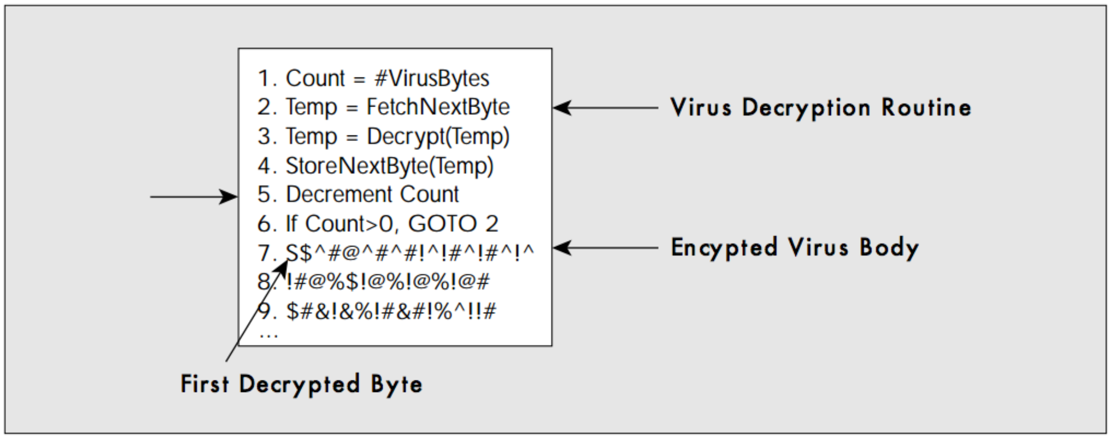


Figure 1: Encrypted virus and its replication.



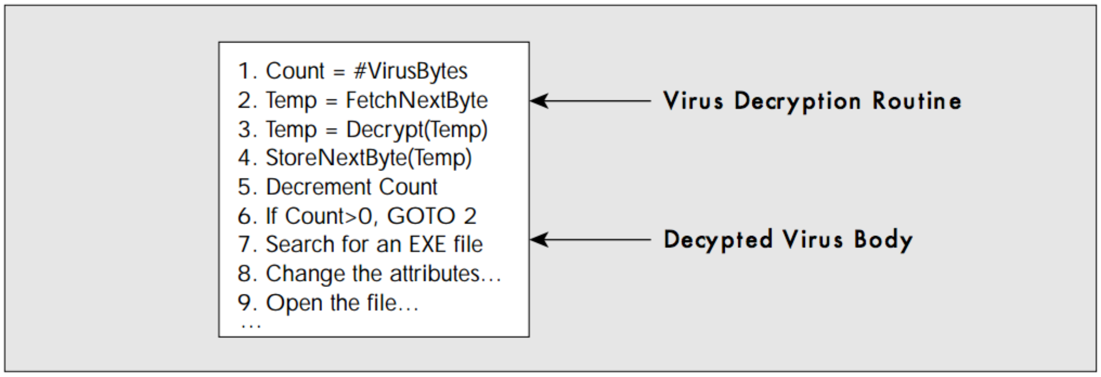
```
1. Count = #VirusBytes
2. Temp = FetchNextByte
3. Temp = Decrypt(Temp)
4. StoreNextByte(Temp)
5. Decrement Count
6. If Count>0, GOTO 2
7. #$^#@^#^#!^!#^!#^!^
8. !#@%$!@%!@%!@#
9. $#&!&%!#&#!%^!!#
...
```

Figure 2: Example of an encrypted virus code before execution.



```
1. Count = #VirusBytes
2. Temp = FetchNextByte
3. Temp = Decrypt(Temp)
4. StoreNextByte(Temp)
5. Decrement Count
6. If Count>0, GOTO 2
7. S$^#@^#^#!^!#^!#^!^
8. !#@%$!@%!@%!@#
9. $#&!&%!#&#!%^!!#
...
```

Figure 3: The encrypted virus during execution.



```
1. Count = #VirusBytes
2. Temp = FetchNextByte
3. Temp = Decrypt(Temp)
4. StoreNextByte(Temp)
5. Decrement Count
6. If Count>0, GOTO 2
7. Search for an EXE file
8. Change the attributes...
9. Open the file...
...
```

Figure 4: The virus code after decryption.

2.3.3 Polymorphic Viruses

To evade detection of decryption routines in encrypted viruses, virus writers began to morph the decryption code. These viruses became known as polymorphic viruses. The first polymorphic virus, 1260, a virus of chameleon family appeared by 1990, was developed by Mark Washburn [8].

Figure 5 shows how a polymorphic virus replicates by applying a morphed decryptor to the next generation of the virus, while keeping the body of the decrypted virus the same [9]. Polymorphic viruses are able to create an unlimited number of new and different decryptors [8]. These viruses utilize code obfuscation techniques such as insertion of junk codes or simple substitution of instructions using the XOR operator to mutate its decryptor and thereby build a new one for the newly infected system [10]. Polymorphic viruses are thus a category of computer viruses that are highly complex and difficult to detect [3]. However, since the body of the decrypted virus remains the same and the virus will decrypt the body eventually during its execution and replication, the current detection mechanism for these viruses involves applying signature based detection when the code is executing in an emulator. This is known as emulator based detection. Such a detection mechanism is highly slow.

2.3.4 Metamorphic Viruses

While a polymorphic virus encrypts its original code to avoid being recognized by anti-virus software, the metamorphic virus changes its code to an equivalent form [5]. Each time the metamorphic virus propagates, it reprograms itself. Figure 6 shows an example of the various generations of a metamorphic virus and how they look different from one another [9]. Although the virus appears to have changed, it effectively serves the same purpose. Thus the metamorphic virus avoids detection by ensuring there is no standard sequence of bytes, that is characteristic of itself and hence avoids

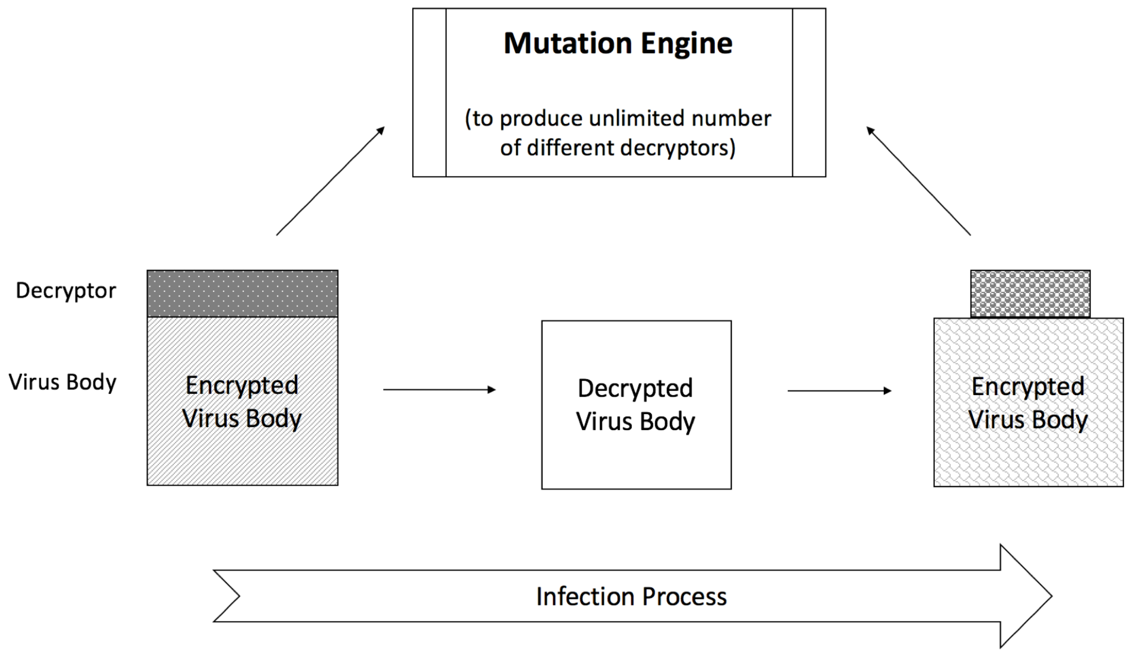


Figure 5: Structure of a polymorphic virus.

detection by static analysis techniques or signature based detection systems. The current detection mechanism for metamorphic viruses involves applying emulator based detection.

2.4 Virus Encryption

Virus encryption typically involves encrypting each byte in the virus body. The encryption used here is generally simple XOR. The simple XOR is very useful because XORing twice with a value gives back the original byte data. For example, if we take a byte value of 0x50 and XOR it with 0x99, we get 0xC9. XORing 0xC9 with 0x99 will give 0x50 back again. Thus, the same XOR can be used for both encryption and decryption. This reduces the amount of work needed to create a virus by not having to write two different methods to encrypt and decrypt. Apart from being simple, XOR is also a quick way to encrypt and decrypt a virus body [8].

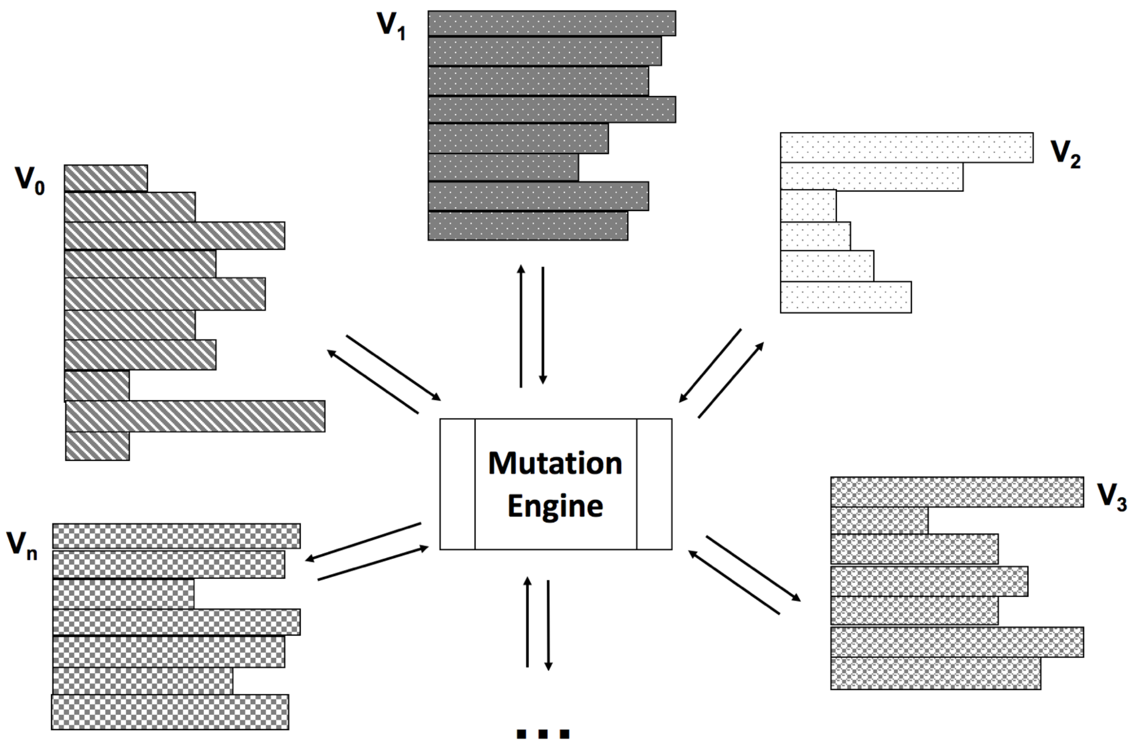


Figure 6: Various generations of a metamorphic virus.

2.4.1 Simple Substitution Cipher

Simple substitution cipher is a special case of XOR encryption. It comprises of two character sets or symbol sets - plaintext characters and key characters. Each plaintext character is mapped to a key character to encrypt the plaintext and get the ciphertext [4]. An example of simple substitution is shown in Figure 7. As will be described in Chapter 3, we use simple substitution in our implementation to encrypt files and generate input data.

2.5 Common Virus Detection Techniques

Today's viruses cause damage estimated to cost billions of dollars each year [11, 12]. This section summarizes three common virus detection techniques used in the industry. Table 1 summarizes the advantages and disadvantages [4, 13] of these techniques. With improvements in virus detection techniques employed by anti-virus

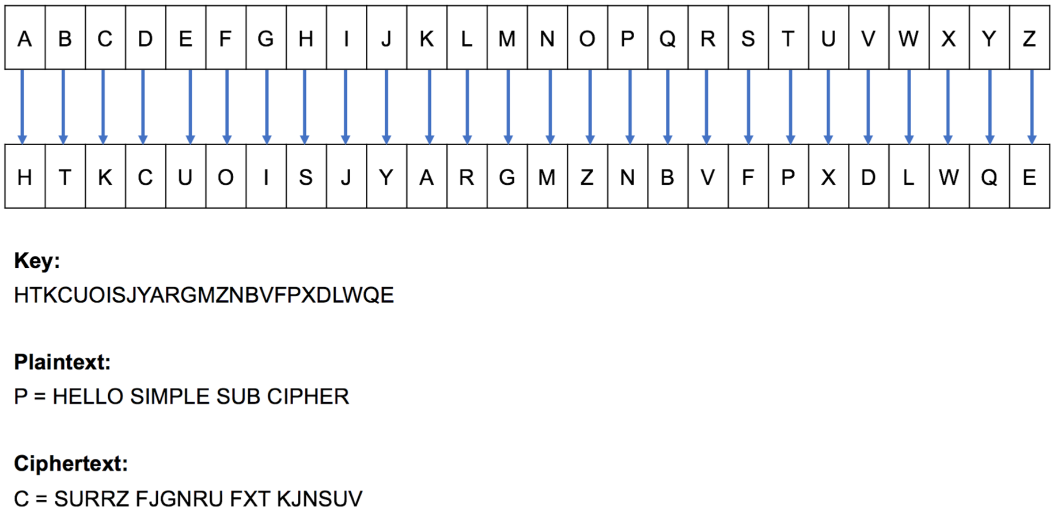

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
H T K C U O I S J Y A R G M Z N B V F P X D L W Q E
Key:
HTKCUOISJYARGMZNBVFPXDLWQE
Plaintext:
P = HELLO SIMPLE SUB CIPHER
Ciphertext:
C = SURRZ FJGNRU FXT KJNSUV

Figure 7: Simple substitution cipher example.

softwares, virus writers consistently produce zero-day viruses that cannot be detected by traditional techniques. In this section, we also point out flaws with these detection techniques and explain why we would need machine learning techniques to improve anti-virus softwares.

2.5.1 Signature Based Detection

To build such a detection system, a signature or a sequence of bytes that are characteristic of a virus is extracted and added to a database of virus signatures. Signature based virus detection systems scan a suspected virus against this database for matching signatures. If a signature is found for the suspected virus in the database, then the anti-virus system will label it as a virus. Most anti-virus softwares found in the industry today are largely based on signature based detection.

While this method of detection is fast, it cannot handle new viruses or zero-day viruses, as the signature of the new viruses will be unknown to the system. Such systems also require a large amount of storage to keep track of all the viruses known till date. Thus anti-virus systems cannot be completely reliant on signature based detection.

2.5.2 Anomaly Based Detection

Anomaly based detection systems monitor for the presence of abnormal activity. If any abnormal activity is detected, then the system alerts to show that there could be malware present in the system [4]. Such a detection system has a higher probability of false alarms and hence is not completely reliant by itself. In order to determine if an alarm is false or not, it needs to be used in conjunction with another detection system.

2.5.3 Emulation Based Detection

Emulation based detection involves executing the suspected malware in a sandboxed or virtual environment, where the anti-virus can identify a specific sequence of instructions that are executed by a known malware. As mentioned in previous sections, emulation based virus detection is highly effective for polymorphic and metamorphic viruses. However, it is slow because it needs to wait for the virus to start executing its replication stage, before it can detect the virus. Also, such systems do not always produce the correct results on all operating systems [1].

2.6 Machine Learning Techniques for Malware Detection

Common virus detection techniques discussed above have not been able to detect all types of viruses. A more generic approach, that could detect even unknown viruses without executing them was needed. This gap is being filled by machine learning algorithms.

As shown in Figure 8, machine learning algorithms are primarily implemented in two phases - training and testing. During the training phase, the algorithm or model is provided input data similar to what it needs to detect. The input data is

Table 1: Strengths and weaknesses of common virus detection techniques

Detection method	Strengths	Weaknesses
Signature based	Efficient, fast, known simple encrypted viruses	Zero-day malware, high storage cost
Anomaly based	New malware	Costly to implement, false alarms
Emulation based	Polymorphic and metamorphic viruses	Costly to implement, slow

transformed to suit the needs of the algorithm before it is provided to the machine learning algorithm. The algorithm runs using these inputs to generate training data. The testing phase is the evaluation phase of real life data. During the testing phase, the output data from the training phase is loaded into the algorithm and used as a baseline to evaluate the real life data against the training data. The types of machine learning algorithms used for malware detection are summarized in the subsections that follow.

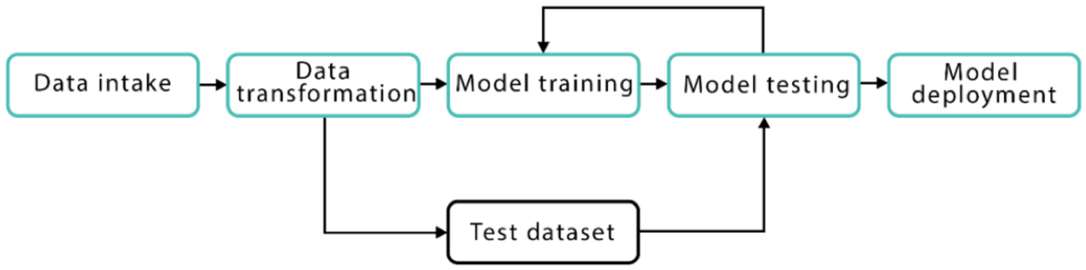


Figure 8: Machine learning implementation.

2.6.1 Data Mining Methods and Artificial Neural Networks

In Data Mining Methods and Artificial Neural Networks, features or characteristics are extracted from the input data. Features could include the size of the malware, n-gram byte sequences, plaintext strings found in the disassembled files, system resource information such as the set of DLLs, etc. These features are provided as inputs to the machine learning algorithm. In [14], three different data mining methods known as RIPPER, Naive Bayes and Multi-Naive Bayes were implemented to classify malware. All three algorithms provided a higher virus detection rate than the common signature based virus detection method. This was one of the first research projects implementing machine learning for malware detection.

Given its success, more projects using machine learning methods such as K-Nearest Neighbors, Support Vector Machines, Artificial Neural Networks, Decision Trees and variations of these methods were tested for the application of malware detection. Some of them provided accuracies as high as 94%-98%. However, most Data Mining Methods also resulted in problems such as high false positive rates and Artificial Neural Networks resulted in overfitting problems. Comparison of these methods and the types of malware that were accurately detected can be found at [15].

2.6.2 Hidden Markov Models

A Markov model is a stochastic model used to represent randomly changing systems, where future states depend only on the current state and not on the events that occurred before it [16]. The systems modeled by a Markov model are also known as Markov Processes. A hidden Markov model (HMM) is a Markov model where the Markov process has unobserved or hidden states. Even though the states are hidden, a HMM can map each observation (or input to the HMM model) to each state in the model with varying probabilities [17]. HMMs can therefore be considered to as state machines, that can help to tag a sequence of inputs or observations, to a state. Some applications of HMMs include parts-of-speech tagging (i.e. classifying different parts of a text as nouns, verbs, etc.), biological protein sequence classification and gene predictions [18].

Figure 9 shows the various parameters associated with a HMM algorithm. These are the notations we will use in the rest of this thesis paper. The parameters used in Figure 9 are explained in Table 2 [17, 18].

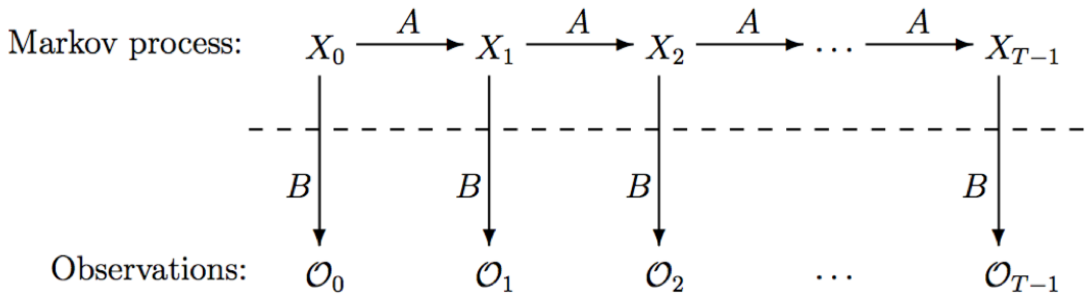


Figure 9: Hidden Markov model.

Table 2: HMM notation

O	is the observation sequence or the input sequence
T	is the length of the input or the observation sequence
M	is the number of unique observations or unique inputs used
N	is the number of states in the model
A	is the state transition matrix
B	is the observation matrix
X	is the set of hidden states, represented by the Markov model
π	is the probability of starting in a particular state

The matrix A contains probabilities for each state transition. Each value in this $N \times N$ matrix represents the probability of moving from state i to state j. The B matrix contains emission probabilities. Each value in this $N \times M$ matrix represents the probability of an observation or an unique input being generated from state i. Thus the transpose of B matrix represents the probability that an observation (or an unique input to the HMM) belongs to a state i. π is generally an array or list with N elements. All the elements in π add up to one. The element in position i of π represents the probability of starting in state i.

2.7 Re-creating HMM Applications

In this section, we present the preliminary work completed to implement an initial version of the HMM algorithm. We begin with the HMM implementation details. We then provide details on the work completed to prove that hidden Markov models are good at detecting encrypted alphabets of the English language and experiments conducted with various parameters in the HMM model.

2.7.1 Implementing HMM

We implemented the HMM pseudo-code specified in [17] as the core HMM algorithm to use for our applications. At this point in time, using static arrays and initializing N, M was sufficient. As we shall see in Chapter 3, this had to be changed to suit the requirements of this thesis. During the course of the preliminary work described in this chapter, we varied the state transition matrix, observation sequence, number of states, observation symbols and length of the observation sequence as required for different applications described in this chapter. We also experimented with the convergence values for the HMM model and the minimum number of iterations needed for convergence. The details of these modifications will be mentioned as part of the applications described in the sub-sections that follow.

2.7.2 HMM and English Alphabet

To test our implementation of the HMM model, we re-create an application of hidden Markov models from [19]. We present the results of this experiment in this section. As part of this experiment, we input English text downloaded from [20] into the HMM as the observation sequence. The observation symbols or the set of unique inputs used comprise of the twenty six letters of the English Alphabet and the space character. All other characters in the English text were ignored and all twenty six letters were converted to lowercase, so that capital letters and lowercase letters were considered the same. We train the model using the parameters $N = 2$, $T = 50{,}000$ and $M = 27$ (the 26 letters of the alphabet and space).

The transpose of the B matrix obtained for this experiment from the HMM model is listed in Table 3. The results show that the HMM has split the 27 observation symbols into two categories, namely vowels and consonants. The two categories are the hidden states of the HMM. The first column of the results is zero for the vowels, whereas it is non-zero for the consonants. Although the letter 'u' is a vowel and not a consonant, it has a non-zero value in its first column. We conclude that this is an anomaly and ignore it, knowing that the HMM model is not always completely accurate.

We repeated the experiment using $N = 3$ and saw that the second column in the B matrix contains zeros for vowels. Repeating the experiment with $N = 4$ produces a B matrix, that has non-zero values in the second column for the vowels. We thus conclude based on these results that our HMM implementation works as expected.

2.7.3 HMM and Encrypted English Alphabet

We extended the application described in the previous section to encrypted characters in English. Using the same observation symbols, we created a program to encrypt the characters of the observation sequence using a simple substitution cipher, where the letter 'a' was substituted with 'b', 'b' with 'c' and so on. We replaced the state transition matrix in the HMM model with a constant row stochastic bi-gram probabilities matrix, which was computed using letter-letter bi-gram frequencies in the Brown corpus downloaded from [20]. Although the Brown corpus was used to

Table 3: Transpose of B matrix for English letters and space

Character	Hidden State 1	Hidden State 2
a	0.0000	0.1335
b	0.0195	0.0000
c	0.0467	0.0002
d	0.0586	0.0002
e	0.0000	0.2496
f	0.0413	0.0000
g	0.0179	0.0064
h	0.0918	0.0000
i	0.0000	0.1365
j	0.0025	0.0000
k	0.0040	0.0006
l	0.0536	0.0000
m	0.0353	0.0000
n	0.1048	0.0000
o	0.0000	0.1445
p	0.0277	0.0041
q	0.0016	0.0000
r	0.0990	0.0000
s	0.0945	0.0006
t	0.1108	0.0514
u	0.0017	0.0463
v	0.0149	0.0000
w	0.0262	0.0000
x	0.0045	0.0000
y	0.0080	0.0207
z	0.0011	0.0000
space	0.1339	0.2055

generate the bi-gram probabilities matrix and the observation sequence, there was no overlap in the files used to generate these inputs.

With $N = 27$, $T = 10{,}000$ and $M = 27$, a 27×27 matrix B was generated by the HMM model. Each row in the matrix represents an encrypted letter and each column the plain-text letter. Plotting each row of the matrix on a line graph showed that the cell that maps the encrypted letter with the plain-text letter had the highest value in that row. This was true for 26 characters. We saw an anomaly for the letter 'j' alone. The graphs plotted for the letters 'a' and 'f' are shown in Figure 10 and Figure 11 respectively. The B matrix generated by the HMM model was thus able to identify the plain text letter for each encrypted letter in the text.

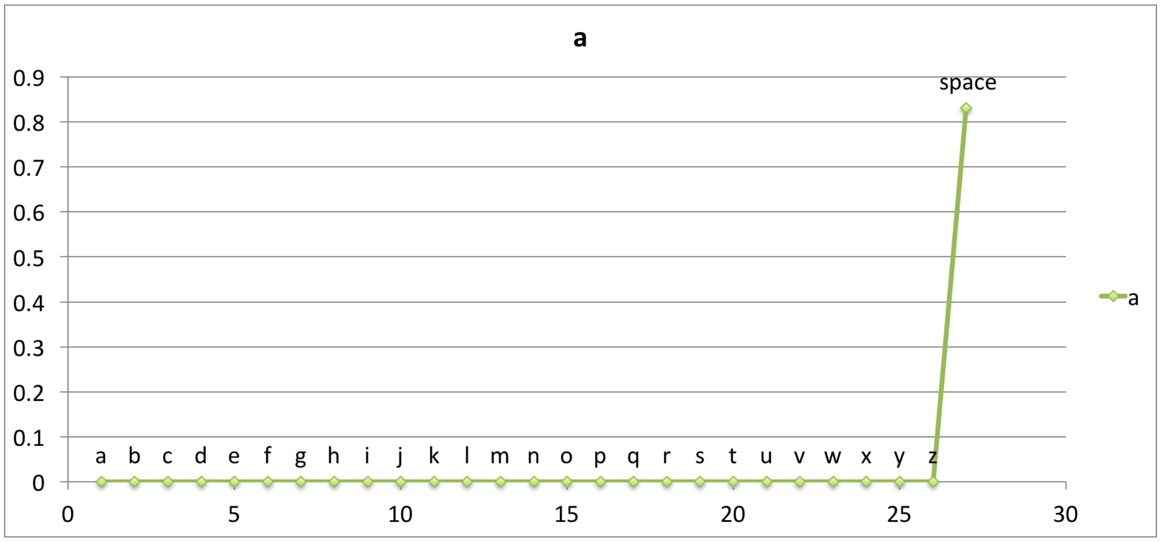


Figure 10: B matrix for 'a' shows that space was replaced with 'a'.

2.8 Discussion

As explained in the previous sections, common virus detection techniques have not been sufficient to detect all types of viruses in every environment. Given that there is sufficient evidence of HMMs being able to detect metamorphic viruses and encrypted keys [21, 22, 23, 24, 25], we propose to apply HMMs on polymorphic and

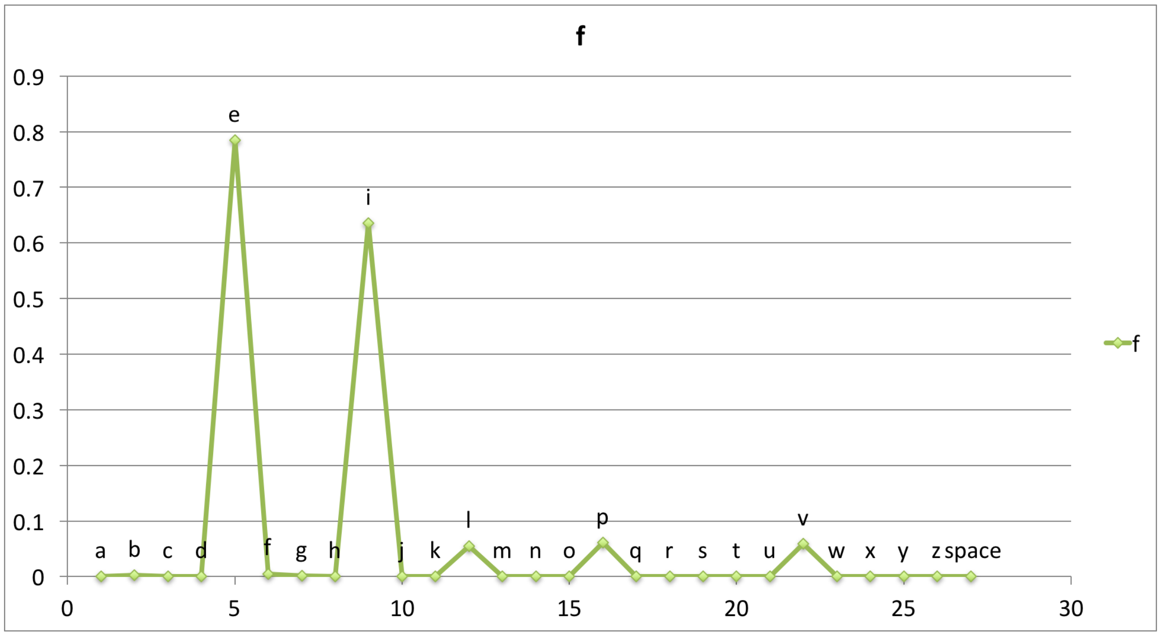


Figure 11: B matrix for 'f' shows that 'e' was replaced with 'f'.

encrypted viruses to classify malware and non-malware sections of code files and thus detect malware. Using a preliminary implementation of HMM, we were also able to prove that HMM is good at identifying the keys for encrypted characters in the English language. With this as the basis, we began experimenting the HMM algorithm with encrypted opcodes to obtain a similar result. The implementation details of this and the results are provided in the next chapter.

CHAPTER 3

Implementation and Results

In this chapter, we describe the experiments performed using HMM with encrypted opcodes and encrypted bytes extracted from executable files. The first part of this chapter details experiments done using opcode based files and the second part provides details about experiments done with encrypted byte files. We start by describing the experimental-setup and scoring mechanisms, using the opcode files as examples. We then explain the experiments performed with encrypted files and encrypted short sequences in the same files and provide the results obtained from these experiments. We also experiment with various parameters in the implementation to optimize the results. The details of these optimizations are also mentioned in this chapter. These experiments are performed for opcode files and then repeated for byte files with minor modifications.

3.1 Experimental Setup

The HMM implementation takes in encrypted opcodes. It also takes in parameters such as N, M and T. If the algorithm converges or if the algorithm has run for a maximum number of iterations without converging, it outputs the training data. This training data, also known as training model is used to test files for the presence of encrypted data. In order to test the files and detect encrypted data in the files, we use the training model to generate scores for an entire file or sections of the file. Using these scores, we classify files or sections of files into encrypted data or plaintext data. We also use the scores to measure the accuracy of the results generated. In the sub-sections below, we explain the implementation of these steps in detail.

3.1.1 Generating HMM Inputs

In this section, we describe how the inputs for the HMM were generated. To start with, we created executable files from 58 C program files. We disassembled these executable files into their assembly code. The assembly code files were saved as ASM files. This was done using the command-line disassembler in NASM disassembler software program. Looking at the ASM files, we figured out that the third column in the file always contains the opcode mnemonics or instructions such as ADD, MOV, JMP, etc. So we extracted only the third columns of each file into text files. This was done using the AWK command line tool. We also extracted the list of unique opcodes and their frequencies in all the files using AWK and saved them to another text file in the sorted order. A preview of this file with the unique opcodes and their frequencies is given in Table 4. This entire process was compiled into a batch script to be run for any number of C files.

We thus generated 669,900 opcodes or observations and 150 unique opcodes to be used as part of the training. To generate encrypted opcodes, we once again used Simple Substitution and replaced each opcode by another opcode. So AND was substituted with ADD, ADD with ADC, CALL with BOUND and so on. These encrypted opcodes are known as the training data or training set. They are also the list of observations for our HMM implementation.

3.1.2 HMM Implementation

The HMM algorithm implemented and described in the previous chapter is being re-used with some changes to suit encrypted opcodes as inputs. Since we wanted to vary the number of unique inputs or opcodes used in the algorithm and optimize the HMM parameters, we modified the implementation to comprise of dynamically allocated matrices, instead of static arrays initialized using $M = 26$ and $N = \{2, 3, 4\}$. Since the arrays are being dynamically allocated, we were able to make the implementation more flexible by passing N, M, T and number of maximum iterations (used to allow the HMM model to converge) as command line arguments. This allowed

Table 4: Preview of the most frequently occurring opcodes and their frequencies, ordered in the descending order of the frequencies.

Opcode	Frequency	Relative Frequency Percentage
add	286726	42.80%
mov	55520	8.29%
pop	29362	4.38%
push	24937	3.72%
jz	20702	3.09%
imul	16818	2.51%
inc	16542	2.47%
cmp	13603	2.03%
or	13479	2.01%
nop	13258	1.98%
jmp	12129	1.81%
call	12062	1.80%
test	11936	1.78%
lea	11660	1.74%
sub	11194	1.67%
jnz	7752	1.16%
and	7520	1.12%
jc	7352	1.10%
xor	6645	0.99%
gs	6288	0.94%

us to write a Python script that would run the HMM executable (generated from the C implementation) for various combinations of parameters and made the result generation process faster.

3.1.3 Generating Training Models

We run the HMM executable for various parameters. We experimented with different values for these parameters. The values used for these parameters in these experiments are specified in the next few sections along with the results. Once the model converges, we output N, M, T, π, A and B to text files and store it, so that it can be later on loaded by the scoring script to score using the model. The data output to the text files are called training models.

3.1.4 Generating Test Files

We use executables that we did not use for the training data, to generate the test files. Just as we did with the training files, we convert these executables into their ASM files and extract out their opcodes. We generated 312 such files to be used as test data.

For experiments that use fully encrypted files, we encrypted all the opcodes in the files by using Simple Substitution on each opcode. For experiments involving plaintext files, we used the unencrypted opcodes that were extracted from the ASM files directly. For experiments involving partially encrypted files, we encrypted the opcodes in the first half of the file using Simple Substitution and used the plaintext opcodes as is in the rest of the file.

3.1.5 Scoring with Test Files

We calculate scores of test files using the alpha pass algorithm mentioned in [17]. We load parameters A, B, N, M and π for the alpha pass algorithm from one of the training models that were saved to files at the end of the HMM training. We also input the opcodes from a test file as the observations list for the alpha pass algorithm.

We compute the alpha pass value for each observation in the test file. We then compute the score of the test file by first adding the *log* of the alpha pass value for all the observations. The actual score is then computed by negating this summed value. However, this score is not normalized. This score depends on the number of observations in the test files and could vary based on that. In order to normalize it, we divide the negated score using the number of observations in the test file. This normalized score is referred to as the score of the test file in the rest of this project.

Normalized scores for encrypted files tend to be nearly equal. Similarly, normalized score for plaintext files will tend to be equal. We will see these results and conclusions in the next few sections.

For experiments where our aim was to detect encrypted files and plaintext files, we ran the scoring algorithm described above for each file. However, for experiments where our aim was to detect encrypted short sequences in partially encrypted files, we had to make small modifications to the scoring mechanism. Instead of generating a score for an entire file, we split the file into sections and generate a score for each section of the file. Thus if a particular section is encrypted, our model will able able to generate a score that is in line with that of an encrypted section. Each of these sections were fixed in length for a given experiment. We also experimented with various lengths for these sections to figure out the optimal length.

3.1.6 Generating Scatter Plots

We obtain the scores for all the files or all file sections for a particular training model using the scoring algorithm. In order to visualize the scores, we plot the scores on a scatter plot. As mentioned before, score values of encrypted files/sections tend to be closer to each other than the scores of plaintext files/sections. The scores of plaintext files/sections also tend to be closer to each other than the scores of encrypted files/sections. Hence, we see a clear separation between the scores of encrypted files/sections and scores of plaintext files/sections on the scatter plot.

3.1.7 Measuring Accuracy of Models

In order to measure the accuracy of our model, we generate a label for each file/section. The label for a file/section is 1 if the file is encrypted and 0 if the file/section is not encrypted. While these labels represent the ideal classification for the test files/sections, the scores for the test files/sections represent the classification generated by our algorithm. For a given training model, we input the list of labels and list of scores for all the test files/sections to a function that will generate the Receiver Operating Characteristic (ROC) curve. This ROC function also provides the True Positive Rate (TPR) values, False Positive Rate (FPR) values and threshold values used to plot the ROC curve. The Area Under Curve (AUC) is the measure of accuracy of our models. The AUC is a value between 0 and 1, where 1 represents 100% accuracy and 0 represents 0% accuracy.

3.2 Language and Performance Experiments

We implemented the HMM algorithm in two languages, namely Python and C. We realized that the Python version of the HMM algorithm was very slow compared to the C implementation of the same algorithm. On analysis, we realized that the HMM algorithm uses large arrays. Python tries to optimize arrays for each iteration in the algorithm and this additional proactive optimization accumulates over a period of time and leads to large delays, even for small input sizes. For the experiments described below, the Python version took more than one hour or two, while the C version took less than two minutes. We hence decided to use the C version of the algorithm for all our applications.

In addition, we also realized printing debug statements to the output console causes delays in the C version of the code. Hence once the algorithm code was tested and completed, all the debug statements were removed to make the HMM algorithm run faster.

3.3 Experimenting with Maximum Number of Iterations

We began the HMM training step with a maximum of 200 iterations. The model did not converge for 200 iterations. We slowly increased the maximum number of iterations by intervals of 100 and found that the model converges at the end of 1000 iterations. We thus fixed 1000 as the maximum number of iterations for all our experiments.

3.4 Experimenting with T

We began training the HMM with all the training data generated. This made the number of observations or $T = 669{,}901$ and $M = 150$. Since the number of observations was so large, the HMM took a long time to converge. T being large also resulted in a highly sparse B matrix. We thus experimented by lowering T all the way to 500.

Setting T greater than 100,000 increased the sparseness of the training models, due to increase in number of observations with low frequencies. We thus decided to fix T at a maximum of 100,000. Setting T to values less than 100,000 did not provide sufficient data for the HMM to converge in most cases. In some cases, it resulted in very low values for the B matrix. So instead of lowering T, we decided to vary M to further reduce the spareness. The results of varying M is provided in the next section. It is important to note that T could be less than 100,000 in the experiments that we describe in the next few sections, if M is reduced.

3.5 Testing with Encrypted Opcode Files and Plaintext Files

The primary aim of experiments in this section is to be able to detect encrypted files and plaintext files. In order to do this and find the optimal HMM parameters for this use case, we generated various training models by varying N and M. We then scored all the encrypted files and plaintext files for each training model.

3.5.1 Experimenting with N, M

We experimented with $N = \{2, 3\}$ and $M = \{20, 25, 30, 35, 40, 50\}$. As shown in Table B.7, the AUC values for all training models were found to be 1. Appendix B shows the corresponding TPR, FPR, threshold values used to create the ROC curve. Thus, all the training models were able to separate out the encrypted files and plaintext files perfectly.

Since the results were similar for all the training models, we only present the graphical results for $N = 2$, $M = 30$ training model. Figure 12 shows the scatter plot of the scores generated using this training model. The scores plotted in red color are the scores of encrypted files. The scores plotted in blue color are the scores of plaintext files. To make the comparison easier to understand, we have plotted the scores of encrypted files and the scores of the plaintext files from which they were generated side by side. Figure 13 shows the ROC curve for this training model.

3.6 Testing with Partially Encrypted Opcode Files

The primary aim of experiments in this section is to be able to detect encrypted sections and plaintext sections in a partially encrypted file. As mentioned before, we experiment with various window sizes or section lengths to figure out the optimal window size that can efficiently separate out the encrypted and plaintext sections. We score each partially encrypted file using these window sizes. As part of our experiments, we also score for varying overlapping lengths between the windows. For

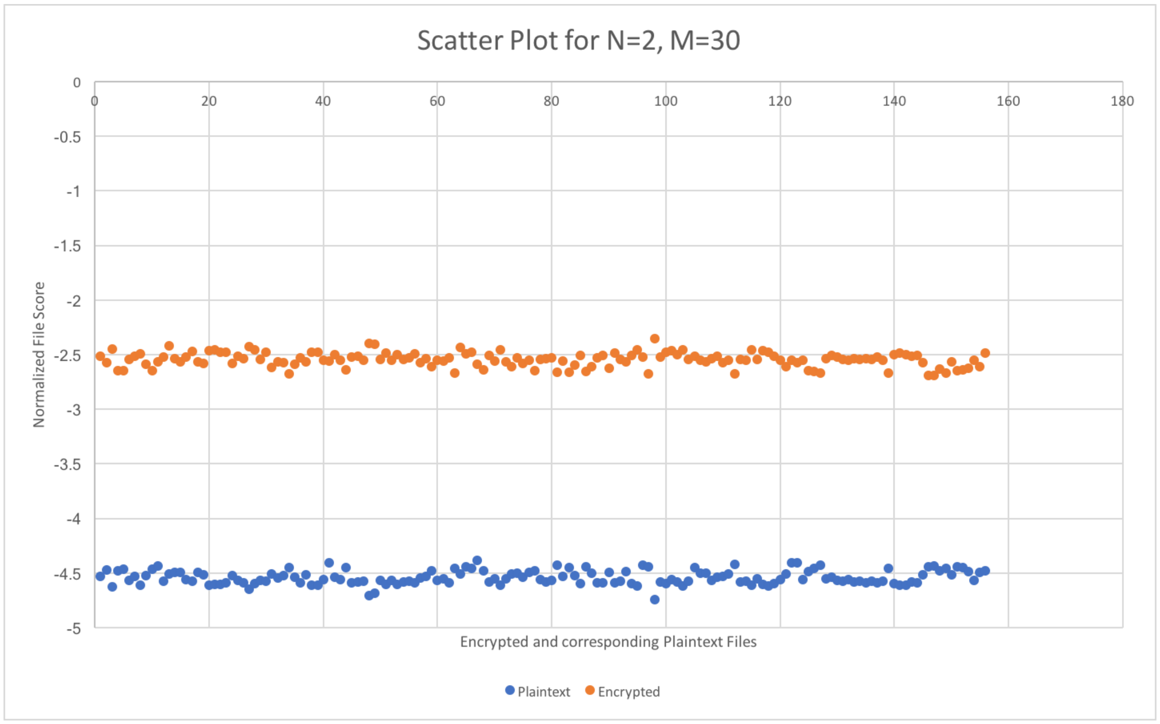


Figure 12: Scatter plot for scores of encrypted files and corresponding plaintext files generated using $N = 2$, $M = 30$ training model.

all these experiments we used the training model generated using $N = 2$, $M = 30$ as the parameters.

3.6.1 Experimenting with Various Window Sizes

We experimented with the following window sizes: {100, 150, 200, 300, 400, 500, 700, 800, 900, 1000, 1200, 1300, 1500, 1700, 1850, 2000, 2100, 2200, 2400, 2500, 2600, 2700, 2900, 3000, 3200, 3400}. The AUC scores generated for these window lengths are shown in Figure 14 and the values for the same provided in B. The corresponding ROC Curves are presented in Figure 15. As we can see from the AUC scores and ROC Curves, when the window length is small, there is insufficient data for our model to classify a section as encrypted or plaintext. Thus, the accuracy is low for small window lengths. As we increase the window length, we see the accuracy of detection reaching 100%. Window sizes between 1850 and 2200 provide the maximum AUC of

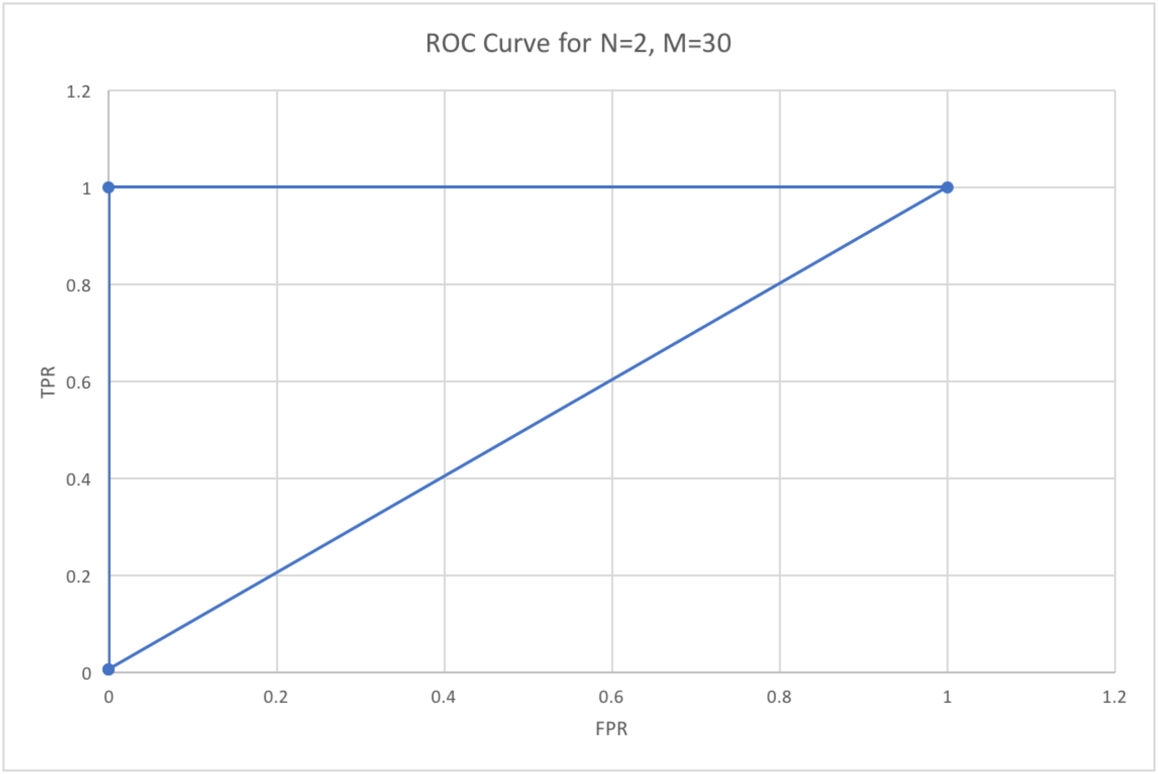


Figure 13: ROC curve for encrypted files and plaintext files generated using $N = 2$, $M = 30$ training model.

1. Beyond 2200, the window lengths become too large and the number of windows in a file reduces to a relatively small value. As a result, very large window sizes are not able to reliably detect encrypted sections in all types of scenarios. The scatter plots for the windows re-iterate the same result. The scatter plots for window sizes = {800, 1300, 1850, 2200} are shown in Figure 16, Figure 17, Figure 18 and Figure 19 respectively. The scatter plots for the remaining window lengths are given in Appendix A.

From the scatter plots, we observed that the encrypted sections have higher scores than the unencrypted sections. Among the encrypted sections, we saw two levels of scores. The higher scores were the majority among the encrypted windows. The lower scores were not as low the unencrypted section scores, but were slightly lower than the scores of the other encrypted sections. On closer inspection, we found that the lower

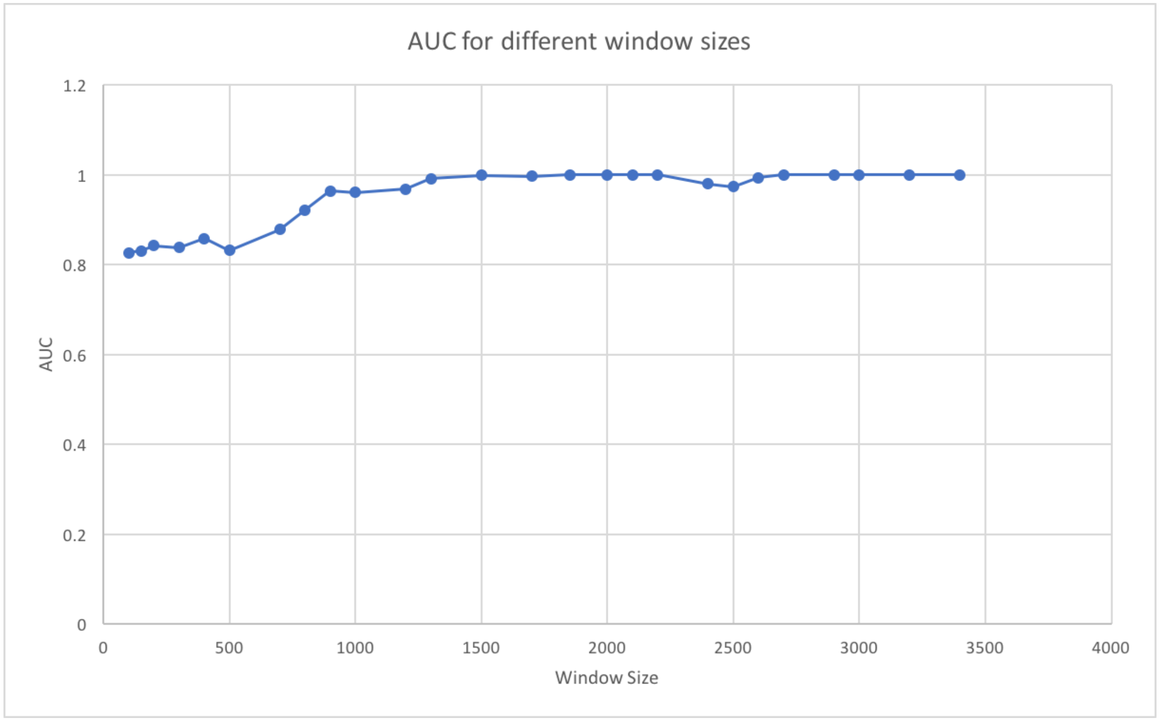


Figure 14: AUC values for all window sizes.

scores among the encrypted sections only corresponded to the sections containing the first 550 to 750 opcodes from each file. We checked the disassembled code for these files using OllyDbg and confirmed that these first set of opcodes that result in the lower level of scores for all files and all window sizes correspond to system functions. These system functions result in a large number of ADD and NOP instructions. This in turn leads to a lower score value for the windows with the opcodes of the system functions. When the window gets as large as 2200, we see that most of the files have just one encrypted window. Since the number of non-system function opcodes outweigh the number of system function opcodes in these large windows, we see only one level of score emerging. Whereas for lower window sizes, the system function opcodes tend to occupy more number of encrypted windows. This results in a scatter plot that shows two layers for the encrypted windows - one layer that corresponds to the encrypted system function opcodes and one that corresponds to the remaining

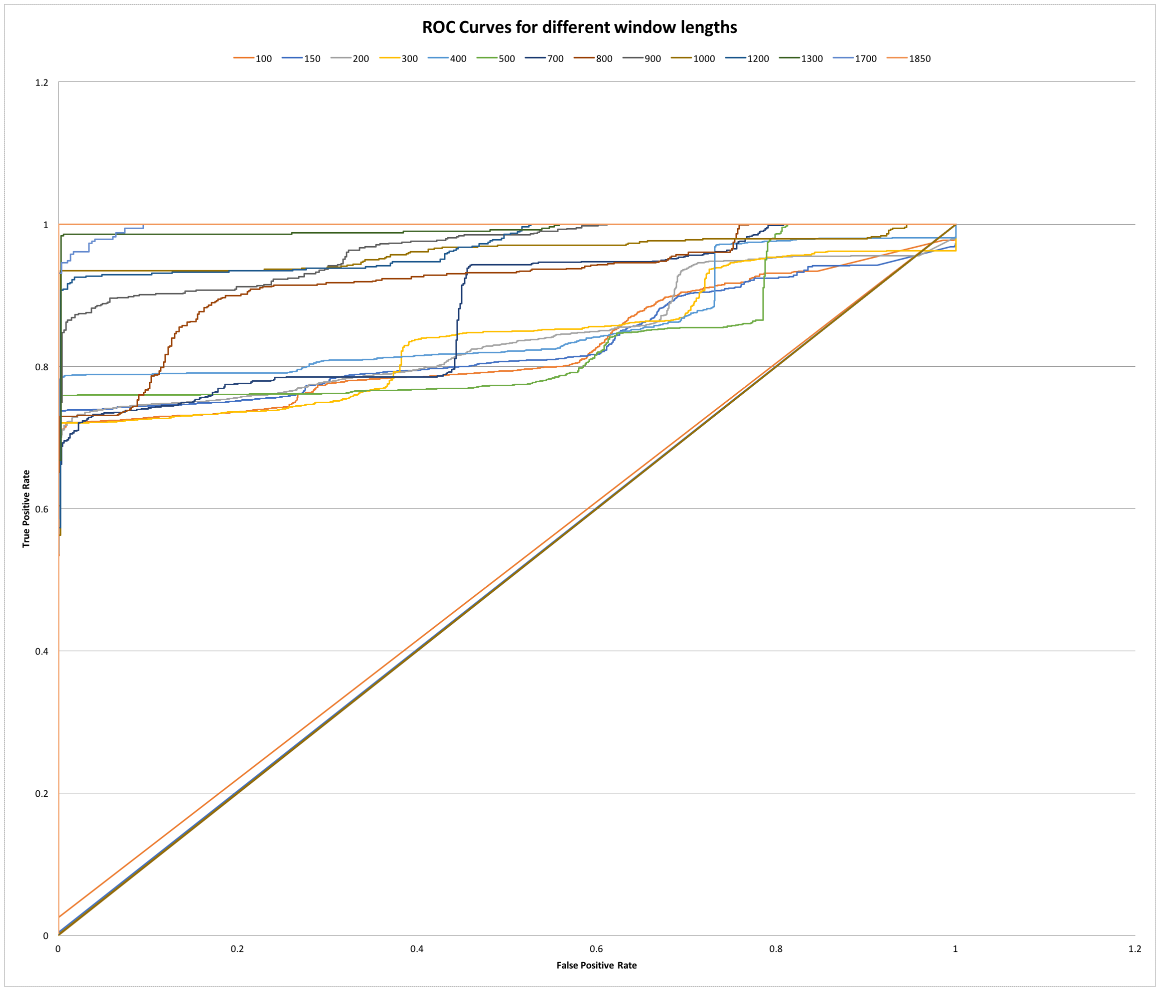


Figure 15: ROC curves for different window lengths with $N = 2$, $M = 30$.

encrypted opcodes.

When we split the opcodes in the test files into these windows, we also realized that the last window always had much lesser number of opcodes compared to the window length. Due to lack of sufficient number of opcodes, the HMM was unable to generate a good score for the last window. We thus decided to ignore the last score of every file from our calculations.

We also expected issues with classification of the middle window of a file with partially encrypted and partially plaintext data. Initially we obtained results by

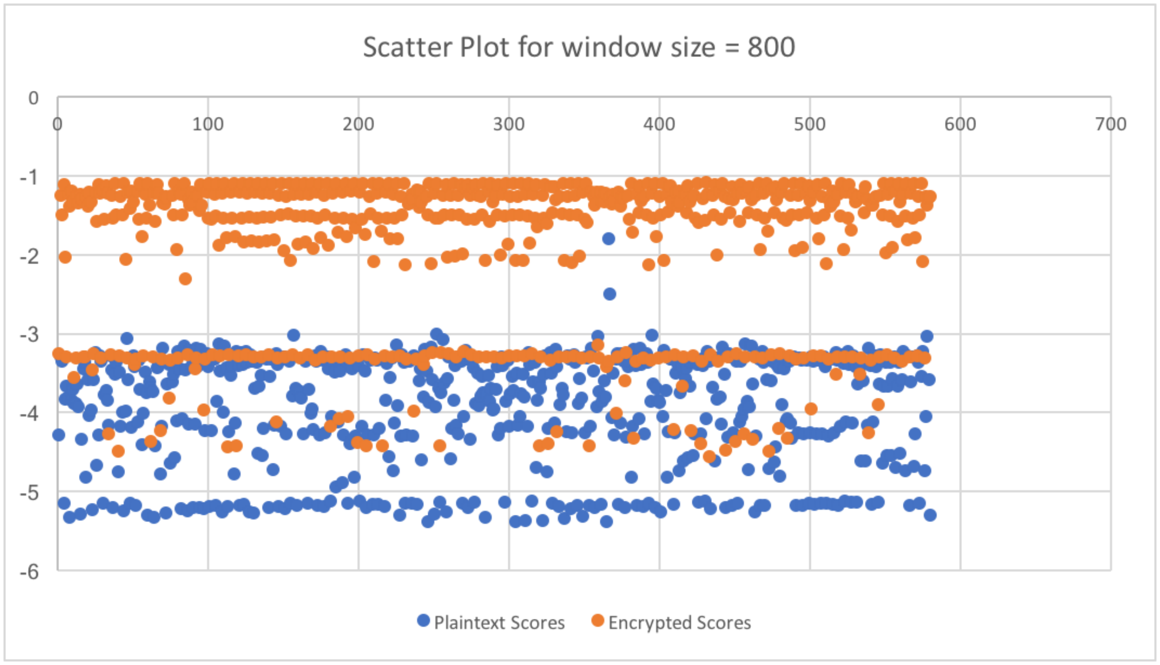


Figure 16: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 800.

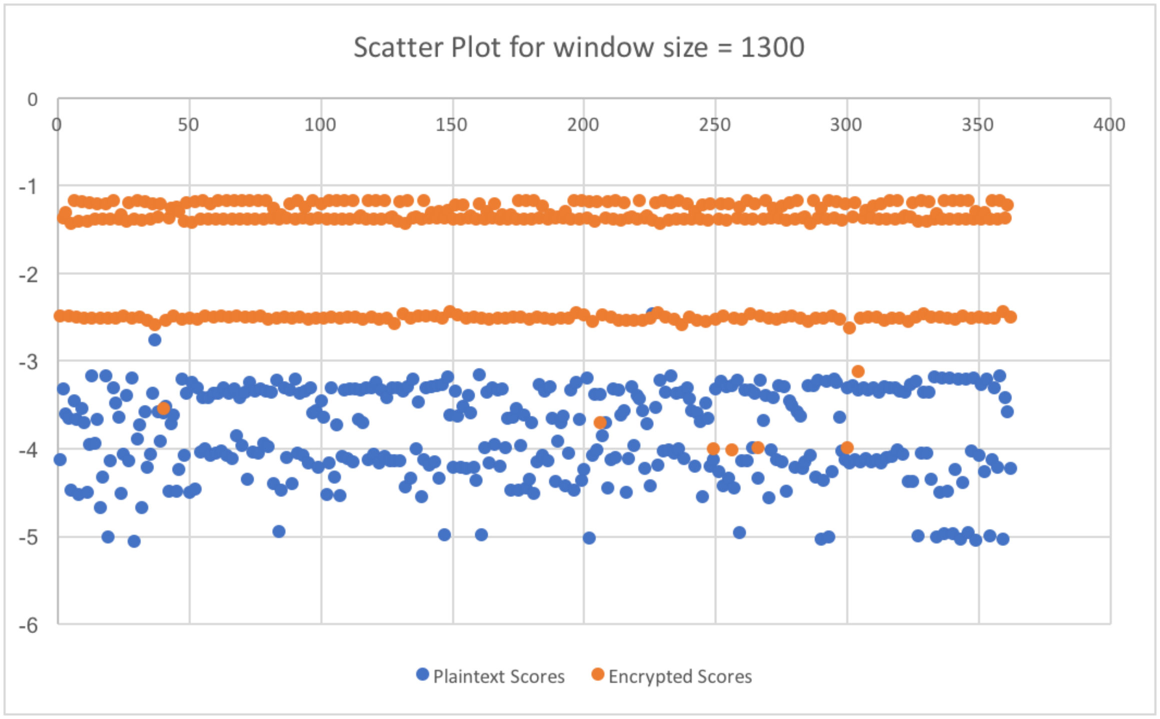


Figure 17: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1300.

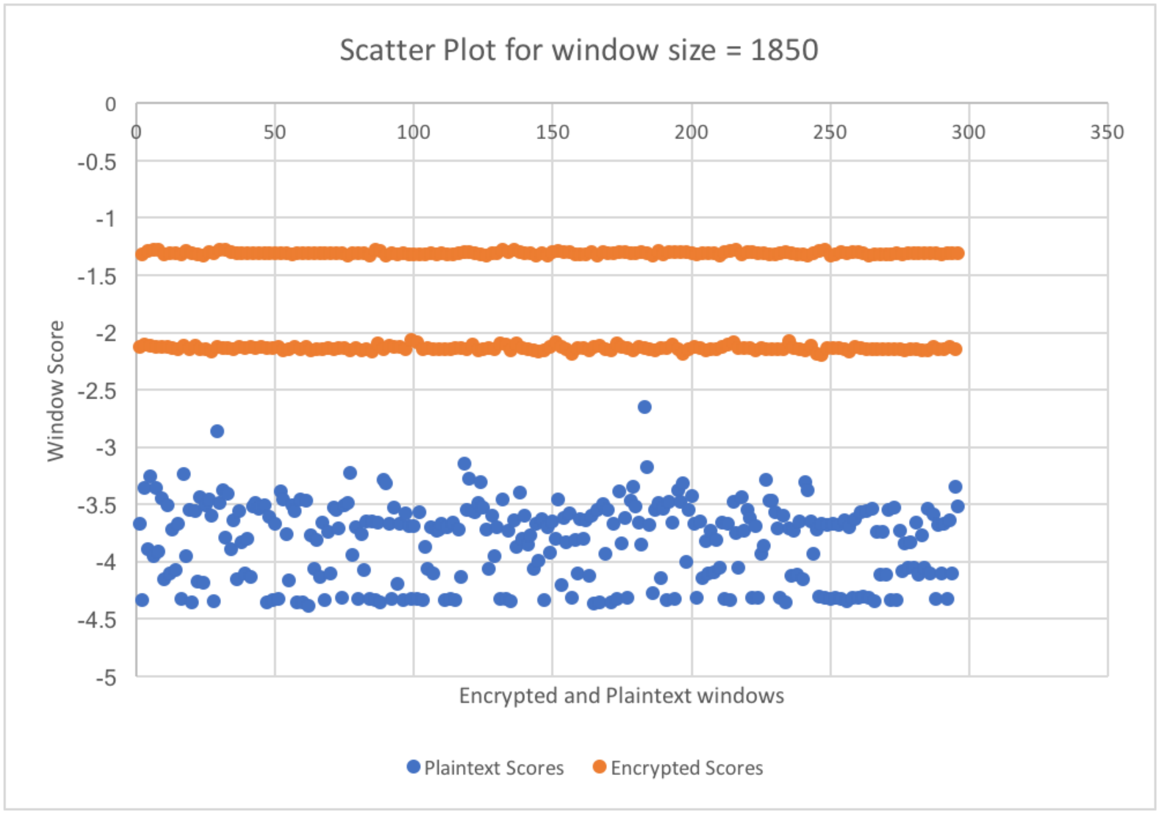


Figure 18: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1850.

labeling this section as an unencrypted section. In most cases, the HMM too classified this window as an unencrypted section. In order to improve the accuracy further, we tried to compute the percentage of encrypted data in the window and labeling it as encrypted if the percentage was greater than 50%. However, this resulted in no significant improvement to the AUC scores. The AUC scores remained the same for all the window lengths.

In this section, we experimented with only one parameter - window lengths. For any pair of consecutive windows, we assumed that the first opcode of the second window is the opcode right after the last opcode of the first window. There is no overlap between any two windows. In the next section, we present the results for experiments conducted by varying overlapping lengths between windows.

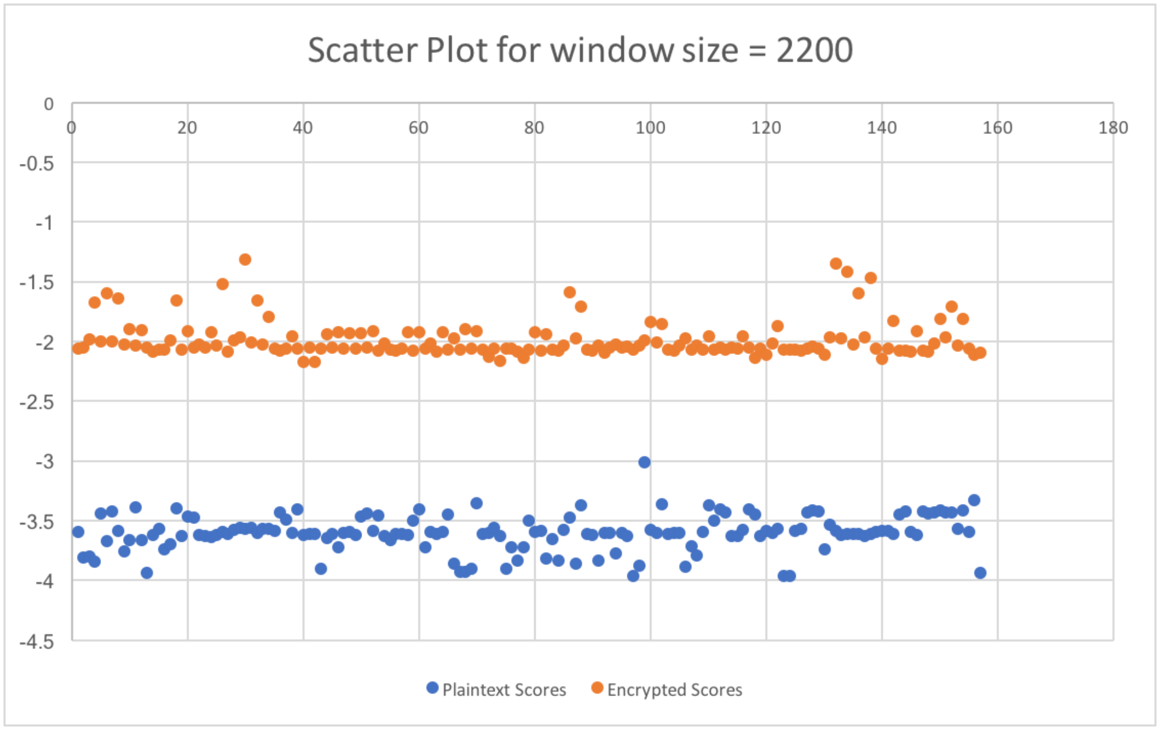


Figure 19: Scatter plot of scores with $N = 2$, $M = 30$ and window size $= 2200$.

3.6.2 Experimenting with Various Window Sizes and Window Sliding Lengths

We define the extend of the overlap between any two windows to be the window sliding length. In all experiments executed in the previous section, we had set the window sliding length to be zero. In this section, we experiment with various values for the sliding length and repeat the experiments using various window lengths again. Figure 20 shows an example of how a file is split into windows for experiments where the sliding length is zero and compares it with windows generated using non-zero window sliding lengths. It shows how windows can overlap in the case of variable sliding lengths.

The experiments in this section use two experimental parameters - window sizes and window sliding lengths. We set the window sliding length to one of the following values: $\{50, 100, 150, 200\}$. For each window sliding length value, we experiment

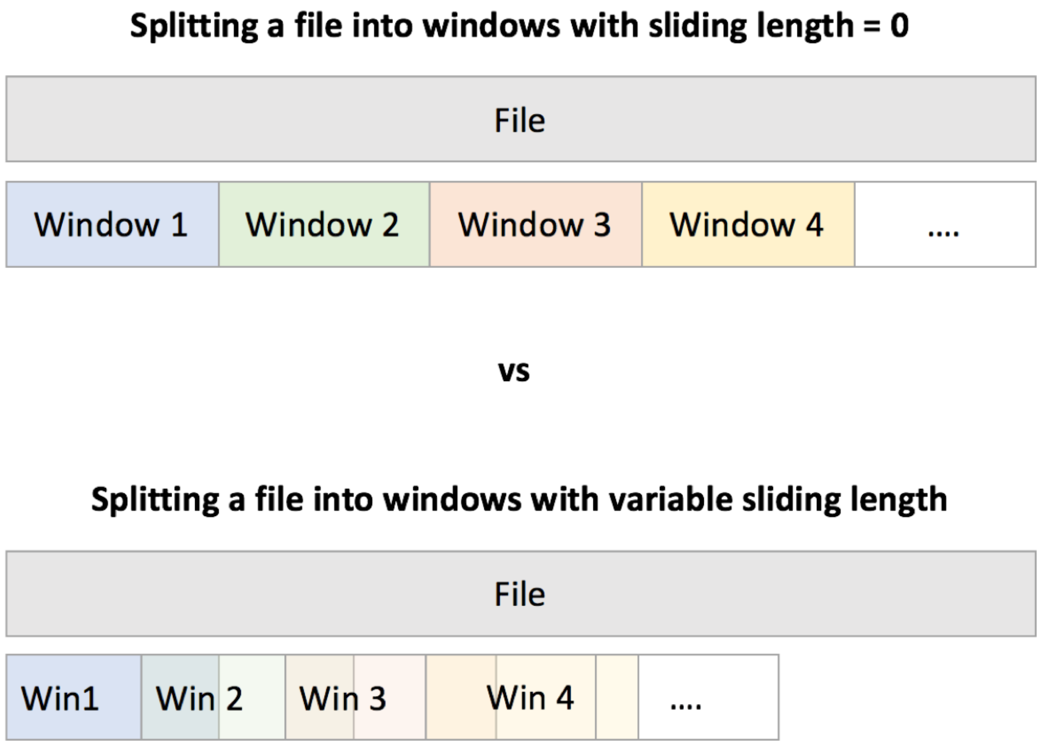

Splitting a file into windows with sliding length = 0
File
Window 1
Window 2
Window 3
Window 4
....
vs
Splitting a file into windows with variable sliding length
File
Win1
Win 2
Win 3
Win 4
....

Figure 20: Splitting a file into windows of fixed sliding lengths and variable sliding lengths.

with each of these window sizes: {300, 400, 500, 700, 800, 900, 1000, 1200, 1300, 1500, 1700, 1850, 2000, 2100, 2200, 2400, 2500, 2600, 2700, 2900, 3000, 3200, 3400}. Figure 21 shows the AUC curves plotted for these window sliding length and window size combinations. Table 5 shows the corresponding AUC values for these curves.

All window sliding length results had a minimum AUC value of 0.8. All AUC curves were very similar to each other. With increase in window size, the AUC values too increased until a certain point. They had a small decline in AUC when we increased the window size further. The ideal window length range was between 1700 and 2100 (inclusive of 1700 and 2100). For window sliding lengths of {50, 100, 150, 200}, the maximum AUC values were at window sizes of {1700, 1850, 2100,

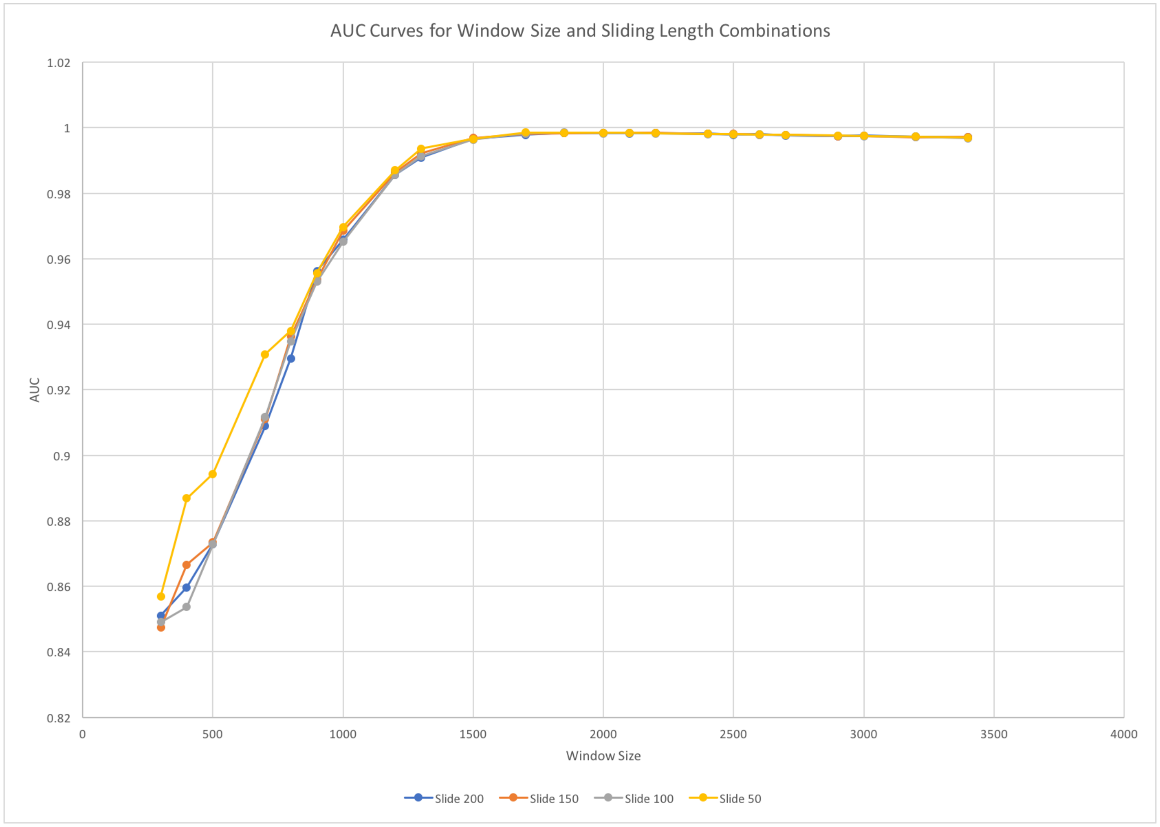


Figure 21: AUC curves for different window sizes and sliding lengths

1850} respectively. Sliding length of 50 performed slightly better than other sliding length values for window sizes less than 1000. Although all the sliding windows had a maximum AUC value that was greater than 0.998, none of the window sliding lengths that we experimented with in this section were able to give us a perfect AUC of 1.0.

3.7 Testing with Encrypted Byte Files and Plaintext Files

In this section, we describe experiments run using bytes in a file, instead of opcodes. The aim of these experiments is to detect encrypted byte files from plaintext byte files. In order to test with bytes of an executable file, we took the 58 C program training files and generated executables using them. We then encrypted each byte in these executable files using the XOR encryption with a fixed key value of 0x77. The encrypted data bytes from the 58 executable files were used as the training data for

Table 5: AUC values for different combinations of window sizes and window sliding lengths.

Window Length	Slide = 200	Slide = 150	Slide = 100	Slide = 50
300	0.85099	0.84736	0.84902	0.85694
400	0.85961	0.86656	0.85366	0.88686
500	0.87283	0.87336	0.87275	0.89425
700	0.9089	0.91098	0.91158	0.9308
800	0.92954	0.93621	0.93486	0.93794
900	0.95628	0.95363	0.953	0.9556
1000	0.96585	0.96854	0.96518	0.96965
1200	0.98563	0.98636	0.98567	0.98706
1300	0.99095	0.99214	0.99129	0.99358
1500	0.99672	0.9969	0.99639	0.99661
1700	0.99785	0.99818	0.99821	0.99853
1850	0.99851	0.99835	0.99859	0.99851
2000	0.99841	0.99831	0.99821	0.99843
2100	0.99825	0.99845	0.99833	0.99841
2200	0.99836	0.99844	0.99827	0.99836
2400	0.9982	0.99818	0.99807	0.99816
2500	0.99788	0.99816	0.99794	0.99803
2600	0.99802	0.99779	0.99786	0.99794
2700	0.99763	0.99787	0.9977	0.99781
2900	0.99742	0.99742	0.9975	0.9976
3000	0.99758	0.99754	0.99738	0.99747
3200	0.99735	0.99709	0.99716	0.99726
3400	0.99709	0.99719	0.99682	0.99694

the HMM. Using $T = 10,000$, we input encrypted bytes with 246 unique values to the HMM algorithm. Given that the HMM algorithm took less than four minutes to generate training models for M=246, we decided to use M=246 instead of varying M. Using $N = \{2, 3\}$, we generated two training models to be used for scoring and testing encrypted and plaintext byte files. The executables used for training the HMM and those used for scoring were completely different datasets.

We scored 312 test files comprising of both encrypted and plaintext byte data. We then generated the scatter plots for these experiments using the scores generated for these files. The scatter plot generated for $N = 2$ is shown in Figure 22. The scatter plot for $N = 3$ is shown in Figure 23. The red dots in the scatter plots represent encrypted byte files and the blue dots denote plaintext byte files. As we can see from the scatter plots, our approach was able to successfully separate the encrypted and plaintext byte files, in the same way that it was able to separate out the encrypted and plaintext opcode files. The resulting AUC value for all the experiments described in this section was 1.0.

3.8 Discussion

In this chapter, we saw that our HMM training models along with the alpha pass scoring mechanism can efficiently classify encrypted files and plaintext files. We also saw that if the file is partially encrypted, our model is able to accurately differentiate between encrypted sections and plaintext sections by using window scoring mechanisms. We plotted scatter plots, ROC curves and AUC graphs to verify our results. The next chapter provides a conclusion for this thesis by summarizing the results obtained and providing details on possible extensions for this project.

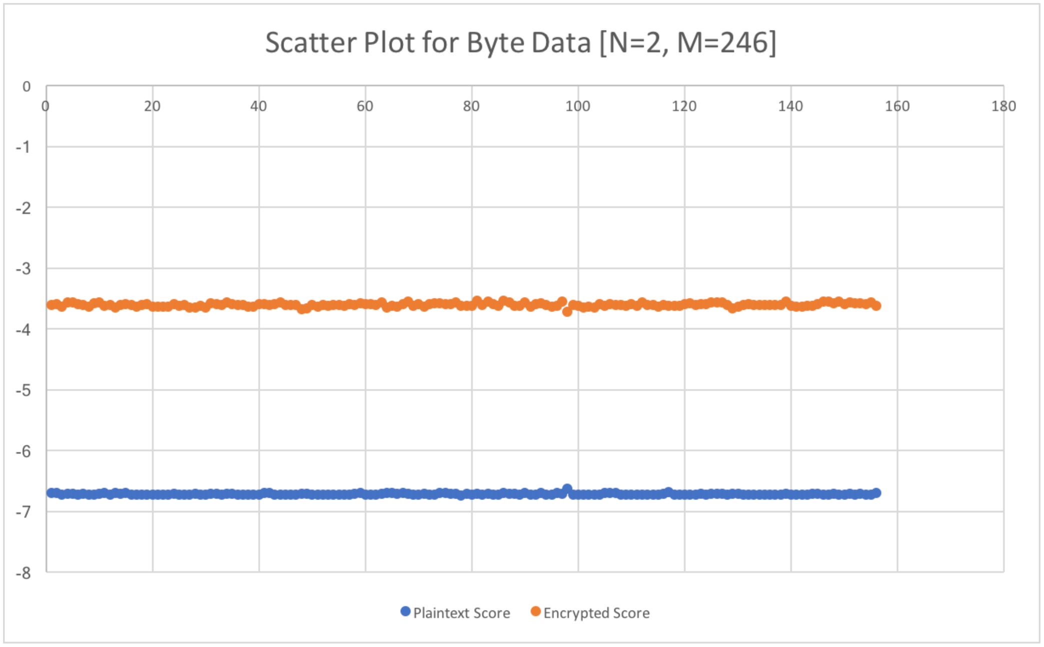


Figure 22: Scatter plot for scores of encrypted byte files and corresponding plaintext byte files generated using $N = 2$, $M = 246$ training model.

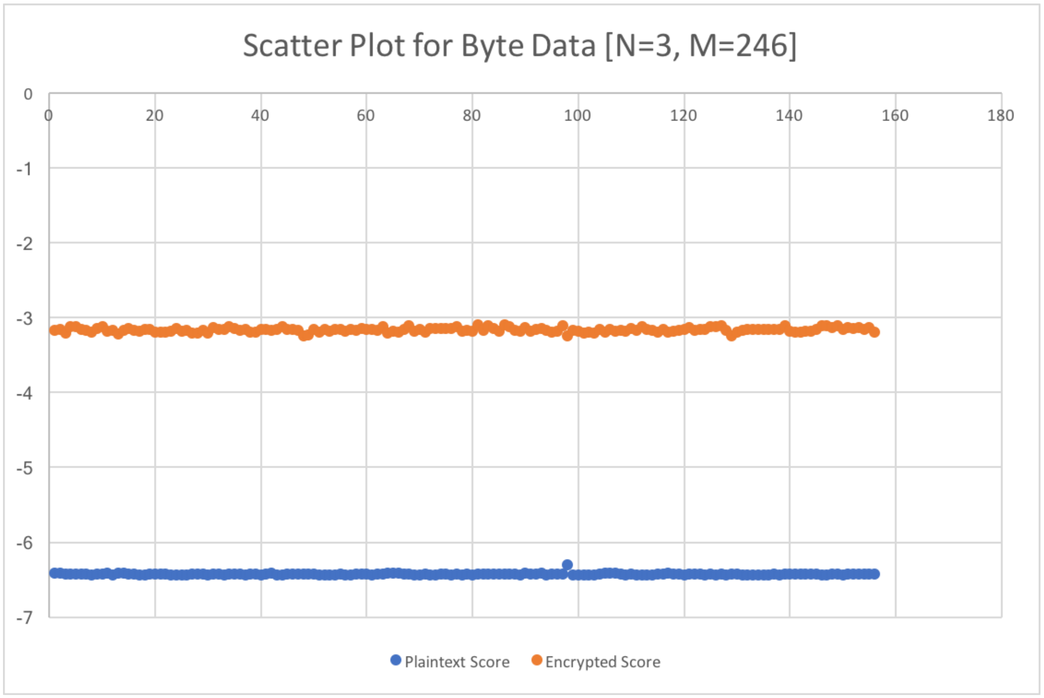


Figure 23: Scatter plot for scores of encrypted byte files and corresponding plaintext byte files generated using $N = 3$, $M = 246$ training model.

CHAPTER 4

Conclusion and Future Work

In this project, we implemented a hidden Markov model to detect encrypted files and encrypted sections in files. For detecting encrypted files from plaintext files, we trained the HMMs using encrypted opcodes from the encrypted files. We generated various training models by varying all the HMM parameters. We used these training models to score encrypted and plaintext files. We plotted the scores generated on scatter plots for each training model and found that there was a clear separation of scores between encrypted files/sections and plaintext files. We measured the accuracy of the scores using ROC curves and AUC values and thus concluded, this method had an 100% accuracy in detecting encrypted files among a mixture of encrypted and plaintext files for all the training models used. We repeated these experiments using encrypted bytes files as training and test data and similarly concluded that our method had 100% accuracy for byte files too.

As for detecting encrypted sections in partially encrypted files, we used the same training models to generate scores for sections of each file. We varied the length of these scored sections to identify the optimal window size to be used for scoring. We also varied the overlap between the scored sections to understand how the overlap of the window lengths affects the results. We figured out that for moderate window lengths of 1850-2200, our model is able to accurately detect the encrypted sections. We also generated results to prove that lower window overlap lengths provided better results for lower window sizes. We also saw that for any window size, the accuracy of our model was between 80%-100%.

As a future extension to this project, we could adapt the same window scoring technique used for opcodes to score windows of encrypted bytes and thus detect encrypted sections in partially encrypted byte data. Applying sliding windows that overlap with each other and windows of different lengths to byte data could provide interesting results. Other extensions to this project could include detecting the key used to encrypt files or sections using HMMs or cryptographic statistical attacks.

To conclude, we were successfully able to classify encrypted code files and plaintext code files using hidden Markov models. We were also able to detect encrypted sections in code files and differentiate them from the plaintext sections in same files.

LIST OF REFERENCES

[1] V. Zwanger, E. Gerhards-Padilla, and M. Meier, "Codescanner: Detecting (hidden) x86/x64 code in arbitrary files," in *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on Malicious and Unwanted Software.* IEEE, 2014, pp. 118–127.

[2] B. Swain. Symantec. "What are malware, viruses, spyware, and cookies, and what differentiates them?" 2009. [Online]. Available: https://www.symantec.com/connect/articles/what-are-malware-viruses-spyware-and-cookies-and-what-differentiates-them

[3] C. Nachenberg, "Understanding and managing polymorphic viruses." in *The Symantec Enterprise Papers*, 1996, vol. XXX. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/understanding-and-managing-polymorphic-viruses-96-en.pdf

[4] M. Stamp, *Information Security: Principles and Practice.* John Wiley & Sons, 2011.

[5] "Computer knowledge." 2013. [Online]. Available: http://www.cknow.com/cms/vtutor/types-of-viruses.html

[6] "History of computer viruses." February 2013. [Online]. Available: http://www.cknow.com/cms/vtutor/robert-slade-computer-virus-history.html

[7] "Brain (computer virus)." [Online]. Available: https://en.wikipedia.org/wiki/Brain_(computer_virus)

[8] P. Szor, *The Art of Computer Virus Research and Defense.* Pearson Education, 2005. [Online]. Available: https://books.google.com/books?id=XE-ddYF6uhYC

[9] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: from encryption to metamorphism," *International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74–83, 2012.

[10] X. Li, P. K. K. Loh, and F. Tan, "Mechanisms of polymorphic and metamorphic viruses," in *2011 European Intelligence and Security Informatics Conference*, Sept 2011, pp. 149–154.

[11] Symantec. "Security 1:1 - part 1 - viruses and worms." 2013. [Online]. Available: https://www.symantec.com/connect/articles/security-11-part-1-viruses-and-worms

[12] "Viruses that can cost you." [Online]. Available: http://www.symantec.com/region/reg_eu/resources/virus_cost.html

[13] S. Venkatachalam. "Detecting undetectable computer viruses." 2010. [Online]. Available: http://scholarworks.sjsu.edu/etd_projects/156/

[14] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, 2001, pp. 38–49.

[15] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *Inf. Secur. Tech. Rep.*, vol. 14, no. 1, pp. 16–29, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.istr.2009.03.003

[16] "Markov model." January 2017. [Online]. Available: https://en.wikipedia.org/wiki/Markov_model

[17] M. Stamp, "A revealing introduction to hidden Markov models," https://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf, 2004.

[18] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[19] R. L. Cave and L. P. Neuwirth, "Hidden markov models for english," in *Hidden Markov Models for Speech*, J. D. Ferguson, Ed., October 1980.

[20] "The brown corpus of standard american english." February. [Online]. Available: http://www.cs.toronto.edu/~gpenn/csc401/a1res.html

[21] A. Shamir and N. Van Someren, "Playing 'hide and seek' with stored keys," in *International conference on financial cryptography*. Springer, 1999, pp. 118–124.

[22] G. Shanmugam, R. M. Low, and M. Stamp, "Simple substitution distance and metamorphic detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 3, pp. 159–170, 2013.

[23] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[24] N. Ganesh, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "Static analysis of malicious java applets," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, ser. IWSPA '16. New York, NY, USA: ACM, 2016, pp. 58–63. [Online]. Available: http://doi.acm.org/10.1145/2875475.2875477

[25] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, Dec 2006. [Online]. Available: https://doi.org/10.1007/s11416-006-0028-7

APPENDIX A

Appendix A: Scatter Plots of Scores for Different Window Lengths

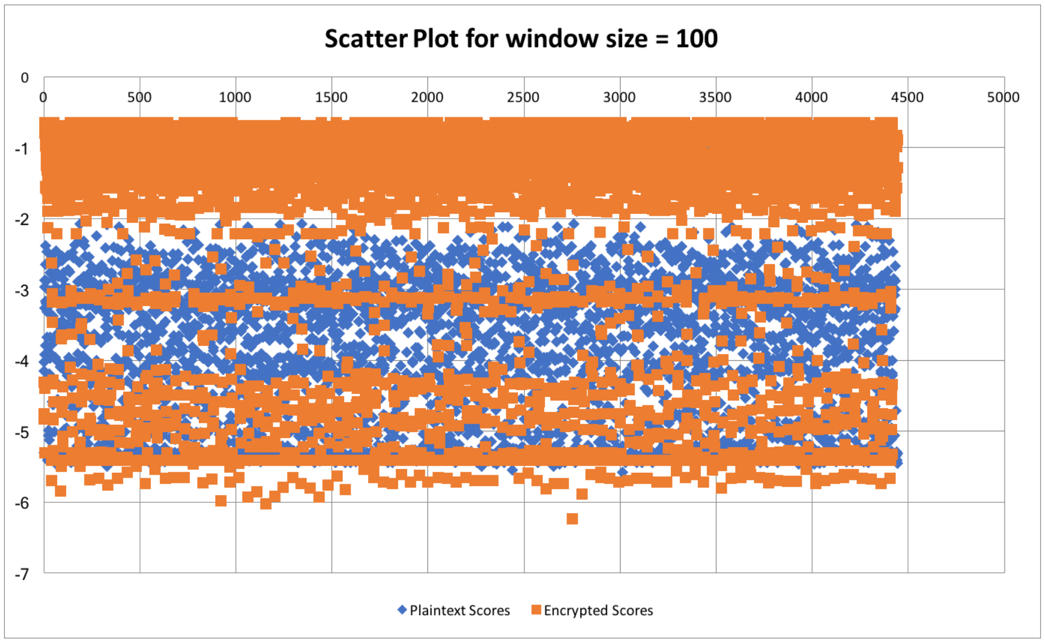


Figure A.24: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 100.

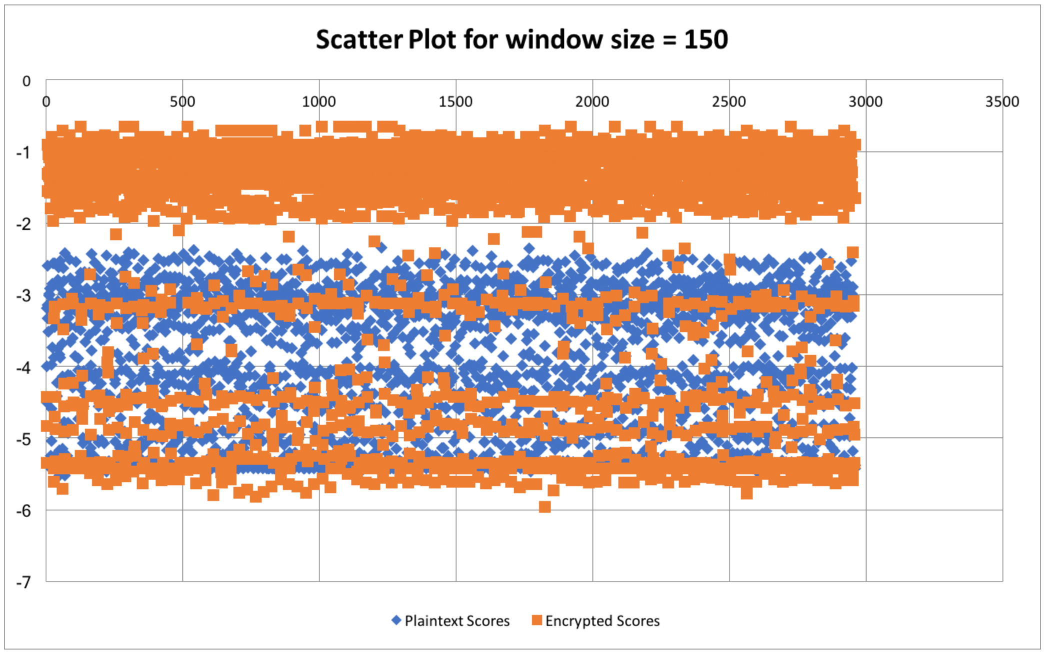


Figure A.25: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 150.

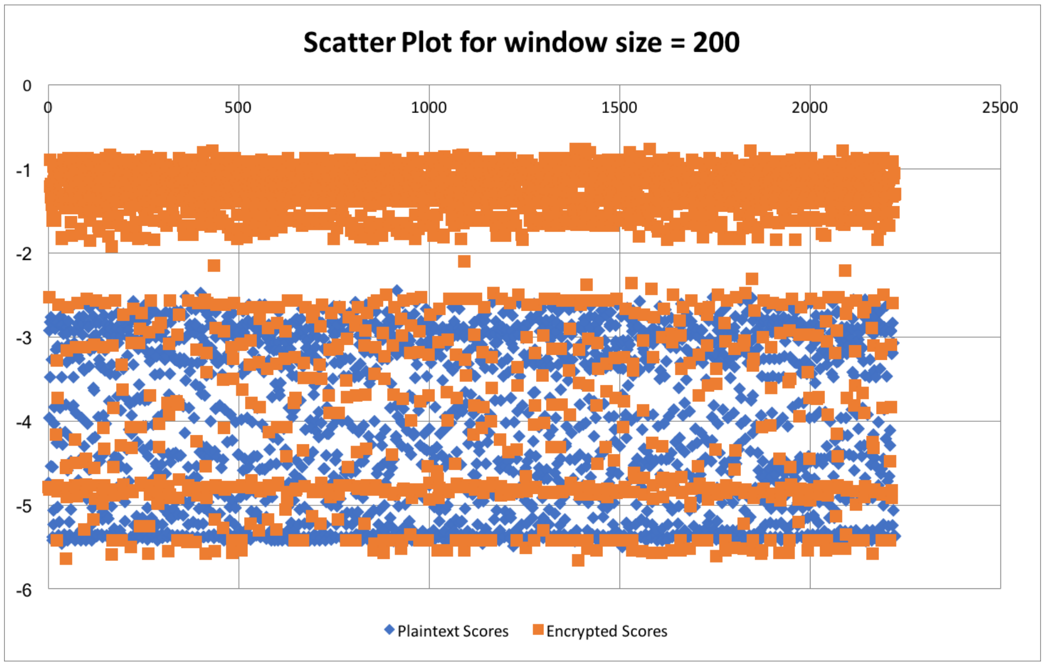


Figure A.26: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 200.

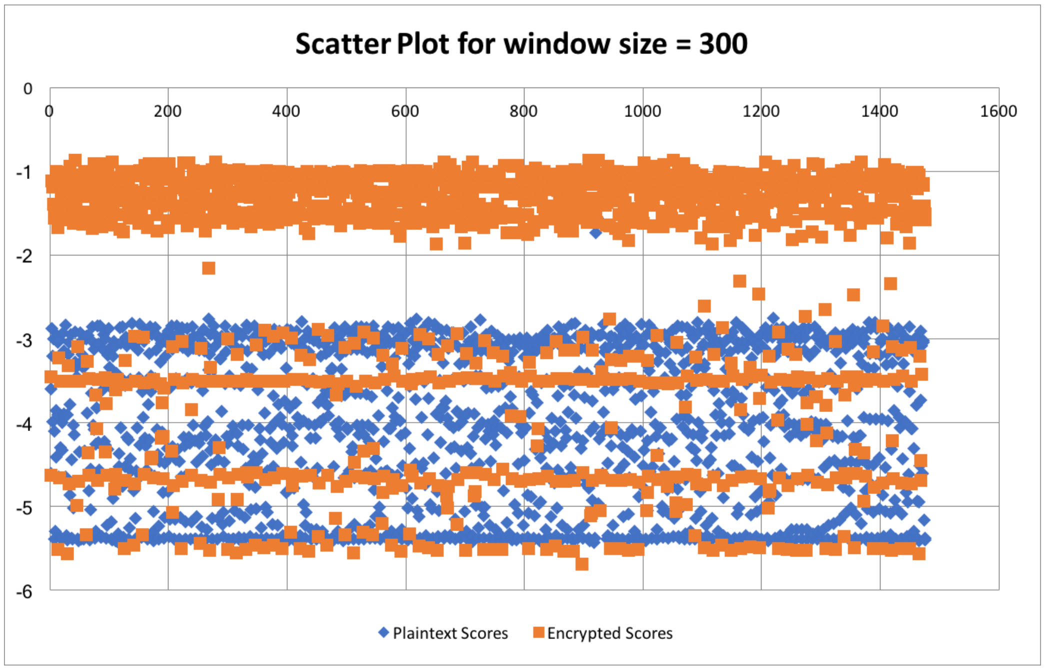


Figure A.27: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 300.

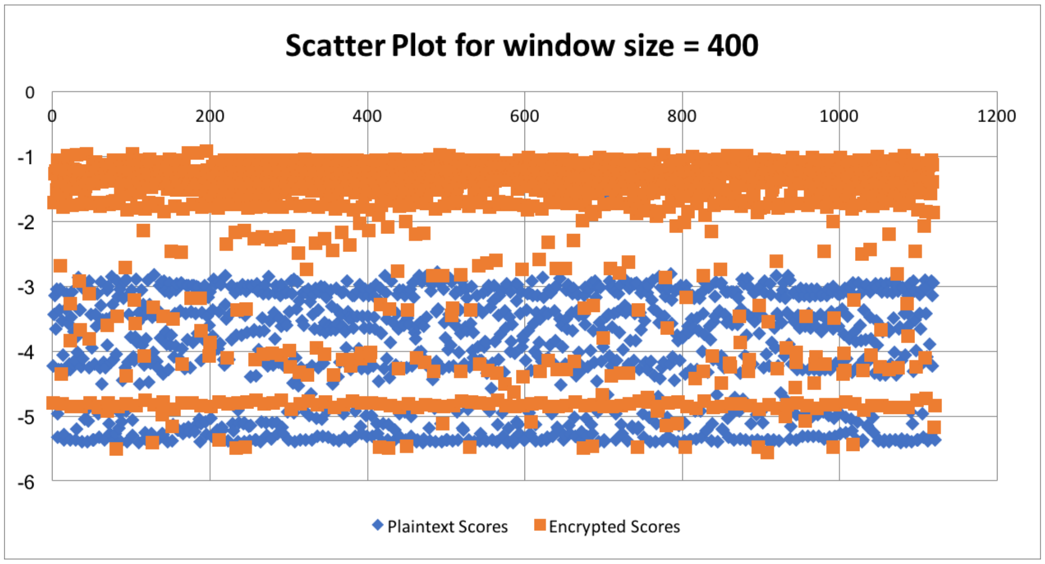


Figure A.28: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 400.

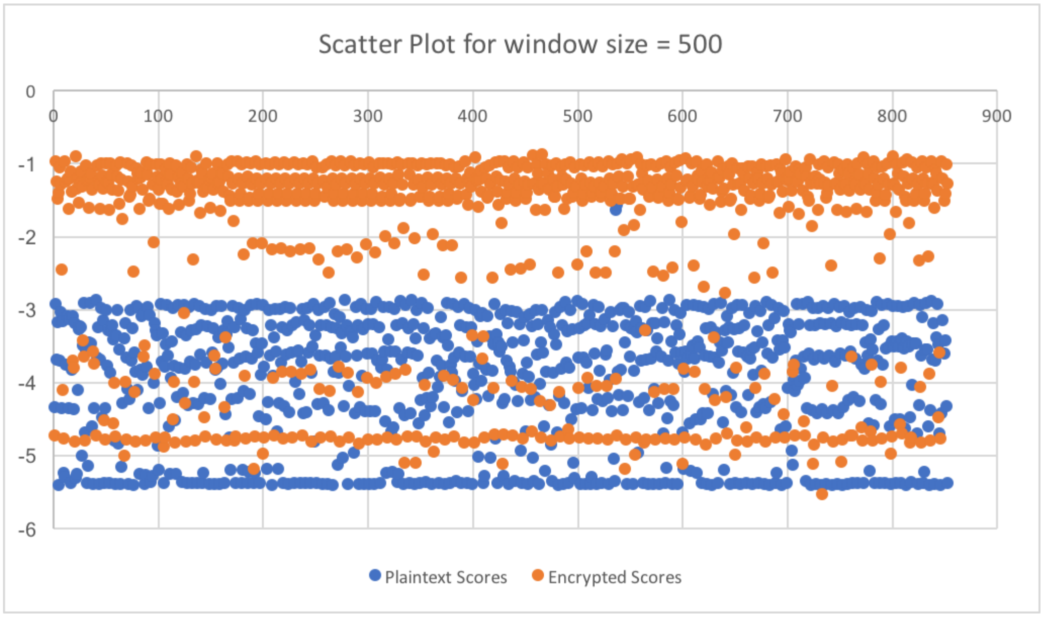


Figure A.29: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 500.

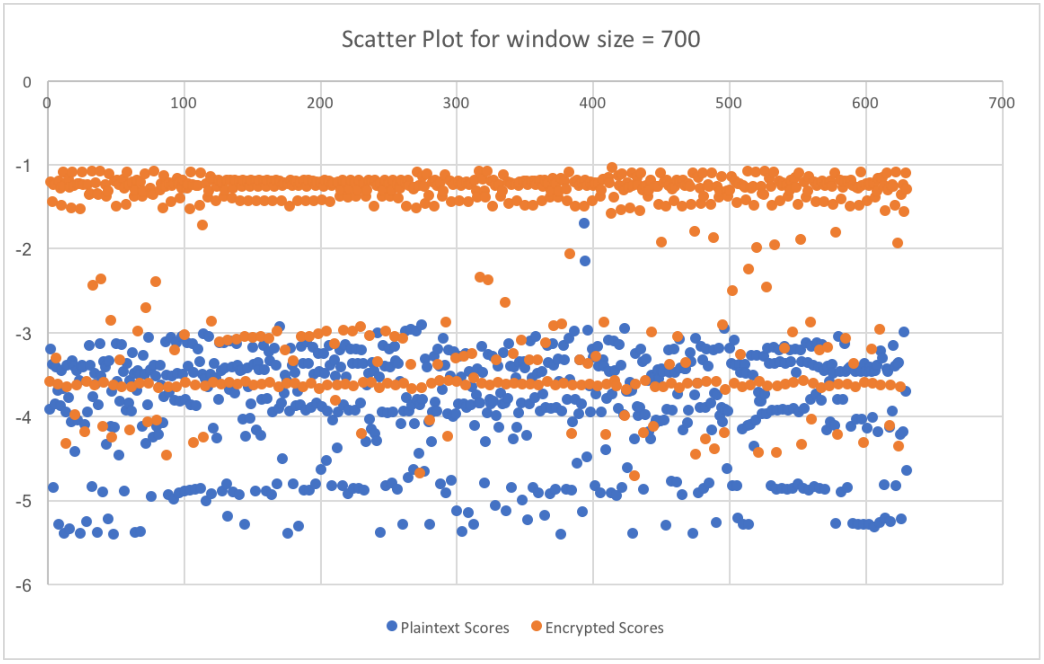


Figure A.30: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 700.

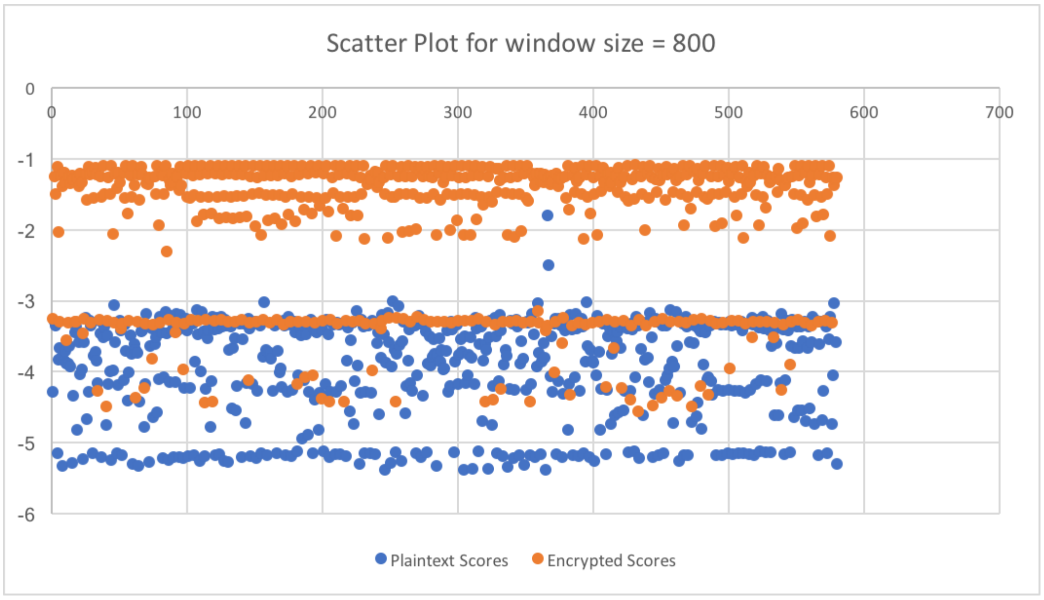


Figure A.31: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 800.

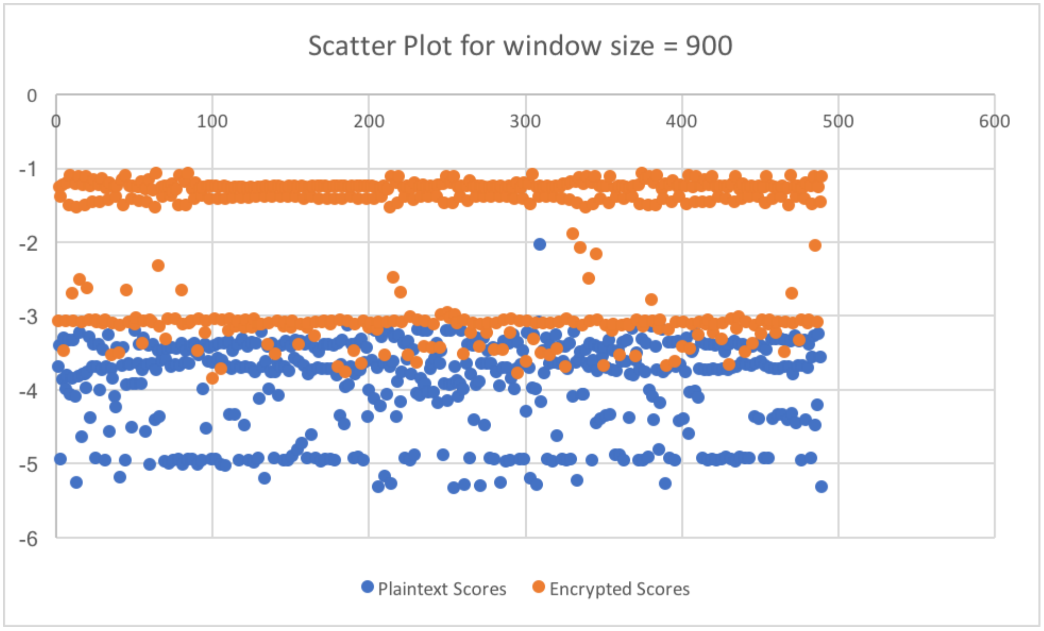


Figure A.32: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 900.

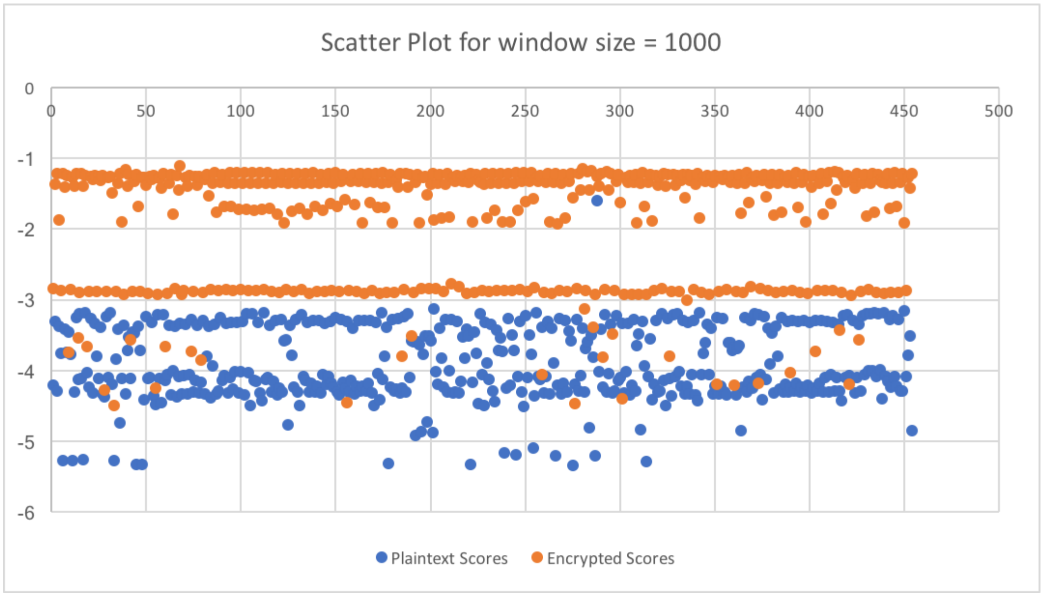


Figure A.33: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1000.

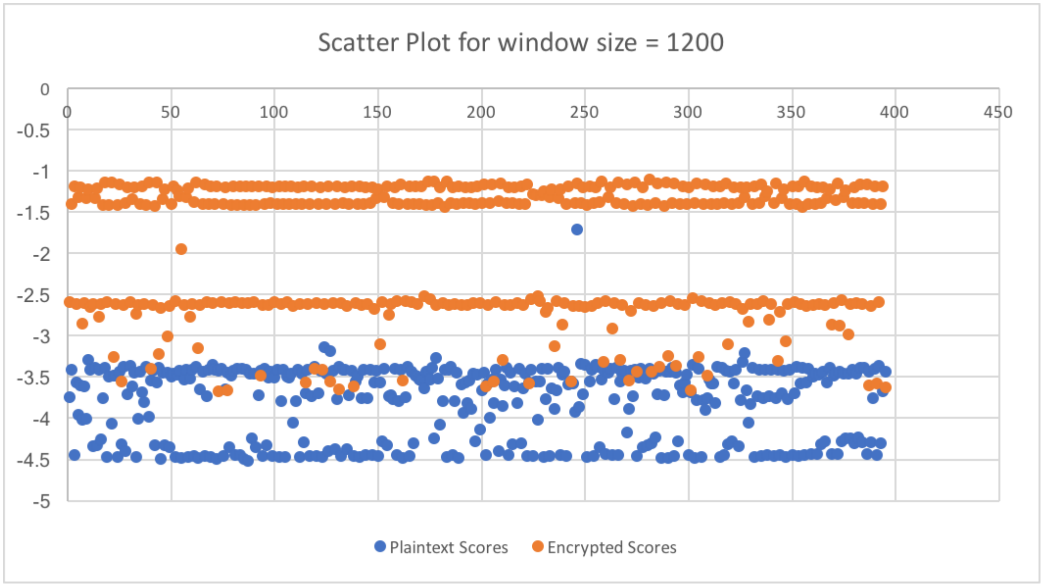


Figure A.34: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1200.

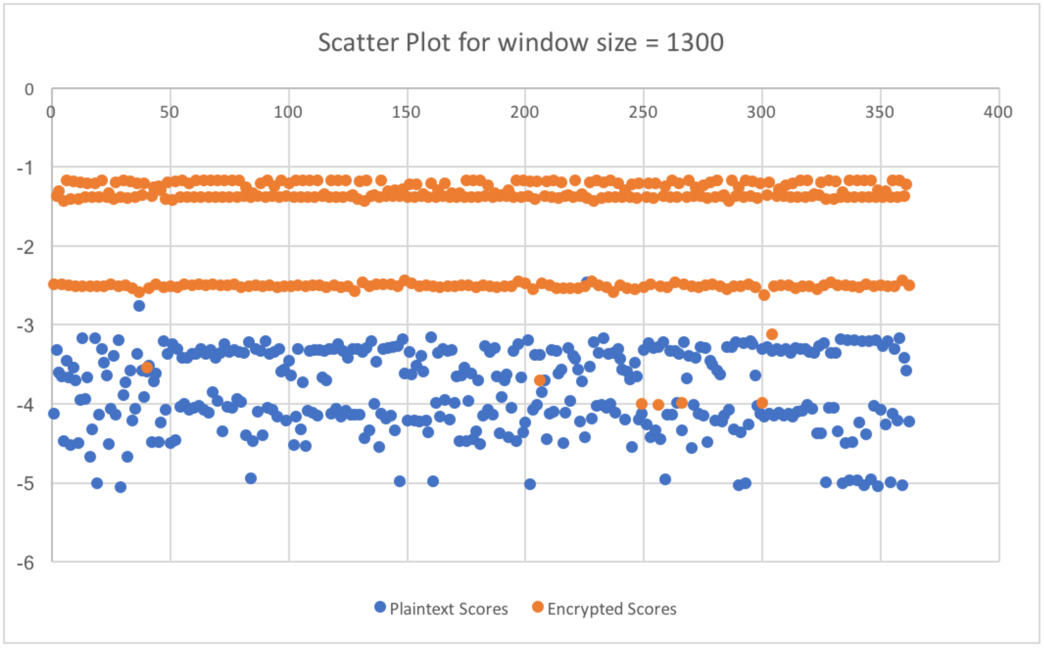


Figure A.35: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1300.

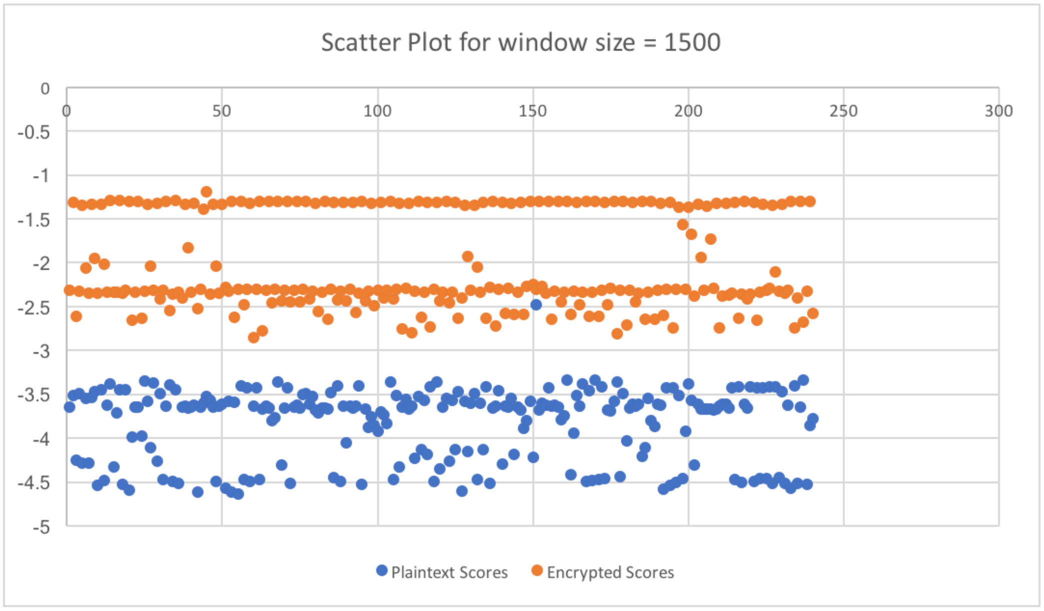


Figure A.36: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1500.

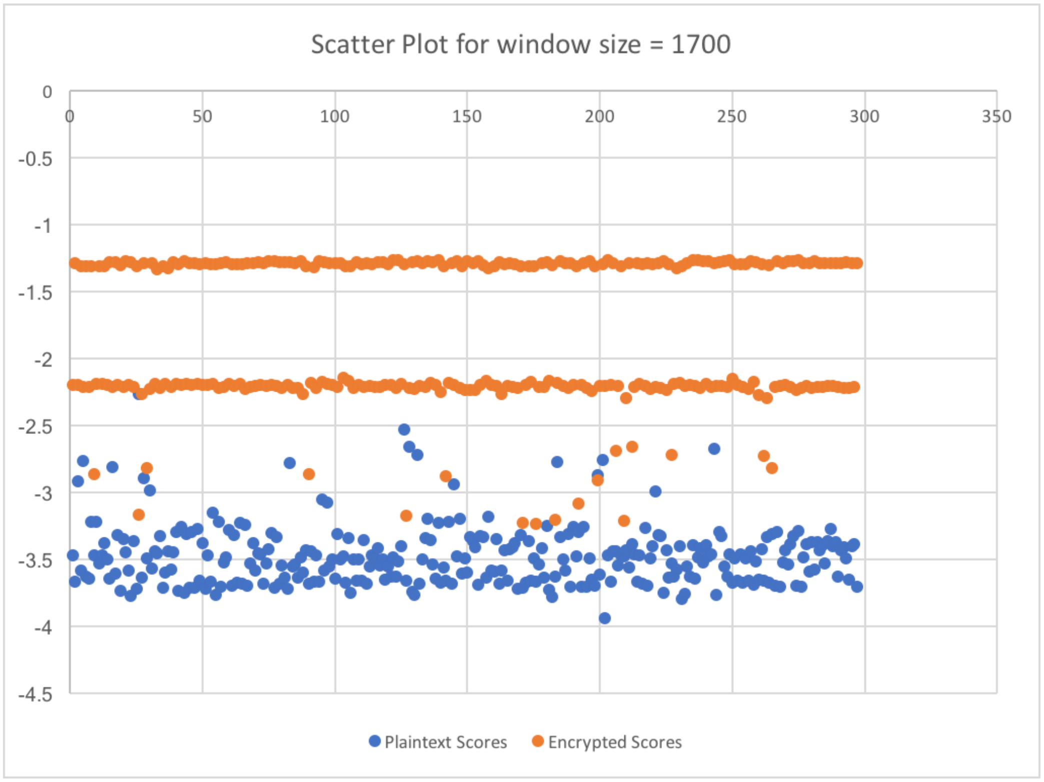


Figure A.37: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1700.

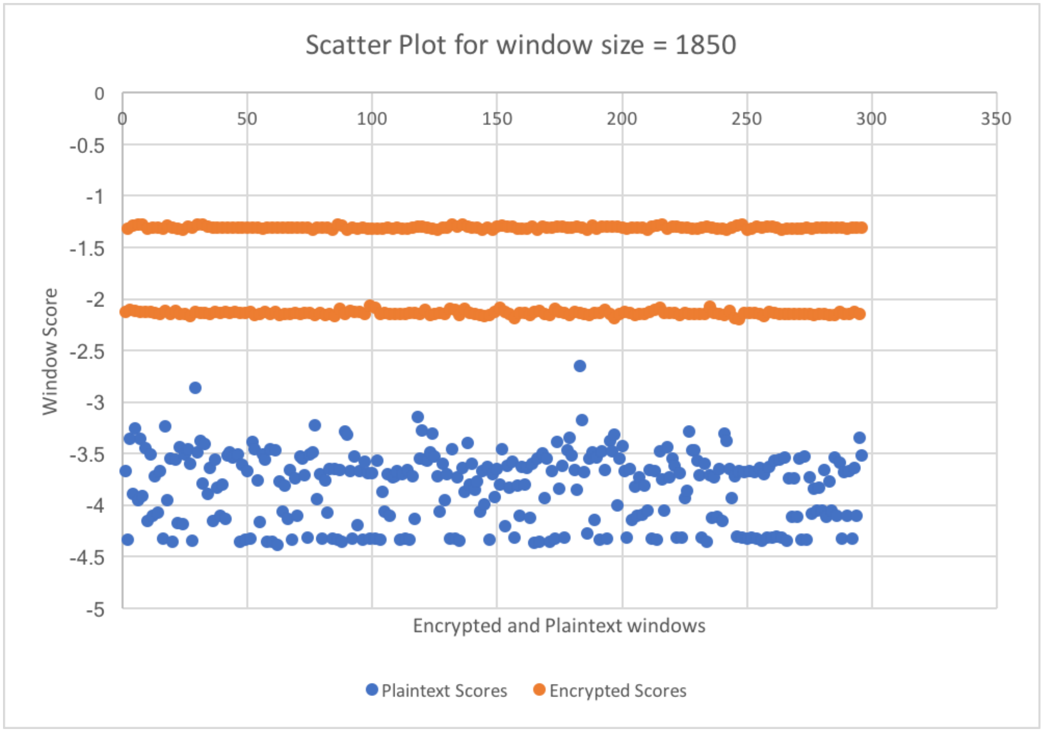


Figure A.38: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 1850.

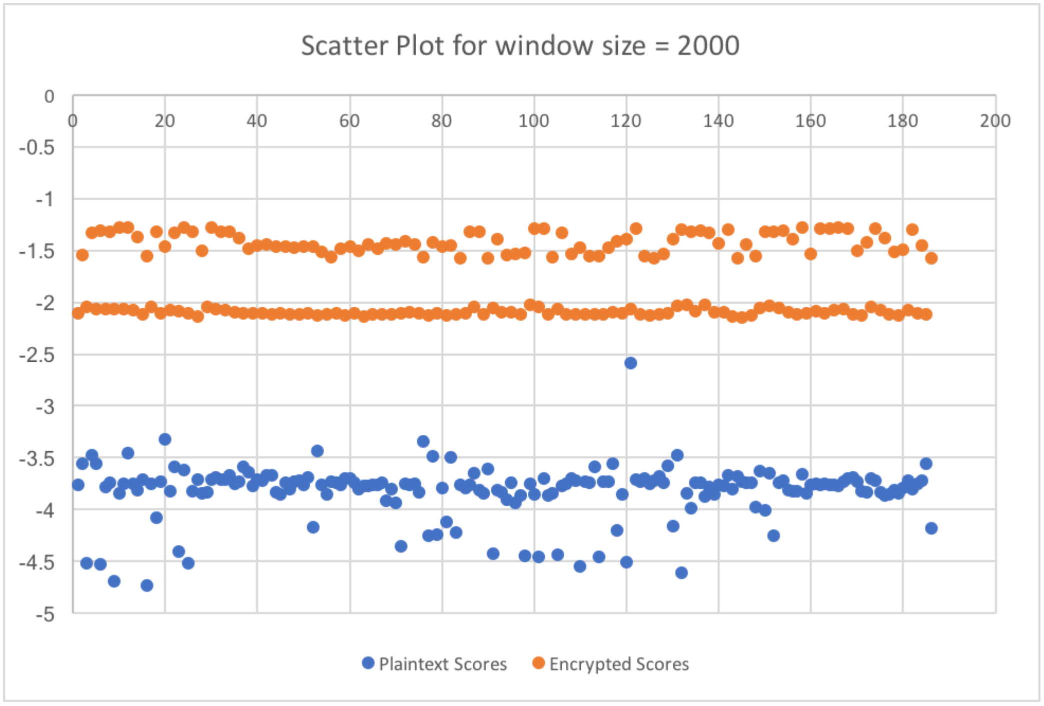


Figure A.39: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2000.

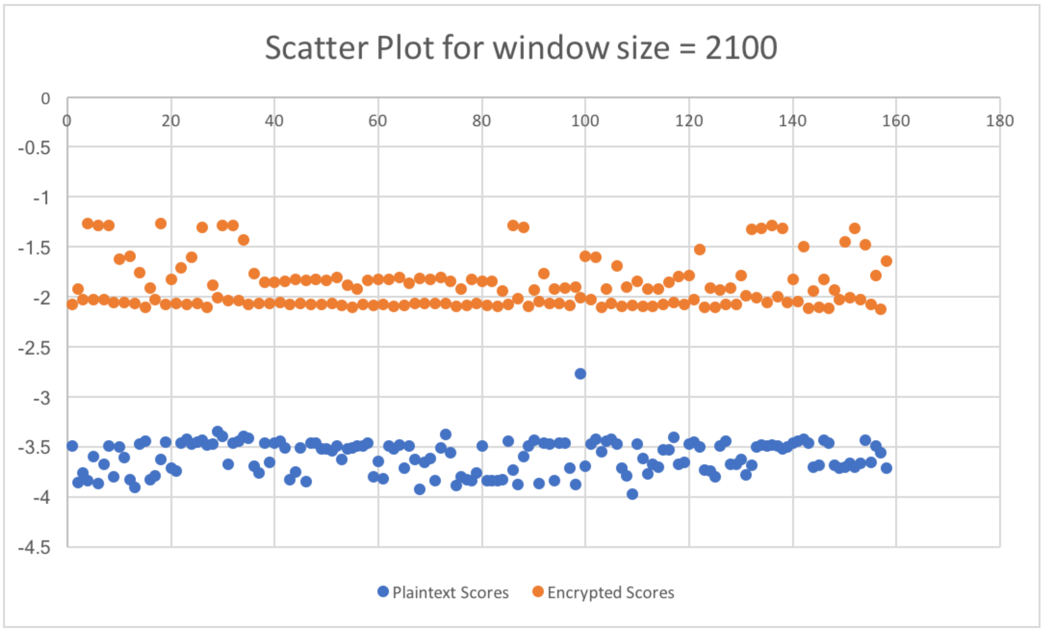


Figure A.40: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2100.

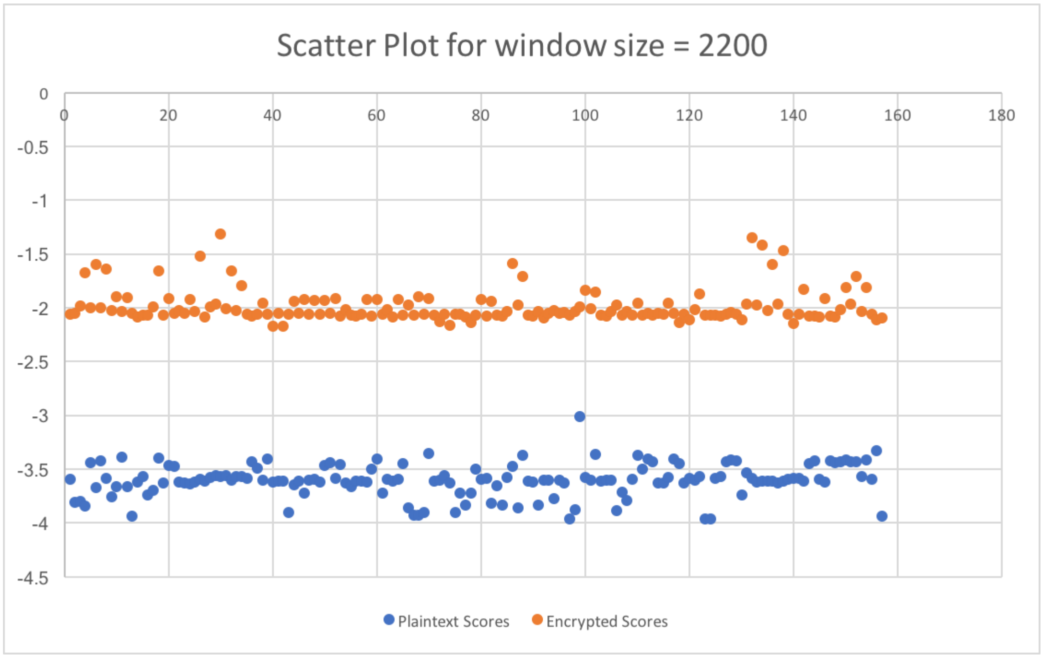


Figure A.41: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2200.

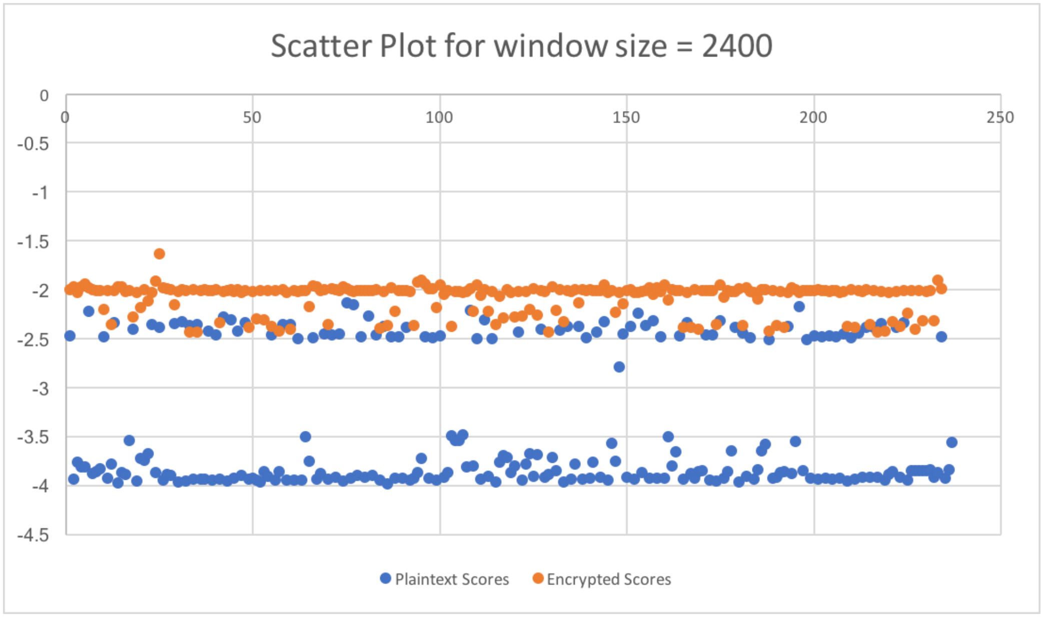


Figure A.42: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2400.

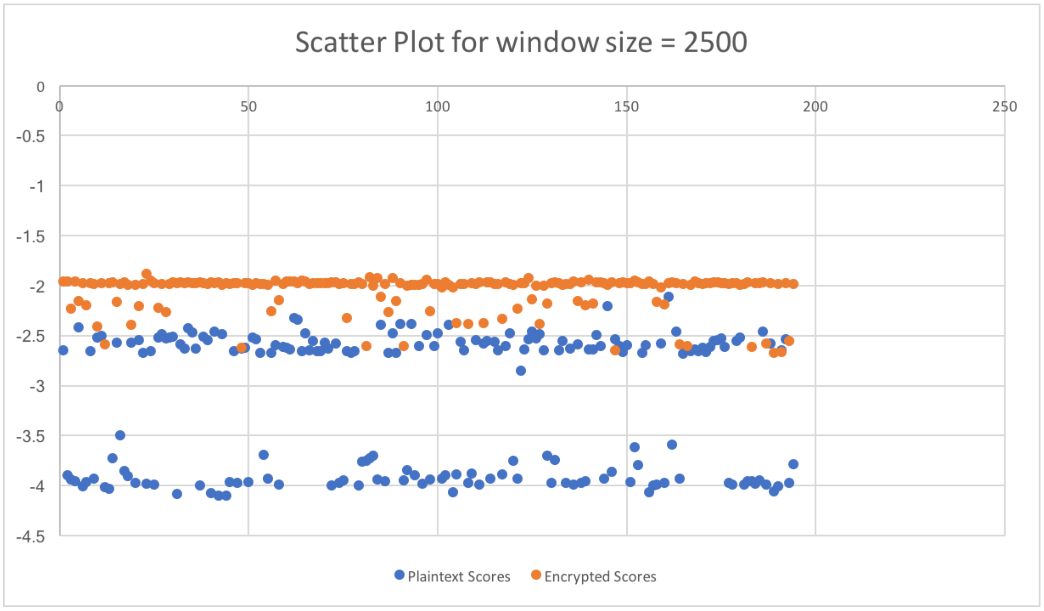


Figure A.43: Scatter plot of scores with $N = 2$, $M = 30$ and window size = 2500.

APPENDIX B

Appendix B: AUC Values

Table B.6: AUC values for different window sizes

Window Length	AUC
100	0.82698
150	0.83133
200	0.84274
300	0.83853
400	0.85834
500	0.83188
700	0.87828
800	0.92157
900	0.96397
1000	0.96104
1200	0.96870
1300	0.99219
1500	0.99921
1700	0.99753
1850	1.00000
2000	1.00000
2100	1.00000
2200	1.00000
2400	0.97986
2500	0.97387

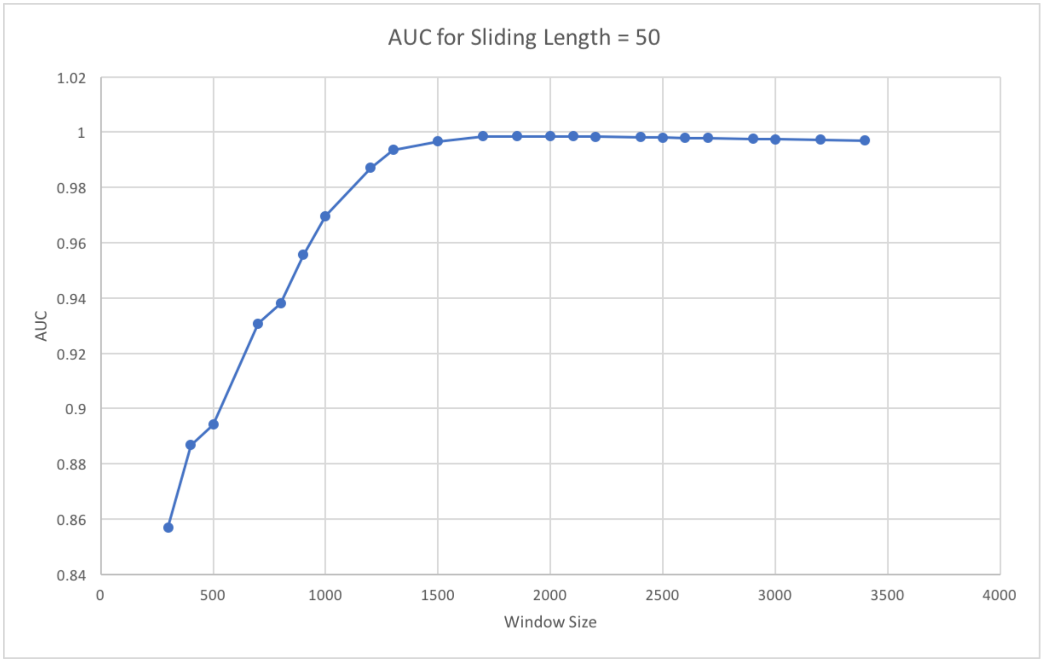


Figure B.44: AUCs for different window lengths using sliding length = 50.

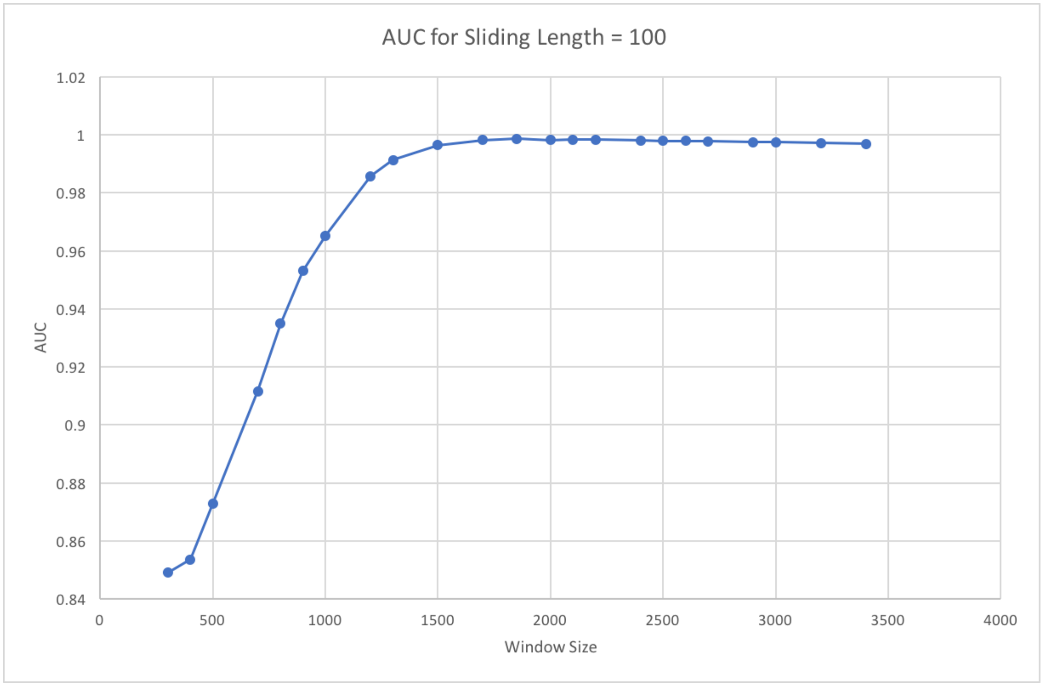


Figure B.45: AUCs for different window lengths using sliding length = 100.

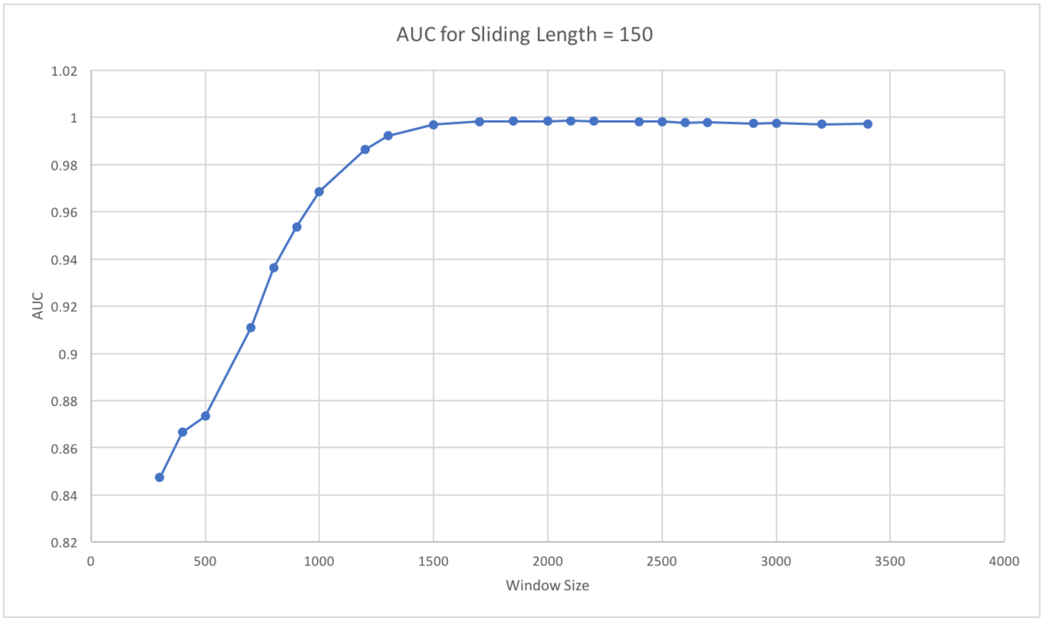


Figure B.46: AUCs for different window lengths using sliding length = 150.

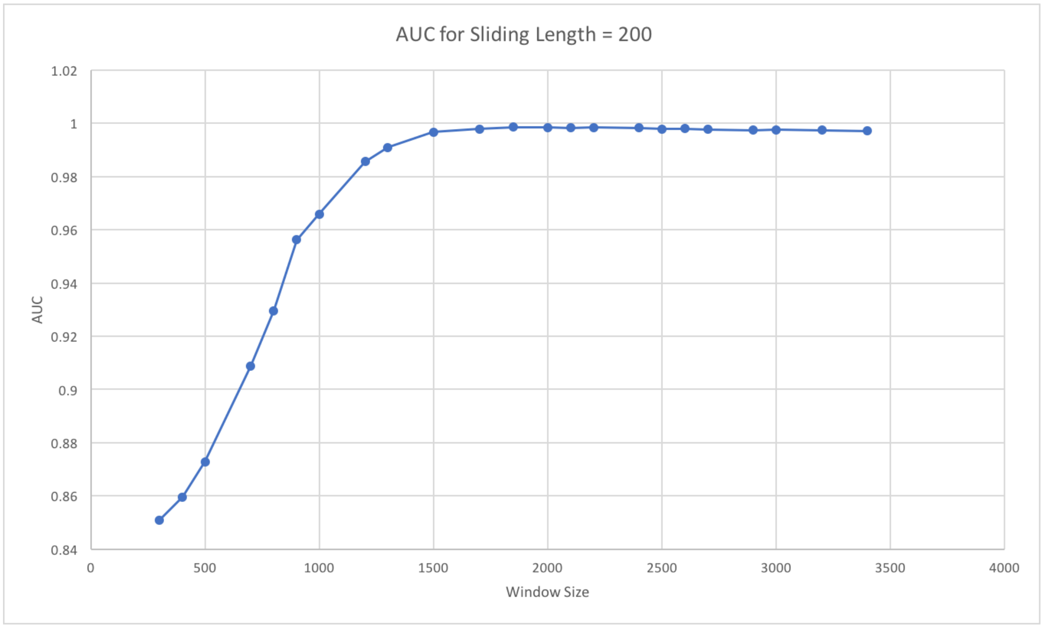


Figure B.47: AUCs for different window lengths using sliding length = 200.

Table B.7: AUC values, ROC threshold, TPR and FPR values for N, M combinations

N	M	AUC	Threshold	TPR	FPR
2	20	1	-0.99181 -1.45959 -3.64948	0.00641 1.00000 1.00000	0.00 0.00 1.00
2	25	1	-1.66186 -2.00135 -4.00151	0.00641 1.00000 1.00000	0.00 0.00 1.00
2	30	1	-2.35073 -2.69141 -4.74532	0.00641 1.00000 1.00000	0.00 0.00 1.00
2	35	1	-2.40085 -2.77691 -4.87067	0.00641 1.00000 1.00000	0.00 0.00 1.00
2	40	1	-2.78617 -3.18041 -4.97579	0.00641 1.00000 1.00000	0.00 0.00 1.00
2	50	1	-3.16389 -3.42296 -5.71160	0.00641 1.00000 1.00000	0.00 0.00 1.00
3	20	1	-0.73771 -1.05490 -4.26691	0.00641 1.00000 1.00000	0.00 0.00 1.00
3	25	1	-1.30815 -1.53355 -4.10362	0.00641 1.00000 1.00000	0.00 0.00 1.00
3	30	1	-1.70695 -2.03049 -4.22782	0.00641 1.00000 1.00000	0.00 0.00 1.00
3	35	1	-1.66362 -1.90545 -4.85888	0.00641 1.00000 1.00000	0.00 0.00 1.00
3	40	1	-2.04157 -2.31019 -4.98767	0.00641 1.00000 1.00000	0.00 0.00 1.00
3	50	1	-2.42950 -2.60104 -5.30480	0.00641 1.00000 1.00000	0.00 0.00 1.00