San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Spring 2014

A Tiered Approach to Detect Metamorphic Malware With Hidden Markov Models

Ashwin Kalbhor
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

 Part of the Computer Sciences Commons

Recommended Citation

Kalbhor, Ashwin, "A Tiered Approach to Detect Metamorphic Malware With Hidden Markov Models" (2014). *Master's Projects*. 360.
DOI: https://doi.org/10.31979/etd.nfq8-vzdx
https://scholarworks.sjsu.edu/etd_projects/360

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

A Tiered Approach to Detect Metamorphic Malware

With Hidden Markov Models

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

by

Ashwin Kalbhor

May 2014

© 2014

Ashwin Kalbhor

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

A Tiered Approach to Detect Metamorphic Malware with Hidden Markov Models

by

Ashwin Kalbhor

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2014

Dr. Thomas Austin	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science
Dr. Mark Stamp	Department of Computer Science

ABSTRACT

A Tiered Approach to Detect Metamorphic Malware with Hidden Markov Models

By Ashwin Kalbhor

Work on the use of hidden Markov models (HMM) to detect viruses has been carried out previously with good results [2], but metamorphic viruses like MetaPHOR [27] and metamorphic worms like MWOR [3] have proven to be able to evade detection techniques based on HMMs. The dueling HMM approach looks to detect such viruses by training an HMM model for each of the metamorphic virus / worm families. The tests and the results from these have shown that this approach has been able to detect the metamorphic MetaPHOR virus with reasonable accuracy but with significantly more overhead.

This paper presents a tiered approach that improves on this by achieving the same results as the dueling approach but with significant performance improvement in terms of time. Essentially the idea is to eliminate most putative malware with the threshold approach, reserving the dueling HMM analysis for more difficult cases. We achieve accurate results with significantly less performance overhead than the dueling HMM strategy. Furthermore, our approach successfully detects MWOR worms with a high degree of accuracy.

ACKNOWLEDGEMENTS

I would like to thank Dr. Thomas Austin from the bottom of my heart for providing me with invaluable guidance and his patience when things weren't working out as they should have been. I would also like to thank Dr. Mark Stamp and Dr. Sami Khuri for their invaluable feedback on the project.

Last but not least I would like to thank my parents and friends for nurturing me and constantly supporting me. Special thanks to Amogh, Akash, Hardik, Lakshmi, Mangesh, Pournima, and Sanya for keeping me on track as well as for proofreading the report.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Malicious software or malware is software which is built with a malicious intent [5]. Malware is inevitable in the current connected world where a lot of information is stored digitally. The intent of a malware can vary from covertly recording sensitive data to downright harmful ones which will cause severe damage to the operating system.

Metamorphism is a technique in computer programs where the program continuously changes its structure over successive generations, thus changing the structural appearance while still retaining their original functionality.

A malware has an inherent need to stay undetected so as to continue what it is doing and also to spread to other hosts. This requirement of a malware to stay hidden has driven malware writers to actively explore techniques using which a malware can morph itself to several unique forms to effectively evade detection. Traditionally malware has been detected by using a variety of techniques like anomaly detection, change detection, and the most widely used signature detection [6].

The signature detection technique looks to match signatures within a file with a stored and verified set of known virus signatures. However, signatures are based on the byte patterns in the body of the malware. If the structure of a malware keeps changing then the signature detection technique will fail to come up with a common signature across all the structurally different generations of the malware. Thus, changing the structure can allow a virus to effectively evade detection. Later research into this field has led to the exploration of the use of hidden Markov models (HMMs) to detect metamorphic malware [2].

However, newer malware like MWOR [3] and MetaPHOR [27] [28] have been shown to be able to evade detection techniques based on HMMs. The dueling HMM approach [4] looks to detect such viruses by training an HMM model for each of the metamorphic virus / worm families, improving accuracy but with significantly more processing overhead.

In our project we build on the research of the dueling HMM strategy [4]. In the dueling HMM strategy an HMM model is trained for every malware family. This however, leads to an increase in the amount of time that it takes to classify the file. This is due to the number of HMMs that the file has to pass through until it can be classified. To reduce the time it takes to classify a file we look at a tiered approach to increase the efficiency.

Furthermore, we refine the dueling HMM strategy by introducing a bias to the design, that is, the HMM representing a given malware family is assigned a penalty compared to the results of benign HMMs. This change allows us to adjust the false positive / false negative ratio of our results more easily. This was not previously done with the dueling HMM strategy.

In this paper, Chapter 2 will describe malware and the classification of malware as well as the different techniques used in detecting malware. Chapter 3 gives a detailed look at metamorphic malware. In Chapter 4, we will have a look at hidden Markov models as well as the threshold and dueling approaches using HMM. In Chapter 5 we will have a detailed look at the design and implementation of this project. Chapter 6 will describe the experiments and the results of the experiments. Chapter 7 will describe the conclusion and the future work.

CHAPTER 2

Malware

Malware can be further divided into sub classes. Based on the characteristics of the malware, it can mainly be subdivided into the following classes as described in [5],

1. Virus
2. Worm
3. Trojan Horse
4. Backdoor
5. Spyware
6. Adware

However we will be confining ourselves to viruses and worms. So let us discuss these two types in more detail.

2.1 Virus

Fred Cohen defines a virus as a software program which has the ability to infect other programs by replicating itself into other program files. A virus can also possess the ability to mutate itself [7].

An alternate definition of a virus as mentioned in Peter Szor's book, "The Art of Computer Virus Research and Defense" [1] states that "*A computer virus is a program that recursively and explicitly copies a possibly evolved version of itself*". Fred Cohen's informal definition of the virus doesn't strictly hold true. In recent times there are programs called companion viruses [1] which will not modify other program files, but will use other means to execute themselves.

Viruses can further be divided into 4 main types as described in [1]: encrypted viruses, oligomorphic viruses, polymorphic viruses and metamorphic viruses. We will look at these in more detail in the following subsections.

2.1.1 Encrypted Viruses

An encrypted virus consists of two parts: the actual virus body that is encrypted and the decryption code section. Whenever a virus is to be executed, the decryptor code is executed and it decrypts the encrypted virus body. Once this is done, the actual virus is executed.

```
lea     si,  Start  ; position to decrypt (dynamically set)
mov     sp,  0682 ; length of encrypted body(1666 bytes)

Decrypt:
xor     [si],  si    ; decryption key/counter 1
xor     [si],  sp    ; decryption key/counter 2
inc     si           ; increment one counter
dec     sp           ; decrement the other
jnz Decrypt          ; loop until all bytes are decrypted
Start:               ; Encrypted / Decrypted Virus Body
```

Figure 1: Decryptor code block for Cascade [1]

Among the first encrypted viruses was a virus called Cascade [8] [7]. Figure 1 taken from Peter Szor's book [1] shows the decryptor for the Cascade virus. Cascade uses the XOR operation to encrypt itself. This can be seen from the instruction 'xor [si], si' and 'xor [si], sp'. The loop containing these instructions will execute till all the bytes have been decrypted. However, since the decryptor is constant throughout all the virus specimens, it is easy to detect this virus since a detection string can be picked from the constant decryptor code.

2.1.2 Oligomorphic viruses

Oligomorphic viruses try to remedy the shortcomings of the encrypted virus, namely the constant signature due to the constant decryptor code. Oligomorphic viruses create mutations of the decryptor in successive generations. Thus, the problem of same signatures was solved.

```
mov     ebp, 00405000h        ; select base
mov     ecx, 0550h            ; this many bytes
lea     esi, [ebp+0000002E]   ; offset of "Start"
add     ecx, [ebp+00000029]   ; plus this many bytes
mov     al, [ebp+0000002D]    ; pick the first key

Decrypt:
nop                           ; junk
nop                           ; junk
xor     [esi], al             ; decrypt a byte
nop                           ; junk
inc     al                    ; slide the key
dec     ecx                   ; are there any more bytes to decrypt?
jnz     Decrypt               ; until all bytes are decrypted
jmp     Start                 ; decryption done, execute body

; Data area
Start :
;       encrypted/decrypted virus body
```

Figure 2: W95 / Memorial 1 [1]

```
mov     ecx, 0550h              ; this many bytes
mov     ebp, 013BC000h          ; select base
lea     esi, [ebp+0000002E]     ; offset of "Start"
add     ecx, [ebp+00000029]     ; plus this many bytes
mov     al, [ebp+0000002D]      ; pick the first key

Decrypt:
nop                             ; junk
nop                             ; junk
xor     [esi], al               ; decrypt a byte
inc     esi                     ;next byte
nop                             ; junk
inc     al                      ; slide the key
loop    Decrypt                 ; until all bytes are decrypted
jmp     Start                   ; decryption done, execute body

; Data area
Start:
;       encrypted/decrypted virus body
```

Figure 3: W95/ Memorial 2 [1]

One simple technique to do so, which was followed by W95 / Memorial, is to have more than one decryptor and then to use a random decryptor to decrypt itself.

The first and the second code snippets are slightly mutated forms. The first decryptor (Figure 2) uses the 'jnz' instruction to check whether all bytes have been decrypted; otherwise it jumps to the "Decrypt" label.

In the second decryptor (Figure 3), the ‘loop’ instruction is directly used. Thus, though the first and the second code snippets appear to be different structurally, they perform the same operation (i.e – Looping). This mutation helps to change the signature of the two variants of the W95 / Memorial virus to generate two different signatures and hence help evade detection.

A virus is called Oligomorphic only when the decryptor can mutate to create a small number of unique signatures.

2.1.3 Polymorphic viruses

Polymorphic viruses are similar to Oligomorphic viruses in the way that both techniques use different decryptors. However polymorphic viruses have a large amount of unique decryptors as opposed to Oligomorphic viruses which have a small amount of mutated decryptors. The large amount of decryptors leads to a large no of detection signatures [6]. It can thus evade signature based detection techniques.

Polymorphic viruses still need to decrypt the virus body before actually being able to execute. This turns out to be an Achilles heel for these types of viruses. Emulation based techniques can be used to detect these types of viruses.

The suspected file is run in a sandbox and the decryptor code is allowed to decrypt the virus body. Once the virus has been decrypted all that remains is to match the signature of the virus against a database [6]. This approach requires the suspected file to actually run before it can be classified; hence the time required to classify the file also increases.

Figure 4, taken from “The Art of Computer Virus Research and Defense” [1], represents different samples of the same polymorphic virus.

In Figure 4, shapes on the left represent the encrypted virus while the shapes on the right represent what the virus will look like once decrypted.

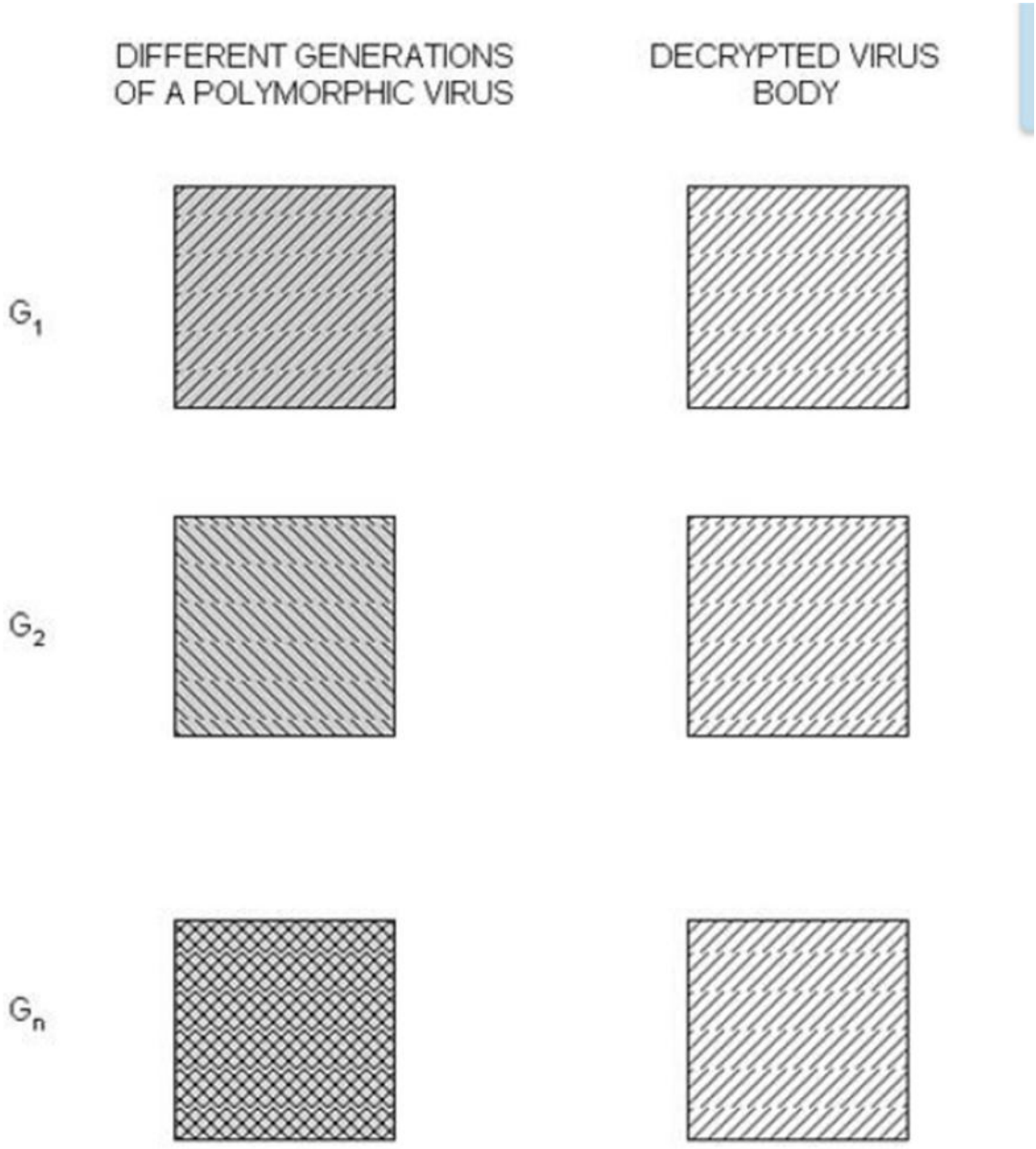

Figure 4 : Polymorphic virus [1]

The virus remains the same while the decryptor changes, thereby changing the overall appearance of the virus.

2.1.4 Metamorphic viruses

A metamorphic virus changes itself in every generation so as to create unique copies of itself. This allows it to evade signature based detection techniques. We will discuss the different techniques used by metamorphic viruses in more detail in Chapter 3.

2.2 Worms

A worm is self-replicating software which does not require the use of a host file to spread from one host to another [9]. Some definitions of worms include worms as special types of viruses, while others define them to be distinct from viruses. Worms can cause much more rapid destruction as they can spread at much faster rates than viruses, thus effectively slowing down networks by causing huge traffic. An example of this is the Slammer worm which infected 90% of all the vulnerable hosts under 10 minutes [10]. A metamorphic worm like MWOR [3], if carrying a destructive payload, could cause huge financial losses before any human intervention.

2.3 Malware Detection

Malware detection can be classified [6] into the following categories based on the detection techniques used.

2.3.1 Signature detection

Signature detection uses a string to uniquely classify a virus. A signature is a common pattern of bits which can be found in all occurrences of the malware. Signature detection is efficient and fast. A drawback of signature detection techniques is that a huge database to the

order of ten to a hundred thousand signatures needs to be maintained and every suspected file's signature is matched against every signature in the database [6].

Another drawback of the signature detection technique is that if there are small changes in the structure of the malware, evasion is possible.

2.3.2 Change detection

In change detection, all the files on the system are constantly monitored for any change in the bits that constitute them. Whenever a virus infects a file, it might need to change bits in the file [6]. This fact is leveraged by the change detection technique. This type of technique will have no false negatives but will have a reasonably large number of false positives since users too will legitimately change files [6].

2.3.3 Anomaly detection

In the anomaly detection technique, a normal / baseline behavior of the user is defined. After the baseline is defined all the activities on the system are monitored [11]. Whenever there is any activity that is significantly deviant from the normal behavior, a red flag is raised. For this type of detection technique the main dilemma is how to classify what behavior is normal and what behavior is abnormal.

2.3.4 String matching using wildcards

String matching is a technique in which recorded samples of bytes (also called as a string) from known viruses are matched against the strings from a file that needs to be scanned. Often the string is not an exact match and instead there are slight variations in the string. To help detect such strings, which differ from each other in a few bytes, wild cards are used. Wildcards

allow the string to be matched even if it differs in the specified bytes. As an example, consider the following order of bytes.

1223 45E8 5C15 9A65 231B

1223 45E8 5C2F 9A65 231B

A string containing wildcards that matches the above byte sequences is

1223 45E8 5C?? 9A65 231B

The interpreter will look to match all bytes one by one until the '5C' byte and proceed only if the bytes match. However, due to the '??' wildcard, the interpreter will match any byte or ignore the byte completely, and for the remaining bytes it will continue matching and proceed only if there is an exact byte match.

2.3.5 Emulation

In code emulation, the suspected file is executed in a simulated environment that matches the environment that the file would have executed in [30]. In this approach, every instruction is executed individually and the changes it produces are observed to see whether the behavior is similar to that of a virus file [31]. Emulators allow the detection of junk / dead code and can defeat techniques like equivalent instruction substitution as demonstrated in [32].

2.3.6 Geometric detection

Geometric detection relies on detecting changes that are made to the structure of the file by the virus [1]. Due to the method it uses to detect a virus, geometric detection technique produces false positives.

An example [1] of this is the W95 / Zmist virus [37]. Whenever ZMist infects a file there is an increase of 32 KB in the data section of the infected file. This change is used by the geometric detection technique to classify the file as malware. Changes in file size, however, are never solely due to malware alone. There might be legitimate changes in the size of the file. Like in the above example, the data section can increase in size in case of run time compressed file. In the geometric detection technique these legitimate runtime compressed files will show up as malware.

CHAPTER 3

Metamorphic Malware

To evade detection, virus writers started using metamorphism to prevent anti-virus programs from latching on to a common signature which could be used to detect the malware effectively. A malware can be made metamorphic by the use of one or more of the techniques in section 3.1. Figure 5 from “The Art of Computer Virus Research and Defense” [1] denotes metamorphic malware.

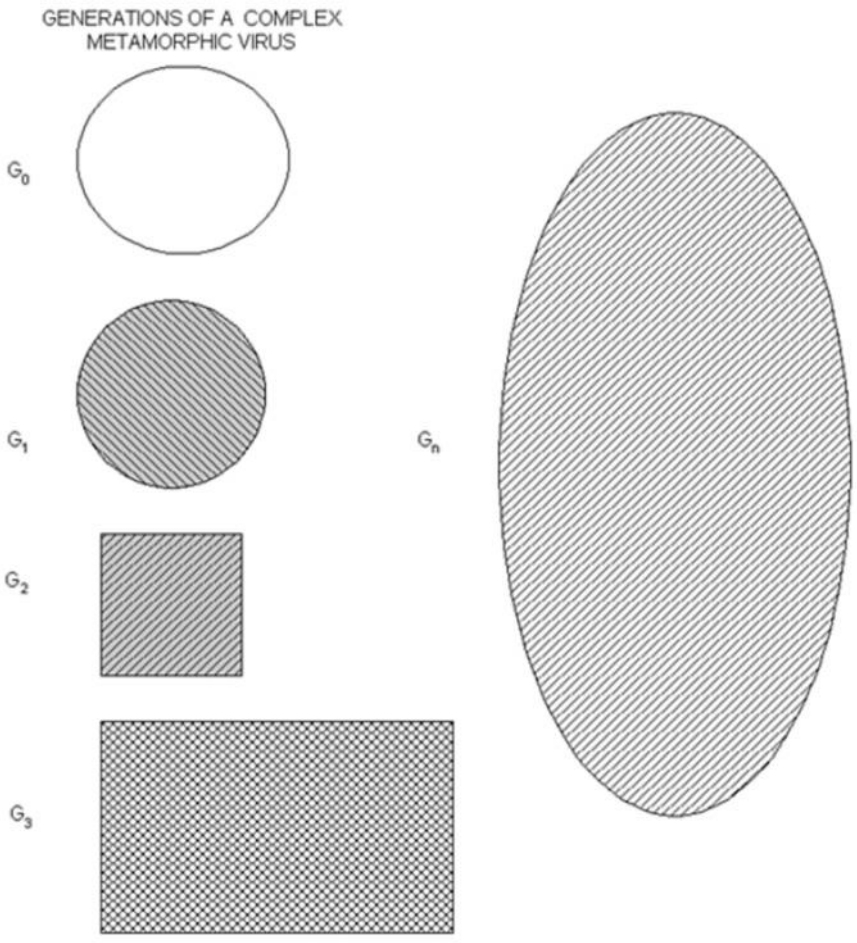

Figure 5: Metamorphic Malware [1]

As Figure 5 denotes, subsequent generations of the virus G (G_1, G_2, G_3.....G_n) will change their byte patterns to appear different from each other and thus evade detection by signature based detection techniques.

In general, for successive generations of the virus to be able to completely evade detection techniques it has to carry out the following steps [33] [34].

1. Decode – The metamorphic malware must first decode itself so that it can execute. To stay undetected the malware needs to encrypt itself so that virus detection techniques don't latch on to constant signatures.
2. Deliver payload – Once the metamorphic malware has been decoded it will execute the malicious code to fulfill the malicious intent with which it was constructed.
3. Morph itself – Once the payload has been delivered, the malware has to now take steps to morph itself to look different in the next generation so as to evade detection techniques. This can be done by a variety of techniques mentioned in the next sub section.

3.1 Techniques used to create metamorphic malware

The techniques used to make malware are discussed in more detail in the following sub sections.

3.1.1 Subroutine Permutation

In sub routine permutation the malware body is divided into multiple subroutines. Over successive generations the virus then rearranges these subroutines to form a unique static structure [12]. The number of unique static structures possible will be directly proportional to the

number of distinct sub routines comprising the virus. If a virus has *n* number of distinct subroutines then the number of possible unique static structures will be given by *n!*

An example of a virus using sub routine permutation is the BadBoy [13] virus. The BadBoy virus had eight unique subroutines. Hence BadBoy had 8! = 40320 unique combinations. An example of subroutine permutation can be seen in Figure 6. The subroutines on the left are the original subroutines while the ones on the right are the permuted subroutines. We can see that although the subroutines have been permuted, program execution will still start at block marked as subroutine 1.

3.1.2 Dead Code Insertion

In this technique, the metamorphic virus uses the addition of dead code like *NOP* instructions to evade detection [14]. Since the added code is dead / junk code, it doesn't affect the functionality of the malware. However, the addition of these instructions will tend to make the program differ from its previous generations. The number of unique structures that can be created are virtually endless.

In Figure 7, we can see dead code added to the body of the program to make it look different. Lines 4 and 5 contain dead code in the form of *NOP*, which stands for no operation. Thus, it has no effect on the functionality of the program but still serves to change the appearance of the program.

Dead code can be such that it is never executed or it could be such that it gets executed but has no effect whatsoever on the original functionality of the virus.

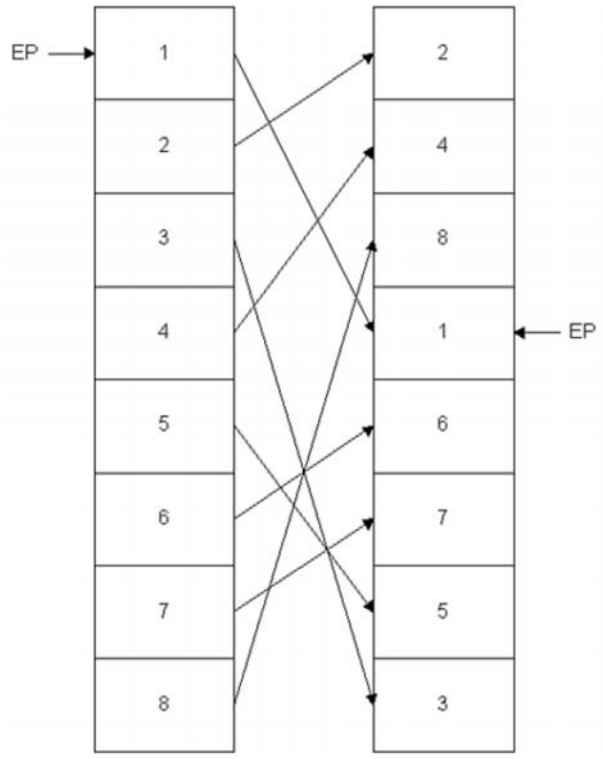

Figure 6: BadBoy - Subroutine permutation [1]

```
call 0h
pop ebx
lea ecx, [ebx+42h]
nop
nop
push ecx
push eax
inc eax
push eax
dec [esp - 0h]
dec eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Figure 7: Dead code insertion [15]

3.1.3 Register Swap

The virus using the register usage exchange technique will execute using different registers over different generations [1]. W95 / Regswap [8] is an example of a malware using this technique.

```
a.)

5A                      pop   edx
BF04000000              mov   edi,0004h
8BF5                    mov   esi,ebp
B80C000000              mov   eax,000Ch
81C288000000            add   edx,0088h
8B1A                    mov   ebx,[edx]
899C8618110000          mov   [esi+eax*4+00001118],ebx

b.)

58                      pop   eax
BB04000000              mov   ebx,0004h
8BD5                    mov   edx,ebp
BF0C000000              mov   edi,000Ch
81C088000000            add   eax,0088h
8B30                    mov   esi,[eax]
89B4BA18110000          mov   [edx+edi*4+00001118],esi
```

Figure 8: Register Swap [1]

As can be seen in the first code snippet ‘move edi, 004h’ is replaced by the ‘move ebx, 004h’. Similarly, ‘move eax, 000Ch’ is replaced by ‘move edi, 000Ch’.

3.1.4 Equivalent Code Substitution

In equivalent code substitution the virus replaces an instruction or a set of instructions in the virus body with an equivalent instruction or set of instructions. For example, something like

‘inc eax’ is the same as ‘add eax, 1’. Similarly, instruction like ‘xor eax, eax’ can be substituted by ‘move ax, 0’.

3.1.5 Random Jump Insertion

Another way to generate metamorphism is to introduce jumps at random places in the code [1]. The jumps are inserted in such a way that although the jumps are always random the sequence of the execution of the virus remains unchanged. Viruses like Zperm [29] use this technique to remain undetected.

Figure 9: Zperm - Random Jump instructions [8]

In Figure 9, which is from Peter Szor’s paper [8], we can see three distinct ways in which a Zperm virus will mutate itself by introducing random jumps, all the while maintaining functional output. In the first part, the first two instructions are in succession after which the flow will jump to instruction 3 present after garbage instructions. The program flow will then go to instruction 4 at the start of the code. This is different from the second sample where, after the first instruction, the program flow will jump directly to the start of the code. Similarly, the third code sample differs from the first two samples based on the jump source and jump destinations which differ from previous generations.

3.1.6 Code Transposition

Code transposition is a technique in which two blocks of code, which are not dependent on each other, are rearranged [14]. This will change the static as well as the dynamic signature of the virus and help evade detection. Take Tamboli's [16] example:

ins1 [reg1] [reg2]

ins2 [reg3] [reg4]

If ins1 and ins2 are not interdependent (i.e. – the output of one operation has no effect on the other) then the two instructions can be transposed as follows –

ins1 [reg3] [reg4]

ins2 [reg1] [reg2]

Other techniques used to make malware metamorphic include using non-normalizable functions [17]. Often a single technique by itself might not be enough to help a metamorphic malware evade detection completely. Hence, more than one of the above techniques are used in conjunction.

There also exist virus construction kits which can be used by anyone to create metamorphic malware. Some of the virus kits that are available to create metamorphic malware include PS-MPC [46], G2 [18], NEG [47], NGVCK [19], MPCGEN [20].

The introduction of these virus construction kits considerably simplifies the virus creation process. In some of the virus construction kits there are preloaded modules which can be selected

to create a virus. This makes the virus creation process very simple and allows even individuals not well versed with assembly language to write malware.

CHAPTER 4

Hidden Markov Model

A hidden Markov model (HMM) is based on the use of statistics to detect patterns. Hidden Markov models have been used in various areas including speech recognition, software piracy detection, biological sequence analysis, study of three dimensional protein structure, neuroscience, as well as electrocardiography characterization. In the past few years, research in the use of HMMs to detect metamorphic malware is well documented [24] [12] [25] [26]. HMMs are trained by using the op codes from known malware data. Once the HMM has been trained it can then be used to score files. The result (log likelihood) can then be used to classify the specimen [22].

4.1 Overview of Hidden Markov Models

HMMs are explained in the below subsections using an example from Stamp's paper [23]

4.1.1 Notations

The notations for an HMM are based on [23] –

Let,

T be the observation sequence length

N be the number of states in the model

M be the number of observation symbols

$Q = \{q_1, q_2, q_3…., q_{N-1}\}$ be the distinct states of the Markov process

$V = \{0, 1,…, M – 1\}$ be the set of possible observations

A be the state transition probabilities

B be the observation probability matrix

π be the initial state distribution

$O = (O_0, O_1, \ldots, O_{T-1})$ be the observation sequence

Then a generic hidden Markov model can be modeled as follows [23]

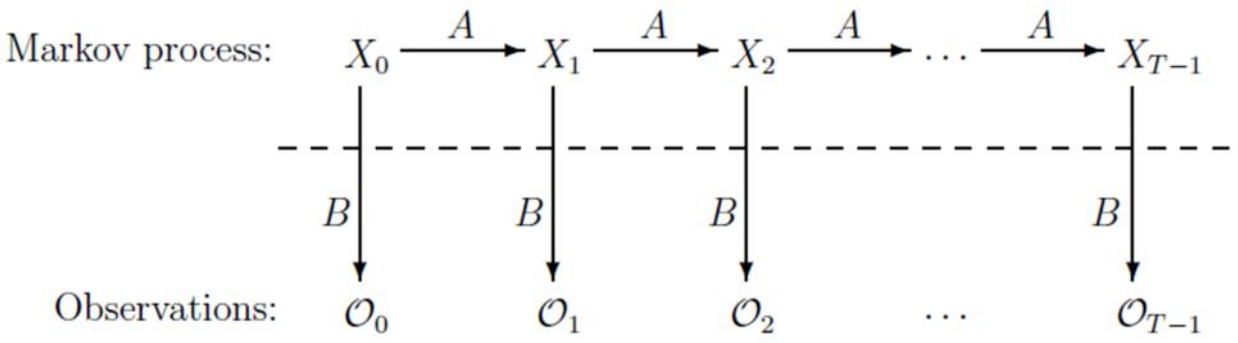

Figure 10 : HMM Model [23]

In this model X_0, $X_1 \ldots X_{T-1}$ denotes the hidden state sequence. The hidden Markov process itself is shown above the hidden line while the observations O represent the observations corresponding to the hidden states and are connected using B (the observation probability matrix)

4.1.2 Example

Consider the example from Prof Stamp's paper [23], imagine the following scenario. Consider the problem where we want to determine the average annual temperature at a specific geographic location over a period of time. To add to the challenge, let us assume that during the period of time under question there was no accurate way to determine the temperature. As

temperatures from the past are unavailable we try to look for an indirect sign which could indicate the temperature prevalent at the time under consideration.

Let there be only two annual temperature descriptions, either ‘hot’ or ‘cold’. Suppose the scientists have, through evidence, come up with the probability of 0.6 for a cold year to be successively followed by a cold year as well as the probability of 0.7 for a hot year to be successively followed by a hot year. This information can be denoted in matrix form as

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{cc} H & C \end{array} \\ \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \end{array}$$

Figure 11: State Transitions in Matrix form [23]

Hence our state transition matrix will look like

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$$

The scientists further discover that there is a relation between the thickness of tree rings for a year and the temperature prevalent at that time. The thickness of tree ring sizes is denoted by Small (S), Medium (M) or Large (L), and using the information provided by the scientists we can represent it as

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{ccc} S & M & L \end{array} \\ \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix} \end{array}$$

Figure 12 : Observation information in matrix form [23]

Hence our observation probability matrix B will be

$$B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$$

Suppose initial state distribution (i.e – Probability of starting in “hot” or “cold”) is denoted by

$$\pi = [\ 0.6 \quad 0.4\ ]$$

Now suppose we have a sequence of observations of tree ring thickness.

O = { S, M, S, L }

Let S = 0, M = 1, L = 2

Hence our observation sequence is

O = { 0, 1, 0, 2 }

Now, we want to find out the most likely sequences of annual temperatures that lead to this observation sequence.

Consider a generic state sequence as follows:

$X = \{ X_1, X_2, X_3, X_4 \}$

and the corresponding observations as

$O = (O_0, O_1, O_2, O_3)$.

With the above assumptions, the probability of sequence X occurring is given by

$$P(X) = \pi_{x0} b_{x0}(\mathcal{O}_0) a_{x0,x1} b_{x1}(\mathcal{O}_1) a_{x1,x2} b_{x2}(\mathcal{O}_2) a_{x2,x3} b_{x3}(\mathcal{O}_3).$$

Figure 13: Probability of sequence X occurring [23]

Where

π_{X0} denotes the probability of the sequence starting in state X_0

B_{X0} (O_0) denotes the probability of the first observation being O_0

$a_{x0, x1}$ denotes the probability of the state transitioning from state X_0 to X_1

Substituting the values from A, B, π and O IN the above equation, the probability of finding a HCCC sequence is

P (HCCC) = 0.6 *(0.1)*(0.3)*(0.2)*(0.6)*(0.7)*(0.6)*(0.1) = 0.000091

We then calculate the probabilities for all the possible state sequences for the given observation sequence. The result of all these, along with a separate column with normalized values, is as shown in Table 1.

To find the most probable sequence we need to look at each position independently and find out which state H or C gives higher probability for that particular position. We then choose the state with the highest probability as the optimal state for that position. As an example, the choice of symbol for the first position in the sequence is either H or C. We add the normalized probabilities of all sequences starting with H as well as for C (calculated as mod (1 – P (H))). The one which has the higher probability is then chosen as the best choice for that position. From Table 1 the sum of normalized probabilities for H is 0.188182, and hence that for C is (1 – 0.18812); which is 0.811818. The calculated probabilities for each position will be as shown in Table 2.

Table 1: Probabilities [23]

State	Probability	Normalized Probability
HHHH	0.000412	0.042787
HHHC	0.000035	0.003635
HHCH	0.000706	0.073320
HHCC	0.000212	0.022017
HCHH	0.000050	0.005193
HCHC	0.000004	0.000415
HCCH	0.000302	0.031364
HCCC	0.000091	0.009451
CHHH	0.001098	0.114031
CHHC	0.000094	0.009762
CHCH	0.001882	0.195451
CHCC	0.000564	0.058573
CCHH	0.000470	0.048811
CCHC	0.000040	0.004154
CCCH	0.002822	0.293073
CCCC	0.000847	0.087963

Table 2: Final Results [23]

	element			
-	0	1	2	3
P(H)	0.188182	0.519576	0.228788	0.804029
P(C)	0.811818	0.480424	0.771212	0.195971

Using Table 2, for each position we can now pick up the state with the highest possibility.

Hence the optimal state sequence will be CHCH.

4.2 Using HMM to detect malware

Wong and Stamp [2] have earlier explored the use of HMM's to detect malware with success. The hidden Markov models (HMMs) are first trained, and then are used to detect malware. Wong and Stamp use the threshold approach [2] using HMM.

The threshold model working can be summarized as follows –

1. Train a single HMM using known malware file op code sequences.
2. Observe the results over multiple runs and find a threshold value such that all files scoring below the threshold can be classified as benign, while the ones scoring above the threshold will be classified as malware.
3. Once the threshold has been defined, files to be classified are scored using the trained HMM. On the basis of the score the file can be classified as either malware or benign.

4.3 Dueling HMM

While the threshold approach achieves good results against many metamorphic viruses as demonstrated by Wong and Stamp in [2], others prove more problematic. In particular, MetaPHOR [4] and MWOR [3] are able to defeat the threshold approach and blend in with benign files.

The dueling HMM strategy addresses these challenges by introducing multiple HMMs. Dueling HMM was first explored by Thomas Austin, Eric Filiol, Sebastian Jose and Mark Stamp in “Exploring Hidden Markov Models for Virus Analysis: A Semantic Approach” [4]. The paper further explores the idea of using HMM to detect malware.

The dueling HMM strategy [4] was developed to create more accurate results than the threshold approach mentioned in section 4.2. The term *dueling* indicates that multiple HMMs are fighting against each other in order to decide a winner. The dueling HMM strategy differs from the threshold approach as the threshold approach relies on a single HMM to help classify the files, while the dueling HMM relies on multiple HMMs to help classify the files.

The mechanics of a dueling HMM strategy can be summarized as follows –

1. Train multiple HMMs (number of HMMs will be the same as the number of malware and benign families). Thus there will be a corresponding HMM for every benign code as well as for every malware family code.
2. The file to be classified is scored against each of the HMMs individually and the scores are recorded.
3. The file is classified on the basis of the dueling HMM scores. The file is classified as belonging to the code of the HMM which scored the highest.

As a result of a new file being needed to be scored against multiple HMMs, the time required to classify the file increases as it needs to be compared against 'N' HMMs (where N is the number of distinct HMMs in the dueling approach). The results for the dueling approach improve as compared to the threshold approach.

One limitation of the dueling HMM strategy is that it does not easily allow for tuning. The HMM that returns the highest probability is always used. In contrast, the threshold value in the threshold approach can be adjusted in a straightforward manner to reduce either the false positives or the false negatives.

In section 5.3 we illustrate how the dueling HMM strategy may be adjusted to allow for better tuning of the results.

CHAPTER 5

Design and Implementation

5.1 Introduction

We implement a tiered approach in which a file has to go through multiple layers of trained HMMs. Previous research [4] has shown that the use of individual HMMs can improve the results as compared to a single HMM threshold based approach. We explore this further by introducing an HMM for every malware family. The goal is to use the threshold approach to help reduce the number of files that are actually passed on to the dueling layer, thus increasing efficiency.

With the introduction of additional HMMs in the layered approach, the efficiency in terms of time decreases as the file needs to be tested against a greater number of HMM models to classify it as either good or bad. The challenge is to reduce the time taken to classify the file; using the threshold model, we filter out the “definitely good” and the “definitely bad” files leaving a small number of files in the gray area which need to be evaluated with the dueling HMM approach [4].

5.2 Design

To minimize the time taken to classify a file we came up with a tiered approach which is illustrated in Figure 14. The approach consists of two distinct layers: the threshold tier and the dueling tier. The file to be classified is initially processed by the first tier, the threshold HMM tier, to see whether it can be classified. If the file cannot be eliminated, it is then sent to the second tier, the dueling HMM tier for classification. The description of the tiers is given below.

Tier 1:

Tier 1 will contain an HMM which will be trained with all of the malware families. The goal of this tier is to quickly eliminate a good chunk of viruses by the use of the threshold technique. The log likelihood per op code is determined for each file and is compared against the threshold. The files that cannot be satisfactorily classified are passed on to the next tier.

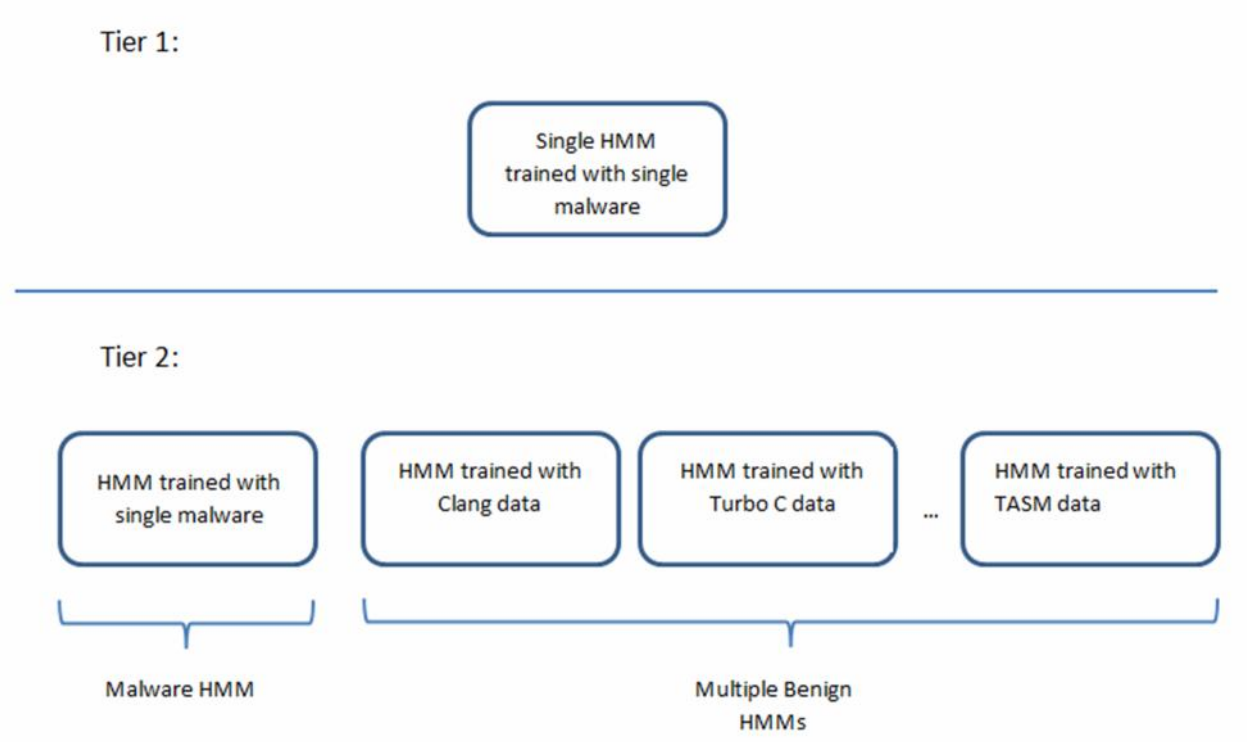

Figure 14: Design

Tier 2:

This tier contains the highest number of HMMs. This tier will have individual HMMs for each virus family as well as all benign code. Thus, each suspected file can now be inspected to

see whether it belongs to a particular category. The file will belong to the category of the HMM which will give it the highest score.

5.3 Biasing the Dueling HMM Strategy

As discussed in Section 4.3 we use biasing. Biasing provides the necessary flexibility to fine tune the dueling model to produce better results in terms of false positives / false negatives. We apply a biasing of -0.25 to the G2 HMM and a biasing of -0.44 to the MPCGEN HMM. Table 3 and Table 4 show how the false positives and false negatives vary with different values of biasing applied for the HMM. '0.0' represents the case where no biasing is present.

Table 3: G2 - Results varying by applied bias

Bias	False Positive	False Positive (%)	False Negative	False Negative (%)
0.00	2/370	0.54	0/50	0
-0.05	2/370	0.54	0/50	0
-0.10	2/370	0.54	0/50	0
-0.15	1/370	0.27	0/50	0
-0.20	1/370	0.27	0/50	0
-0.25	0/370	0	0/50	0

In Table 3, we vary the bias applied from '0' to '-0.25' in increments of 0.5, to see how the biasing changes the result. As can be seen in the Table 3 the false positives gradually decrease from 2 to 0 while the false negative rates stay constant at 0.

Similarly, in Table 4, we see the results corresponding to the bias applied. The bias in case of MPCGEN was varied from '0.0' to '-0.44' in increments of 0.05. The number of false positives for MPCGEN decreases from 4 for no biasing to 0 with a biasing of '-0.44'.

Table 4: MPCGEN – Results varying by applied bias

Bias	False Positive	False Positive (%)	False Negative	False Negative (%)
0.00	4/370	1.08	0/50	0
-0.05	4/370	1.08	0/50	0
-0.10	3/370	0.81	0/50	0
-0.15	3/370	0.81	0/50	0
-0.20	2/370	0.54	0/50	0
-0.25	2/370	0.54	0/50	0
-0.30	2/370	0.54	0/50	0
-0.35	1/370	0.27	0/50	0
-0.40	1/370	0.27	0/50	0
-0.44	0/370	0	0/50	0

5.4 Implementation

This section includes what datasets were used, how they were generated and how the HMMs are created.

5.4.1 System Details

The specifications of the host system used are given below.

Type	Specification
Processor	Intel Core i5-3210M CPU @ 2.5GHz
RAM	8.00 GB
System Type	Windows 7, 64 – bit operating System

The Linux system that was used was running Ubuntu 12.04. Ubuntu was run on the host machine using VMware player (version 5.0.2). The specifications for Ubuntu are as follows:

Type	Specification
Processor	Intel Core i5-3210M CPU @ 2.5GHz
RAM	1.00 GB
System Type	Ubuntu 12.04 , 64 – bit operating System

5.4.2 Data set

The data sets used in the implementation of this project include the benign data sets for Clang [35], Cygwin [37], GCC [38], MingW [40], TASM [39], Turbo C [41] as well as the set of benign files used by MWOR to insert dead code in order to morph.

We included benign data sets for GCC, MingW and Clang as these are the more popular compilers that are used. We also included the dataset for TASM as MetaPHOR is compiled using TASM. Additionally it also serves to help compare hand written code with compiled code. As mentioned above, among other techniques, MWOR pulls code from 20 specific Linux executables. Hence these Linux executables were disassembled and used as a data set. Previous experiments [2] use Cygwin utility executables to test with NGVCK; hence the Cygwin dataset was added as well. The benign datasets used were a part of previous experiments carried out by Dr. Austin et al. [4]

The malware data set consisted of MetaPHOR [27], MPCGEN [20], G2 [18], NGVCK [19], and MWOR [3], with padding ratios of 1, 2, 3 and 4, represented as MWOR (PR 1), MWOR (PR 2), MWOR (PR 3), MWOR (PR 4) respectively. The malware sets used are from previous experiments carried out under the guidance of Dr. Stamp [2] [3]. The MetaPHOR dataset was used in previous experiments by Dr. Austin et al. in [4].

We chose to include malware datasets G2, NGVCK and MPCGEN as these were previously used in experiments by Wong and Stamp [2]. MetaPHOR was chosen since it is particularly challenging in terms of the techniques used by it to remain undetected. Since MWOR has been shown to be able to evade detection [3] by the threshold approach we decided to include the MWOR dataset as well.

Table 5: File distribution

Benign

Dataset	**Number of files**
Clang	73
Cygwin	16
GCC	74
MingW	72
TASM	56
Turbo C	64
Linux Utilities	16

Malware

Dataset	**Number of files**
G2	50
MPCGEN	50
NGVCK	200
MetaPHOR	60
MWOR (PR 1)	100
MWOR (PR 2)	100
MWOR (PR 3)	100
MWOR (PR 4)	100

Once the disassembled version of these files was obtained they were then processed to give a sequence of numbers. This was done by assigning a fixed number to an op code and whenever that op code was encountered during the processing it would be replaced by its numerical equivalent.

The number of files belonging to each dataset used in the project is shown in Table 5.

5.4.3 Training HMMs

To train HMMs we use the Jahmm library [43]. The HMM is trained using files from the above datasets. Once the HMM has been trained it still needs some modifications to be made to it so that it is feasible to use it. In order to account for the HMM giving odd results for op codes not encountered in training, we smoothed the model. There are various ways in which smoothing can be carried out including, but not limited to, Additive Smoothing / Laplacian smoothing [44], exponential smoothing [45] etc. We use the additive smoothing method to smooth. We add a small probability ($1 * 10^{-6}$) and then divide every individual probability by the sum of the changed probability.

The scores produced by the HMM at this point will be the log likelihood scores. However, as explained by Sudarshan [3], log likely hood scores depend on the length of the sequence as well. Hence, larger files will score higher. To mitigate this, we take the length of the sequence into account. To make the scores independent of the length of the files we divide the score by the number of op codes in the sequence, similar to the procedure followed by Sudarshan in [3].

5.4.4 Clustering

A cluster consists of objects which are found to be similar and hence are grouped together. As mentioned in “Algorithms for Clustering” [49], “A cluster is an aggregation of points in the test space such that the distance between any two points in the cluster is less than the distance between any point in the cluster and any point not in it”. It essentially means that points in a cluster are closer to each other than they are to other points outside the cluster. The

extent to which clustering can be helpful depends on the type of data as well. Clustering helps surface patterns that are not directly visible. Clustering can be used to find similarities between data points which consist of multiple dimensions.

We use K-means to cluster the files passed from tier 1 to tier 2. K-means is an easy and fast way to cluster data. It uses Lloyd's algorithm. K-means clustering works by ultimately dividing the given dataset into K subsets. Each of these subsets is initialized either by using a random point or using specific ways like *uniform* initialization. These points are the initial centroids say (C_{a1}, C_{a2}, C_{a3}, C_{a4}…C_{ak}).

For every record in the dataset, the distance of the record from each of the centroids is calculated and the record is classified as belonging to the cluster whose centroid is at the least distance from the point. Once the point has been added to the cluster the centroid of that cluster is recalculated to adjust for the most recently added record. Once all the data points have been classified we use the centroid values from the above process. Let the centroids that we get after one iteration be (C_{b1}, C_{b2}, C_{b3}, C_{b4}…$C_{bk)}$.

Each of the records from the dataset is then scored against these new centroids (C_{b1}, C_{b2}, C_{b3}, C_{b4}…C_{bk}). At the end of each iteration new centroids are generated which are again used to classify the records from the dataset. This readjusting of the centroids and the clusters around them goes on till convergence is reached or if the change in centroid values is small.

The scikit-learn [48] library in Python was used to carry out K-means clustering. We used Silhouette scores as a measure of quality of the clusters. In section 6.2.3.1 we have a look at the results for k-means clustering with different number of centroids.

CHAPTER 6

Experiments and Results

We experiment with three different approaches, the threshold approach [2], the dueling approach [4] and the tiered approach. We also compare and contrast the results of the three approaches in terms of execution time, as well as in terms of the false positives and false negatives.

In our experiments we use five-fold cross validation. The data for each of the data sets is divided into five parts. Out of these five parts we choose four parts to train the HMM and the remaining part is used to test the trained HMM. We do this for every subset of data. Thus, we carry out tests such that every one of the subsets is tested with an HMM that has been trained with the other four subsets.

Using the four parts of the data the HMM is created using the k-means learner module. We look at each of these in detail in the following sections.

6.1 Data sets

The data sets we use include the benign datasets of Cygwin utilities as well as Linux executables. Linux utilities were included as MWOR pulls code from these files to disguise itself as a benign file. These datasets are explained in more detail in Section 5.4.2.

6.2 Experiments

In the following sections we look at how the experiments were carried out for the different approaches.

6.2.1 Threshold Approach

In this approach, for every malware family code we train an HMM using four subsets of the malware code and then use the trained HMM to score the fifth subset of the malware code as well as all the benign code. Once the file has been scored in terms of LLPO the score is checked to see where it stands in relation with the threshold. The closer the score is to zero, higher the probability of the file being a malware file. If the score is above the threshold value (towards the direction of zero value) the file is classified as a malware and if the score is below the threshold the file is classified as benign.

Table 6: Results for Threshold Approach

Threshold						
	False Positives	False Positives (%)	False Negative	False Negative (%)	Threshold	Total time (milliseconds)
G2	62/370	16.75676	4/50	8	-2.66659	548.224
MPCGEN	89/370	24.05405	0/50	0	-2.90398	495.467
MetaPHOR	198/370	53.51351	10/60	16.67	-2.65207	538.665
NGVCK	166/370	44.86486	16/200	8	-3.57961	550.97
MWOR (PR 1)	0/370	0	0/100	0	-2.6251	866.268
MWOR (PR 2)	1/370	0.27027	0/100	0	-2.5394	964.157
MWOR (PR 3)	4/370	1.081081	0/100	0	-2.63274	1152.988
MWOR (PR 4)	4/370	1.081081	0/100	0	-2.60028	1341.365

Results in Table 6 show that the threshold approach does better for G2 and MPCGEN, with MetaPHOR being particularly hard to detect. These results are different from Wong's and Stamp's [2] results due to the fact that we use a benign dataset which is different from the one used in their experiments.

6.2.2 Dueling Approach

In the dueling approach an HMM is trained for the malware code as in the threshold approach. Additionally, we build similar HMMs for every benign dataset. Once the HMMs have been built, every file is scored against the malware HMM as well as the benign HMMs. The file will have multiple scores, each associated with one HMM. The file is then classified on the basis of the HMM that generated the highest score for that file. Once the HMM which generates the highest score for the file being classified is identified, if the HMM was built from malware data then the file is classified as malware otherwise it is classified as a benign file.

In case of G2 and MetaPHOR we add a penalty / bias of '-0.25' and '-0.44' respectively, to all scores generated by the malware HMM, in order to eliminate any false positives that may be generated.

Table 7: Results for the Dueling Approach with biasing

Dueling						
Family	False Positives	False Positives (%)	False Negatives	False Negatives (%)	Average time (millisec)	Bias
G2	0/370	0	0/50	0	1030.566	- 0.25
MPCGEN	0/370	0	0/50	0	981.1332	- 0.44
MetaPHOR	4/370	1.081081	9/60	15	1113.473	-
NGVCK	2/370	0.540541	75/200	37.5	1155.605	-
MWOR (PR 1)	0/370	0	0/100	0	2019.35	-
MWOR (PR 2)	0/370	0	0/100	0	2413.054	-
MWOR (PR 3)	0/370	0	0/100	0	2897.533	-
MWOR (PR 4)	0/370	0	0/100	0	3433.045	-

Table 7 shows that the results improve considerably in terms of the false negatives and the false positives. However, as a side effect of each file being scored against multiple HMMs the files take longer to be classified.

6.2.3 Tiered Approach

In the tiered approach we use the threshold approach in conjunction with the dueling approach to achieve comparable results. The aim is to eliminate as many files as can be eliminated using the threshold approach (Tier 1), thus needing the dueling approach (Tier 2) to classify a smaller number of files, leading to considerable gain in terms of the execution time to classify the file.

We first use the threshold approach to score the files and eliminate any "definitely good" files, files which score below the threshold. The threshold is different for different malware code. The files which cannot be eliminated using the threshold approach are then scored using the dueling HMM approach.

Table 8: Results for Tiered Approach

Tiered								
	False Positive	False positive (%)	False Negative	False Negative (%)	Threshold	Eliminated at tier1	Bias	Time (millisecs.)
G2	0/370	0	0/50	0	-2.6739625	249/420	-0.25	613.258992
MPCGEN	0/370	0	0/50	0	-2.9039755	281/420	-0.44	603.359175
MetaPHOR	4/370	0.1	9/60	15	-2.7918600	152/430	-	860.566666
NGVCK	2/370	0.5405	75/200	37.5	-3.6203424	209/570	-	736.999253
MWOR (PR 1)	0/370	0	0/100	0	-2.6251045	370/470	-	1512.64964
MWOR (PR 2)	0/370	0	0/100	0	-2.5393950	369/470	-	1968.82529
MWOR (PR 3)	0/370	0	0/100	0	-2.6096006	366/470	-	2531.71884
MWOR (PR 4)	0/370	0	0/100	0	-2.5927906	366/470	-	3187.04291

For this approach, the threshold for Tier 1 was chosen in such a way that there were no false negatives except in the case of MetaPHOR and NGVCK, and hence they differ from the thresholds chosen in the standalone threshold approach.

Table 9: Threshold tier in the tiered approach

Threshold							
	False Positive	False Positive (%)	False Negative	False Negative (%)	Threshold	Eliminated	Total time(millisec)
G2	NA/370	0	0/50	0	-2.6739625	249/420	548.224
MPCGEN	NA/370	0	0/50	0	-2.903975	281/420	495.467
MetaPHOR	NA/370	0	0/60	0	-2.791860	152/430	538.665
NGVCK	NA/370	0	2/200	1	-3.620342	209/570	550.97
MWOR (PR 1)	NA/370	0	0/100	0	-2.625104	370/470	866.268
MWOR (PR 2)	NA/370	0	0/100	0	-2.539395	369/470	964.157
MWOR (PR 3)	NA/370	0	0/100	0	-2.632744	368/470	1152.988
MWOR (PR 4)	NA/370	0	0/100	0	-2.600275	368/470	1341.365

6.2.3.1 Cluster Analysis

In this section we analyze the results of clustering for files passed to tier 2 from tier 1. Results with different centroid numbers are available in Appendix D. For clustering with 5 centroids, the distribution of malware files and benign files is given in Table 10. The detailed information of how every file is clustered is represented in Table 11.

Table 10: 5 Centroids, Benign - Malware Distribution by Clusters

Cluster -> Type	0	1	2	3	4
Benign	421	3	152	4	11
Malware	4	396	60	98	162

Table 10 shows that there were 4 malware files which are clustered into cluster 0 while there are 421 benign files which were clustered into the same cluster.

Table 11: 5 Centroids, Detailed distribution by Cluster

	Zero	One	Two	Three	Four
Linux_Utilities	0	3	0	0	0
Clang	266	0	0	0	0
GCC	148	0	45	0	0
MinGW	0	0	87	0	0
TASM	3	0	8	2	10
Turbo C	4	0	12	0	1
G2	0	0	0	50	0
MetaPHOR	0	0	60	0	0
MPCGEN	0	0	0	50	0
MWOR (PR 1)	1	99	0	0	0
MWOR (PR 2)	1	99	0	0	0
MWOR (PR 3)	1	99	0	0	0
MWOR (PR 4)	1	99	0	0	0
NGVCK	0	0	0	0	162

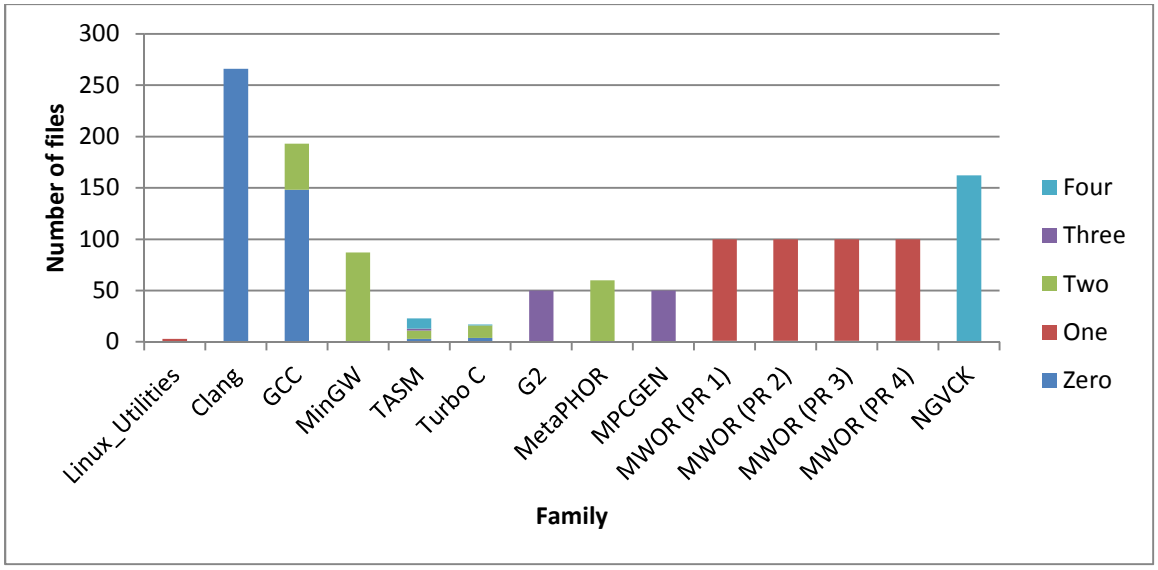

Figure 15: 5 Centroids, Classification by family

In Table 11, cluster 3 has 3 Linux_Utility files and 396 MWOR files clustered together as MWOR inserts dead code pulled from these Linux Utility files. Clusters 1, 3 and 4 mostly contain mainly contains malware files while clusters 0 and 2 mainly contain all benign files. The distribution of the files among clusters is illustrated in Figure 15 and Figure 16.

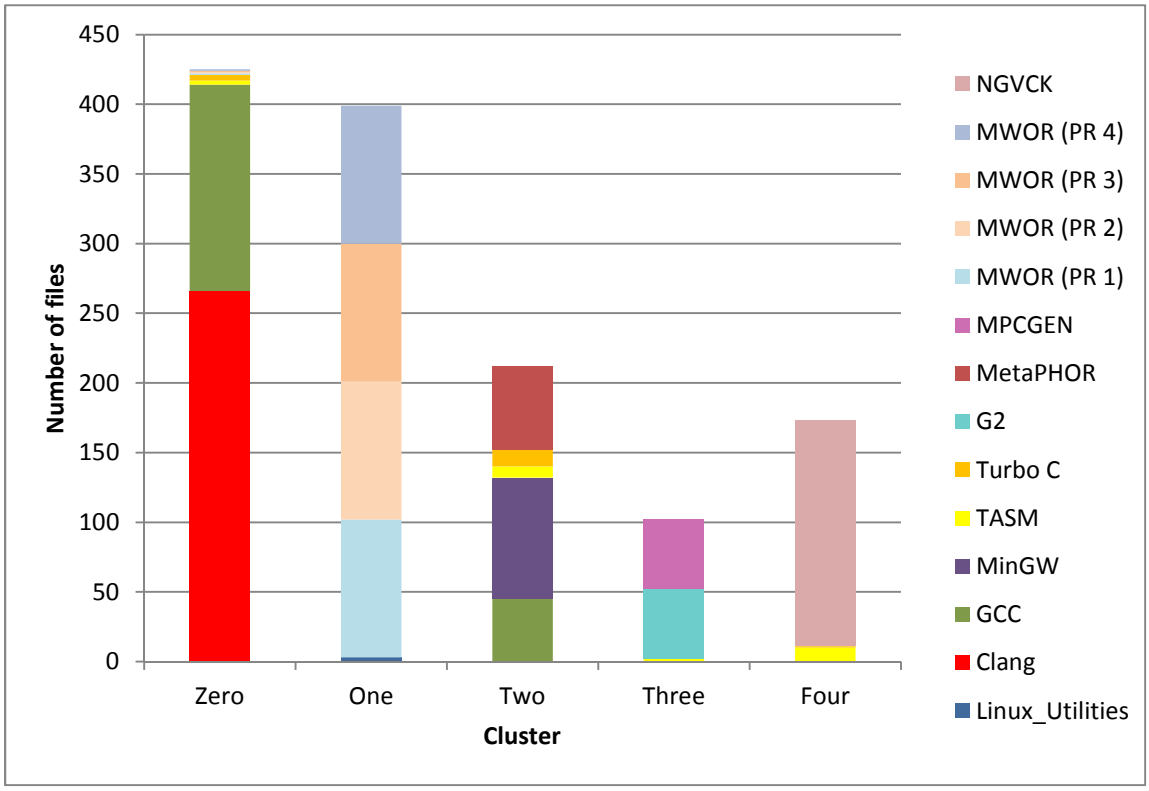

Figure 16: 5 Centroids, Cluster Composition

6.3 Results

In this section we examine the results based on the time taken as well as the accuracy of each of the approaches.

6.3.1 Comparison: Time

Table 12 lists out the time taken in milliseconds by each approach to classify every malware family. Table 12 shows that the threshold approach is the fastest method followed by

the tiered approach. The dueling approach, as expected, is the slowest of the three. The execution time for the dueling approach is on average two to three times the execution time of the threshold approach. The tiered approach lies snugly in the middle with the execution time ranging from 1.1 times – 2.5 times the time taken by the threshold approach, with most of the execution times for the tiered model approaching the execution time for the threshold model.

Table 12: Comparison - Performance in terms of time

Timing comparison (in Seconds)			
Family	Threshold (sec)	Dueling (sec)	Tiered (sec)
G2	548.224	1030.566	613.25899
MPCGEN	495.467	981.133	603.35918
MetaPHOR	538.665	1113.473	860.56667
NGVCK	550.970	1155.605	736.99925
MWOR (PR 1)	866.268	2019.350	1512.64965
MWOR (PR 2)	964.157	2413.054	1968.82529
MWOR (PR 3)	1152.988	2897.533	2531.71885
MWOR (PR 4)	1341.365	3433.045	3187.04291

6.3.2 Comparison: False Positive and False Negative

Table 13: Comparison - False positives and false negatives

Approach	Threshold		Dueling		Tiered	
	FP	FN	FP	FN	FP	FN
G2	62/370	4/50	0/370	0/50	0/370	0/50
MPCGEN	89/370	0/50	0/370	0/50	0/370	0/50
MetaPHOR	198/370	10/60	4/370	9 /60	4/370	9/60
NGVCK	166/370	16/200	2/370	75/200	2/370	75/200
MWOR (PR 1)	0/370	0/100	0/370	0/100	0/370	0/100
MWOR (PR 2)	1/370	0/100	0/370	0/100	0/370	0/100
MWOR (PR 3)	4/370	0/100	0/370	0/100	0/370	0/100
MWOR (PR 4)	4/370	0/100	0/370	0/100	0/370	0/100

Table 13 shows how the different approaches compare against each other in terms of false positives and false negatives for different malware families. The results from Table 13 show that in terms of false positives and false negatives, the tiered approach is as good as the dueling approach with the exception of MetaPHOR where the tiered approach succeeds in eliminating all false positives. Surprisingly, the threshold approach eliminates the files which might cause the dueling approach to cause false positives.

6.3.3 Consolidated Comparison

Table 14, clearly shows that the threshold approach is the best approach in terms of time with the tradeoff being the performance in terms of the false positives and false negatives. The dueling model is much more accurate than the threshold model however the execution time required is large. The tiered model on the other hand takes the best from both approaches and comes close to matching the time performance of the threshold while providing false positive / false negative performance which is comparable to the dueling approach.

Table 14: Consolidated comparison - False positives (FP), false negatives (FN), time

Approach	Threshold					Dueling					Tiered				
Malware Family	FP	FP (%)	FN	FN (%)	Time (millisec.)	FP	FP (%)	FN	FN (%)	Time (millisec.)	FP	FP (%)	FN	FN (%)	Time (millisec.)
G2	62/370	16.75676	4/50	8	548.223929	0/370	0	0/50	0	1030.56631	0/370	0	0/50	0	613.259
MPCGEN	89/370	24.05405	0/50	0	495.466803	0/370	0	0/50	0	981.13317	0/370	0	0/50	0	603.359
MetaPHOR	198/370	53.51351	10/60	16.67	538.664901	4/370	1.08	9/60	15	1113.47348	4/370	0.1	9/60	15	860.567
NGVCK	166/370	44.86486	16/200	8	550.969502	2/370	0.54	75/200	37	1155.60505	2/370	0.54	75/200	37	736.999
MWOR (PR 1)	0/370	0	0/100	0	866.268014	0/370	0	0/100	0	2019.34958	0/370	0	0/100	0	1512.65
MWOR (PR 2)	1/370	0.27027	0/100	0	964.157236	0/370	0	0/100	0	2413.05431	0/370	0	0/100	0	1968.825
MWOR (PR 3)	4/370	1.081081	0/100	0	1152.98839	0/370	0	0/100	0	2897.53263	0/370	0	0/100	0	2531.719
MWOR (PR 4)	4/370	1.081081	0/100	0	1341.365078	0/370	0	0/100	0	3433.04472	0/370	0	0/100	0	3187.043

CHAPTER 7

Conclusion and Future Work

This project proposed the idea of combining different approaches in order to detect malware in a more efficient way in terms of time. In this experiment, we implemented the threshold model from Wong and Stamp [2] as well as expanded the dueling HMM [4] approach to include an HMM for each malware family code. We also implemented a tiered model which uses the threshold approach along with the dueling approach. Finally, we illustrated how to apply a bias to the dueling HMM strategy, allowing for tuning of the results.

The experiments were carried out on a metamorphic malware like MetaPHOR and MWOR with different padding ratios, as well as virus construction kits including G2, NGVCK, and MPCGEN.

The tiered model takes the best from both approaches in an effort to help detect malware efficiently. The results from the experiments that were carried out during the course of the project show that the tiered approach is always faster than the dueling approach. The improvement ranges from almost 40% improvement to a minimum improvement of 8% percent in terms of time.

Although the tiered model is faster, unlike the threshold model, it does not sacrifice on accuracy. The accuracy for the tiered approach is the same as the dueling HMM approach.

The tiered approach was able to correctly identify the malware files with an accuracy of 89.07% while keeping the false positive rate to a minimum. Thus we are able to achieve the results comparable to dueling HMM without sacrificing on the performance in terms of time.

An additional way to speed up the tiered approach would be to find a way that would also be able to eliminate the ‘definitely bad’ files. ‘Definitely bad’ is a file such that it’s LLPO score is so close to zero that it can definitely be classified as being a malware file.

LIST OF REFERENCES

[1] Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Pearson Education.

[2] Wong, Wing. (2006), *Analysis and Detection of Metamorphic Computer Viruses*. Master's Projects. Paper 153.Retrieved from http://scholarworks.sjsu.edu/etd_projects/153

[3] Sridhara, S. M., & Stamp, M. (2013). Metamorphic worm that carries its own morphing engine. *Journal of Computer Virology and Hacking Techniques*, *9*(2), 49-58.

[4] Austin, T. H., Filiol, E., Josse, S., & Stamp, M. (2013, January). Exploring hidden Markov models for virus analysis: A semantic approach. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on* (pp. 5039-5048). IEEE..

[5] Aycock, J. (2006). *Computer Viruses and malware* (Vol. 22). Springer.

[6] Stamp, M. (2011). *Information security: principles and practice*. John Wiley & Sons.

[7] Konstantinou, E., & Wolthusen, S. T. E. P. H. E. N. (2008). Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, *15*.

[8] Ször, P., & Ferrie, P. (2001, September). Hunting for metamorphic. In *Virus Bulletin Conference*.

[9] Symantec. *Viruses, worms, and Trojans* (2010). Retrieved from http://service1.symantec.com/support/nav.nsf/docid/1999041209131106

[10] Kannan, J., Subramanian, L., Stoica, I., & Katz, R. H. (2005, July). Analyzing cooperative containment of fast scanning worms. In *Proceedings of Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)* (pp. 17-23).

[11] Idika, N., & Mathur, A. P. (2007). A survey of malware detection techniques. *Purdue University*, 48.

[12] Desai, P. (2008). , *Towards an Undetectable Computer Virus*. Master's Projects. Paper 90. Retrieved from http://scholarworks.sjsu.edu/etd_projects/90

[13] Govindaraju, A. (2010). *Exhaustive Statistical Analysis for Detection of Metamorphic Malware*. Master's Projects. Paper 66. Retrieved from http://scholarworks.sjsu.edu/etd_projects/66

[14] Balakrishnan, A., & Schulze, C. (2005). Code obfuscation literature survey.*CS701 Construction of Compilers*, *19*.

[15] Christodorescu, M., & Jha, S. (2006). *Static analysis of executables to detect malicious patterns*. Wisconsin University. Department of Computer Sciences.

[16] Tamboli T. (2013). *Metamorphic Code Generation from LLVM IR Byte code*, Master's Projects. Paper 301. Retrieved from http://scholarworks.sjsu.edu/etd_projects/301

[17] Owens, R., & Wang, W. (2011, November). Non-normalizable functions: A new method to generate metamorphic malware. In *Military Communications Conference, 2011-Milcom 2011* (pp. 1279-1284). IEEE.

[18] G2. VX Heavens. Retrieved from http://download.adamas.ai/dlbase/Stuff/VX%20Heavens%20Library/static/vdat/creatrs1.htm

[19] NGVCK. VX Heavens, Retrieved from *http://vxheaven.org/vx.php?id=tn02*

[20] MPCGEN . VX Heavens, *http://vxheaven.org/vx.php?id=tm02*

[21] Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257-286.

[22] Wong, W., & Stamp, M. (2006). Hunting for metamorphic engines. *Journal in Computer Virology*, *2*(3), 211-229.

[23] Stamp, M. (2004). A revealing introduction to hidden Markov models. *Department of Computer Science San Jose State University*.

[24] Attaluri, S., McGhee, S., & Stamp, M. (2009). Profile hidden Markov models and metamorphic virus detection. *Journal in computer virology*, *5*(2), 151-169.

[25] Lin, D., & Stamp, M. (2011). Hunting for undetectable metamorphic viruses. *Journal in computer virology*, *7*(3), 201-214.

[26] S. Venkatachalam. (2010). *Detecting undetectable computer viruses*. Master's Projects, Paper 156. http://scholarworks.sjsu.edu/etd_projects/156.

[27] MetaPHOR. Retrieved from http://spth.virii.lu/29a6/29A-6.602.txt

[28] Metamorphism in practice or "How I made MetaPHOR and what I've learnt", http://vxheavens.com/lib/vmd01.html

[29] Zperm. VX Heaven. Retrieved from http://vxheaven.org/vl.php?dir=Virus.Win32.ZPerm

[30] SecureList. *The Evolution of Technologies used to Detect Malicious Code.* Retrieved from https://www.securelist.com/en/analysis/204791972/The_evolution_of_technologies_used_to_detect_malicious_code#emulation

[31] Symantec, *"VBA Emulation: A Viable Method of Macro Virus Detection?" Part One*, http://www.symantec.com/connect/articles/vba-emulation-viable-method-macro-virus-detection-part-one

[32] Priyadarshi,S. (2011). *Metamorphic detection via Emulation*, Master's Project. Paper 177. http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1176&context=etd_projects

[33] Al Daoud, E., Jebril, I. H., & Zaqaibeh, B. (2008). Computer virus strategies and detection methods. *Int. J. Open Problems Compt. Math*, *1*(2), 12-20.

[34] Walenstein, A., Mathur, R., Chouchane, M. R., & Lakhotia, A. (2007, March). The design space of metamorphic malware. In *Proceedings of the 2nd International Conference on i-Warfare & Security (ICIW)* (pp. 241-248).

[35] Clang. Retrieved from http://clang.llvm.org/

[36] W32 / Zmist. Retrieved from http://vxheaven.org/vl.php?dir=Virus.Win32.ZMist

[37] Cygwin. Retrieved from http://www.cygwin.com/

[38] GCC. Retrieved from http://gcc.gnu.org/

[39] TASM. Retrieved from http://trimtab.ca/2010/tech/tasm-5-intel-8086-turbo-assembler-download

[40] MinGW. Retrieved from http://www.mingw.org/

[41] Turbo C, Retrieved from http://edn.embarcadero.com/article/20841

[42] Annachhatre, Chinmayee. (2013). *Hidden Markov Models for Malware Classification*. Master's Projects. Paper 328. Retrieved from http://scholarworks.sjsu.edu/etd_projects/328

[43] François, J. M. (2006). Jahmm-hidden markov model (hmm): an implementation in Java**.** http://www.run.montefiore.ulg.ac.be/~francois/software/jahmm/

[44] Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval* (Vol. 1, p. 6). Cambridge: Cambridge university press.

[45] Natrella, M. (2010). NIST/SEMATECH e-handbook of statistical methods.

[46] PSMPC. VX Heaven. Retrieved from http://vxheaven.org/vx.php?lang=de&id=tp00

[47] NEG. VX Heaven. Retrieved from http://vxheaven.org/vl.php?dir=Virus.MSExcel.Neg

[48] Scikit-learning. http://scikit-learn.org/stable/

[49] Jain, A. K., & Dubes, R. C. (1988). *Algorithms for clustering data*. Prentice-Hall, Inc.

APPENDIX A

Scatter Plots for Threshold Approach Results

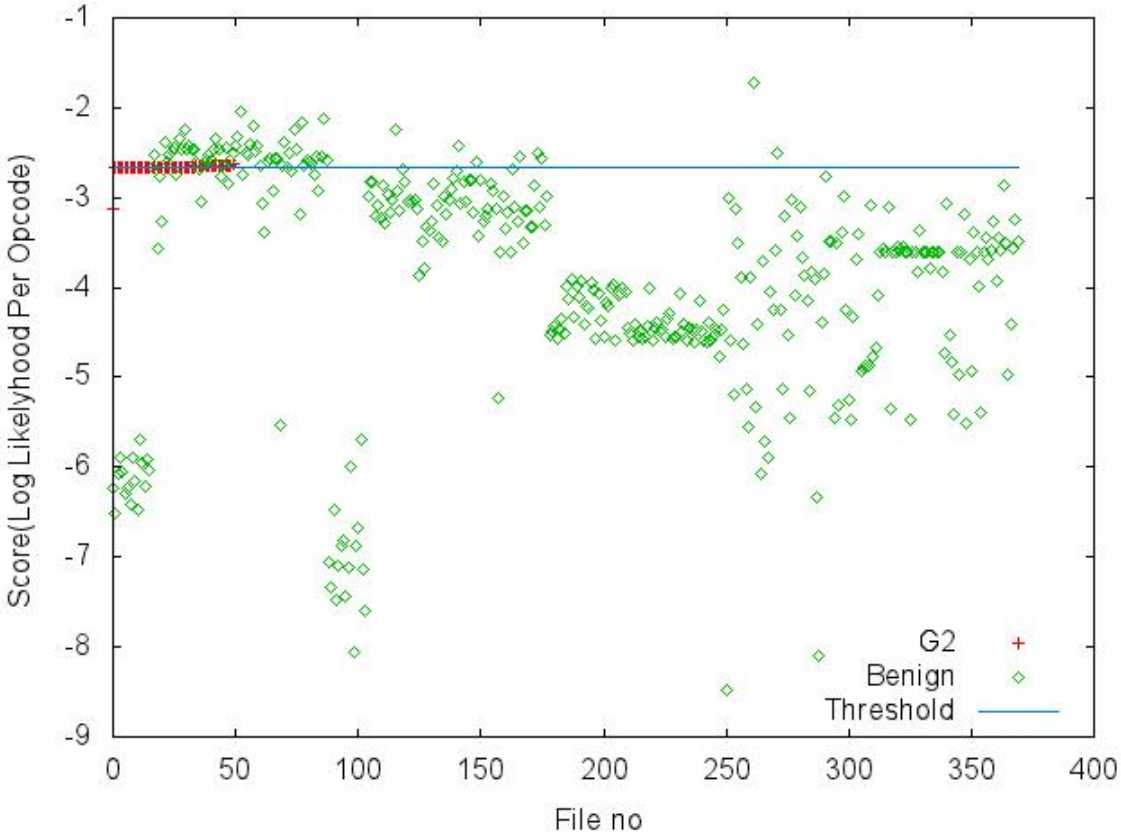

Figure A.1: G2 with threshold = -2.66659

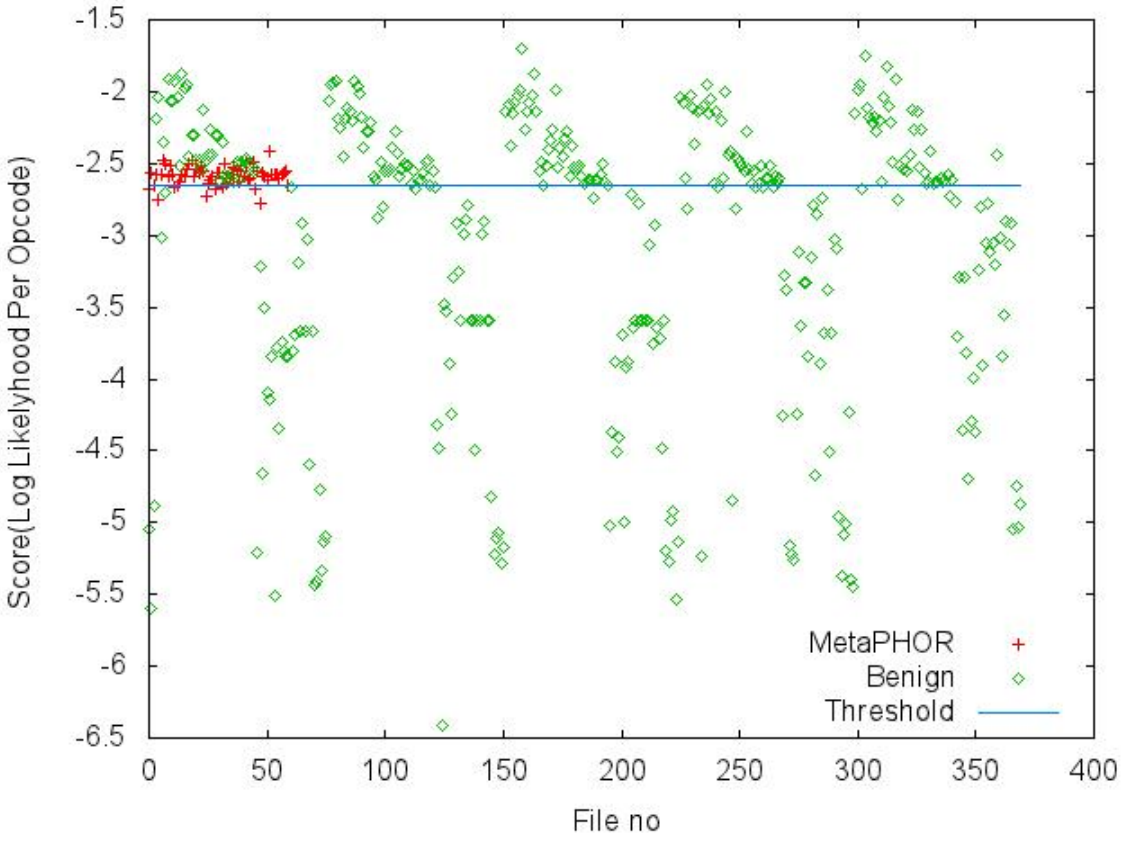

Figure A.2: MetaPHOR with threshold = -2.65207

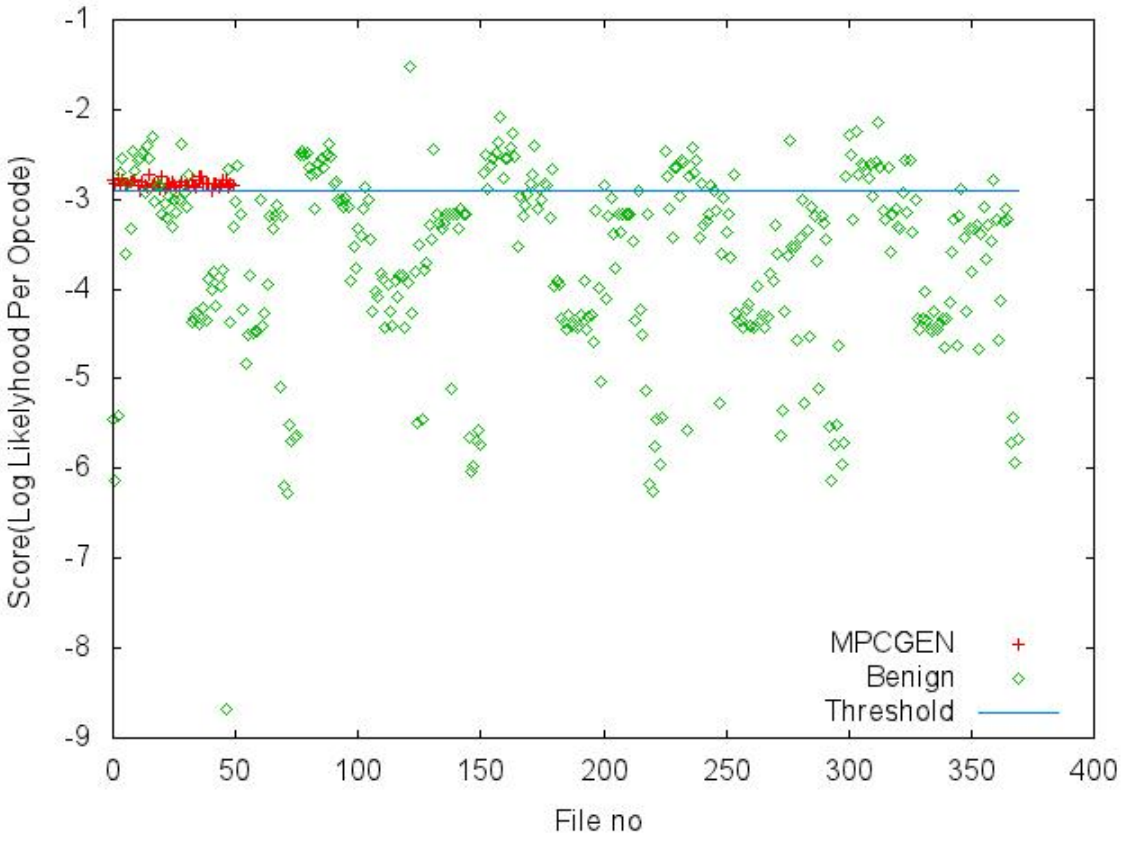

Figure A.3: MPCGEN with threshold = -2.90398

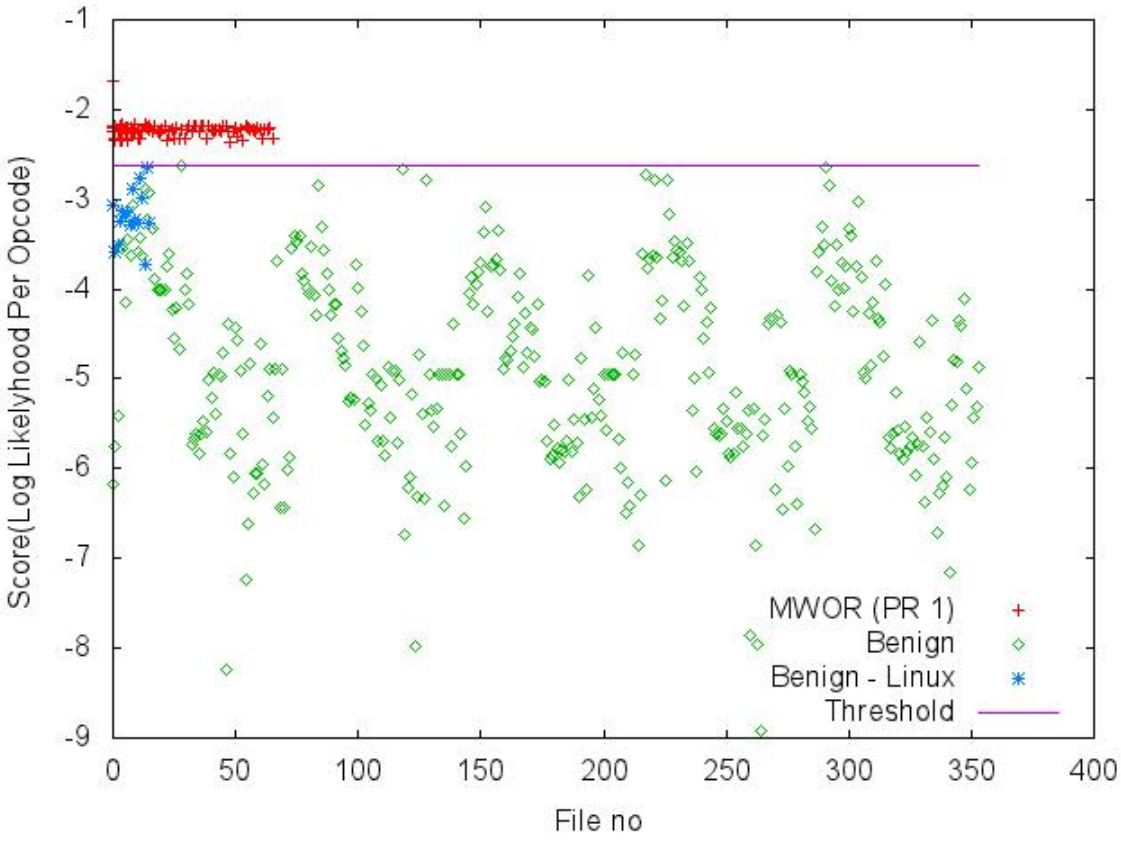

Figure A.4: MWOR (PR 1) with threshold = -2.6251

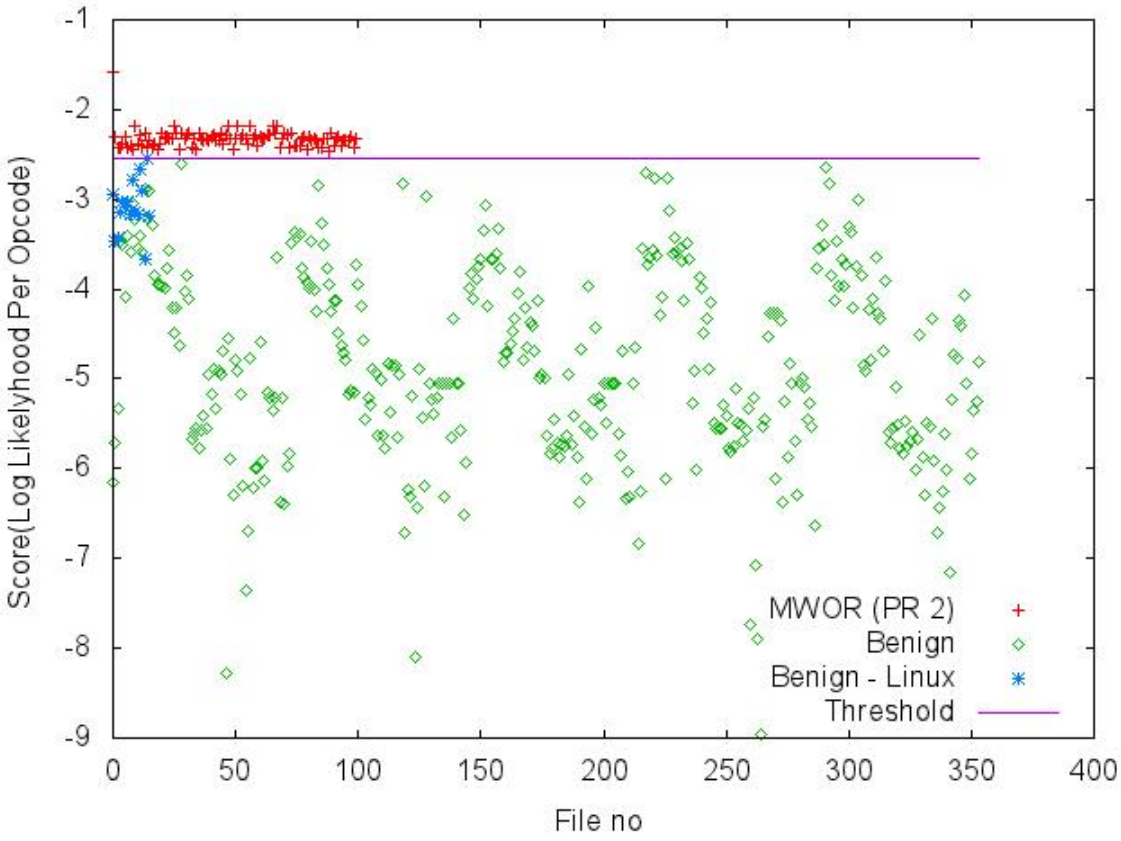

Figure A.5: MWOR (PR 2) with threshold = -2.5394

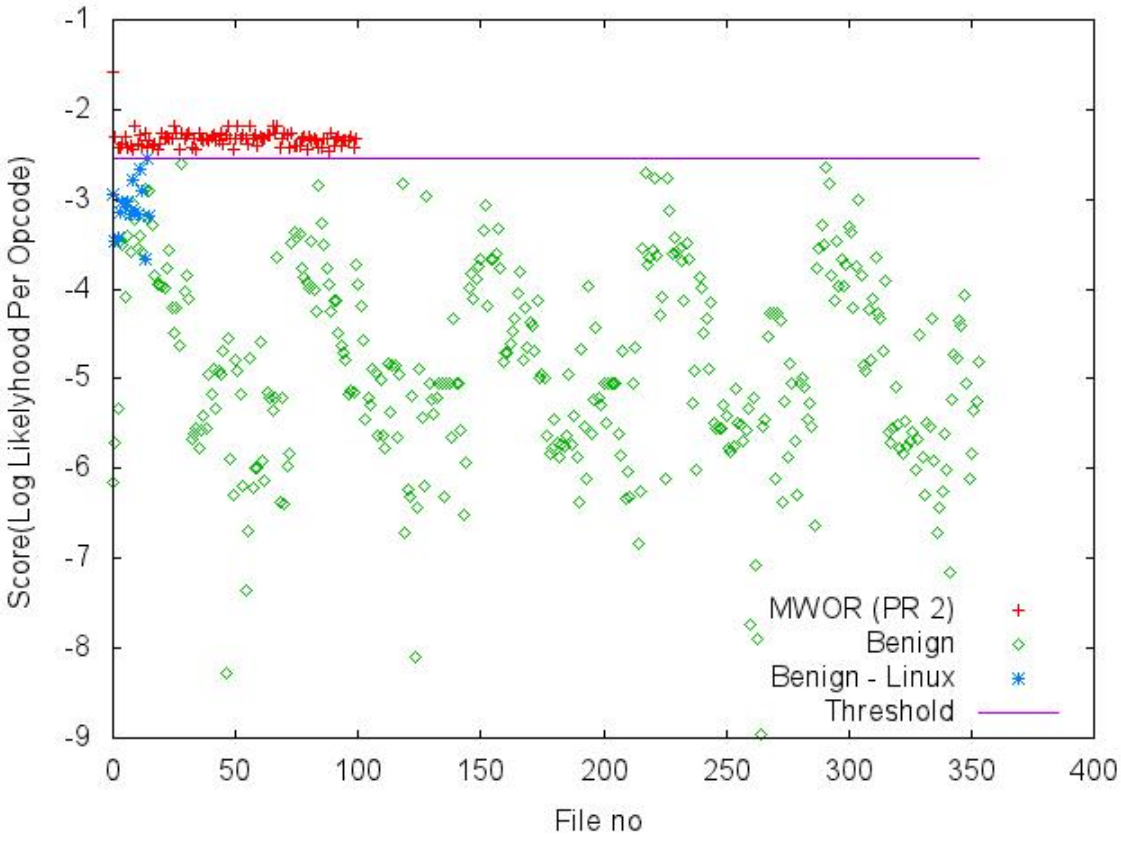

Figure A.6: MWOR (PR 3) with threshold = -2.63274

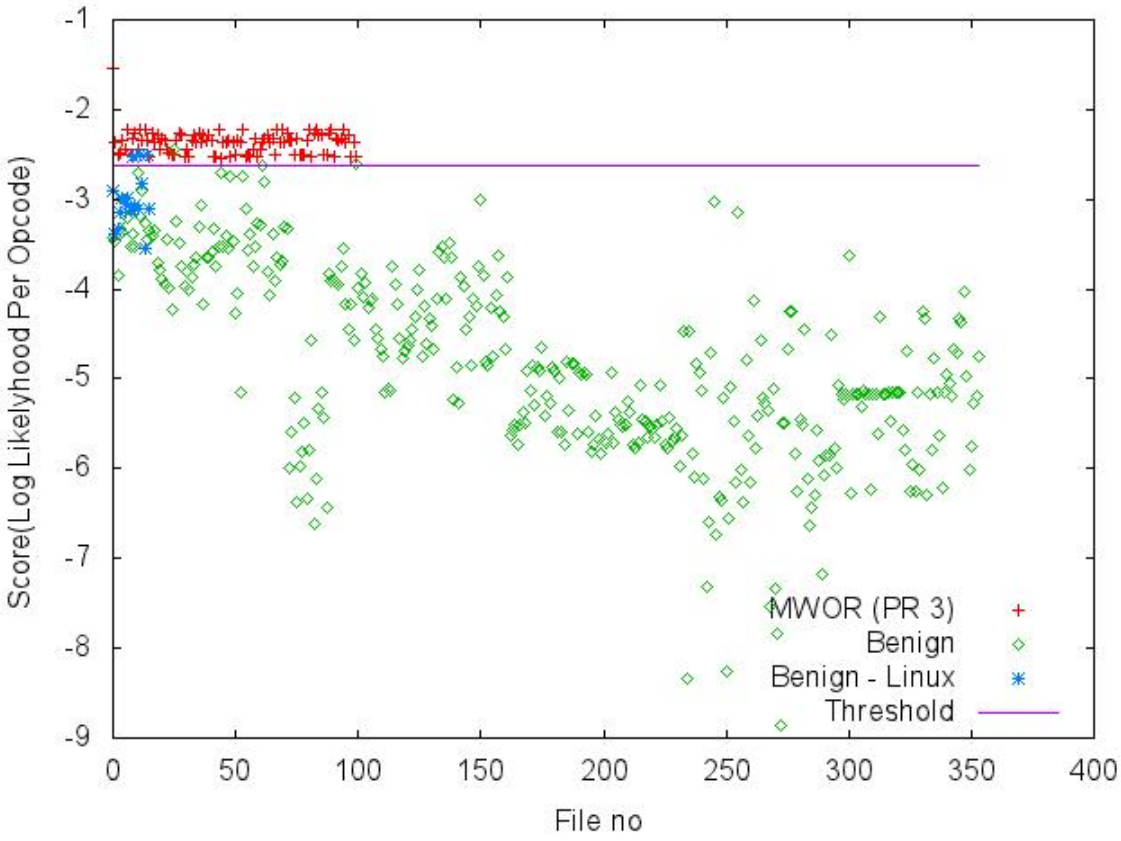

Figure A.7: MWOR (PR 4) with threshold = -2.60028

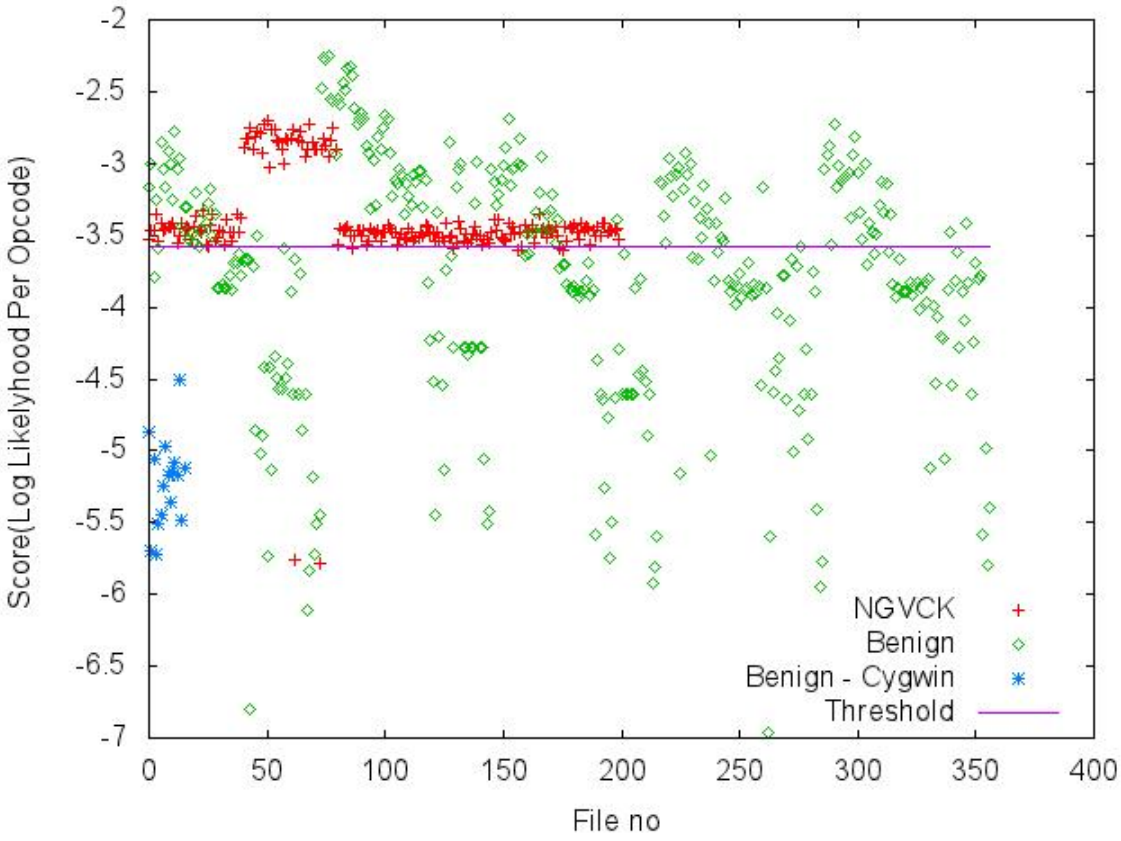

Figure A.8: NGVCK with threshold = -3.57961

APPENDIX B

Charts for the Dueling Approach

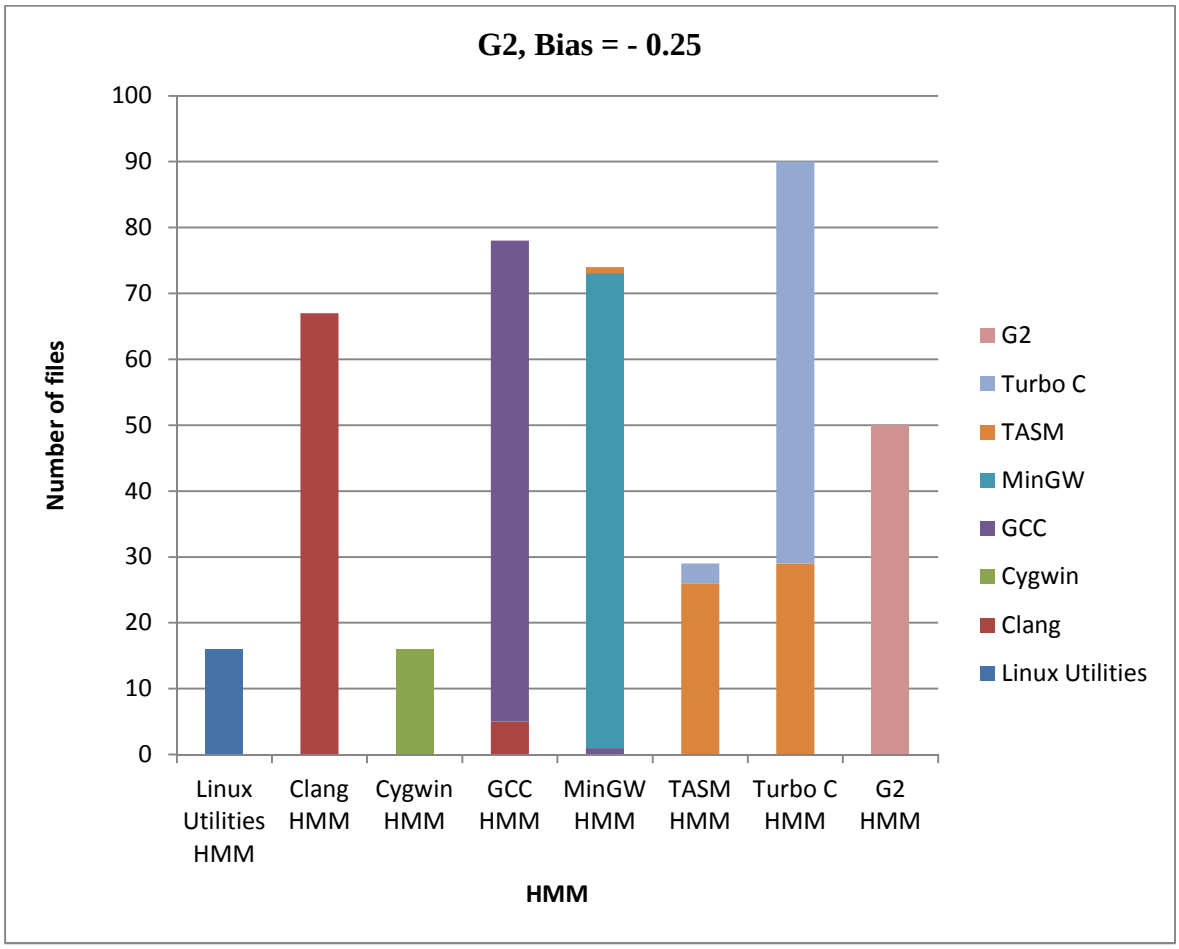

Figure B.1: G2 with bias = -0.25

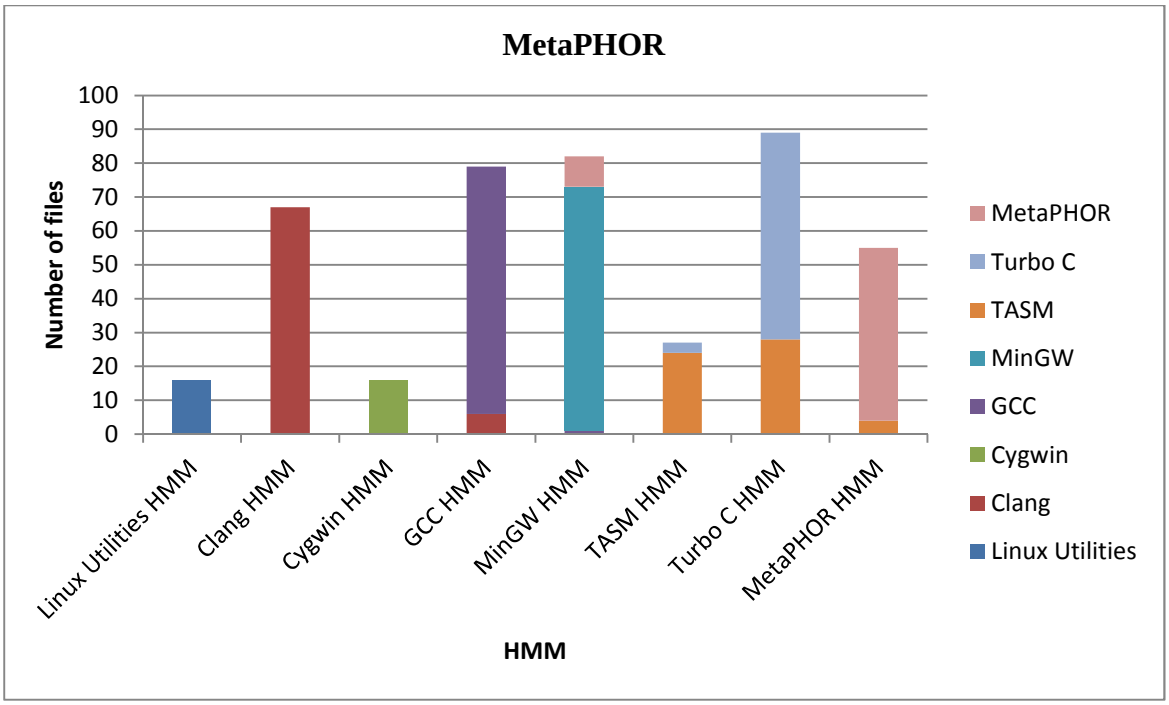

Figure B.2: MetaPHOR

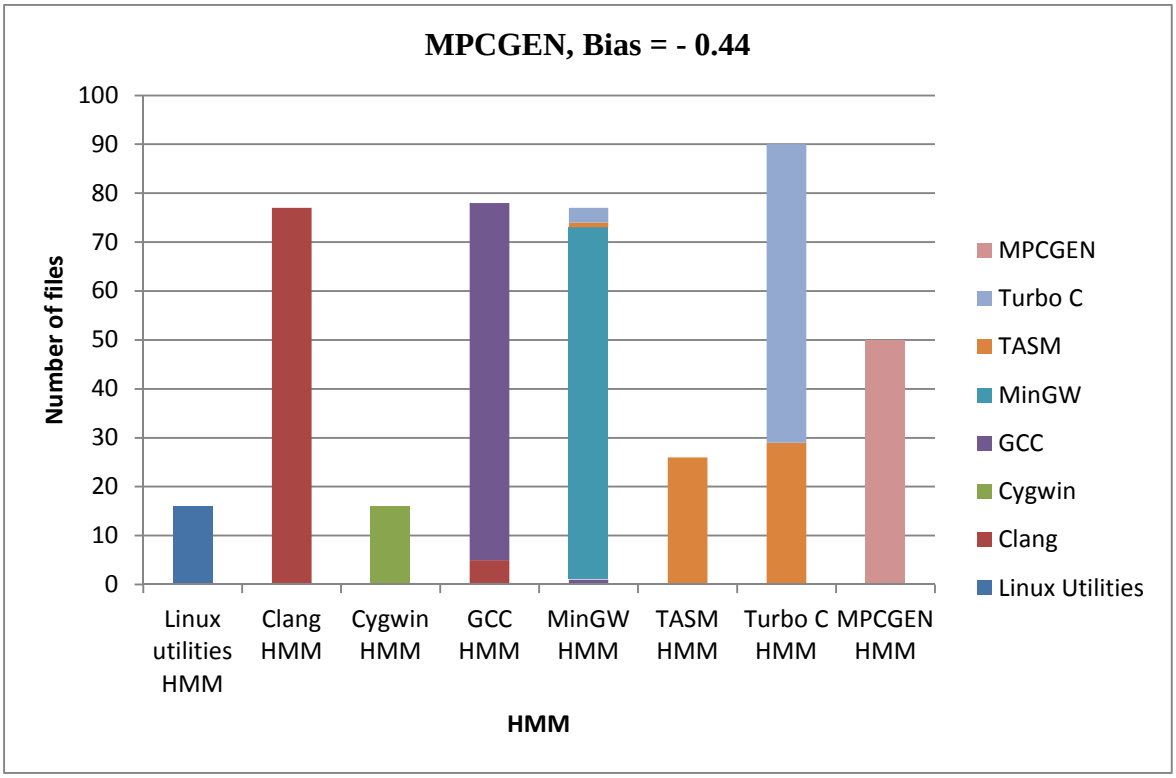

Figure B.3: MPCGEN with bias = -0.44

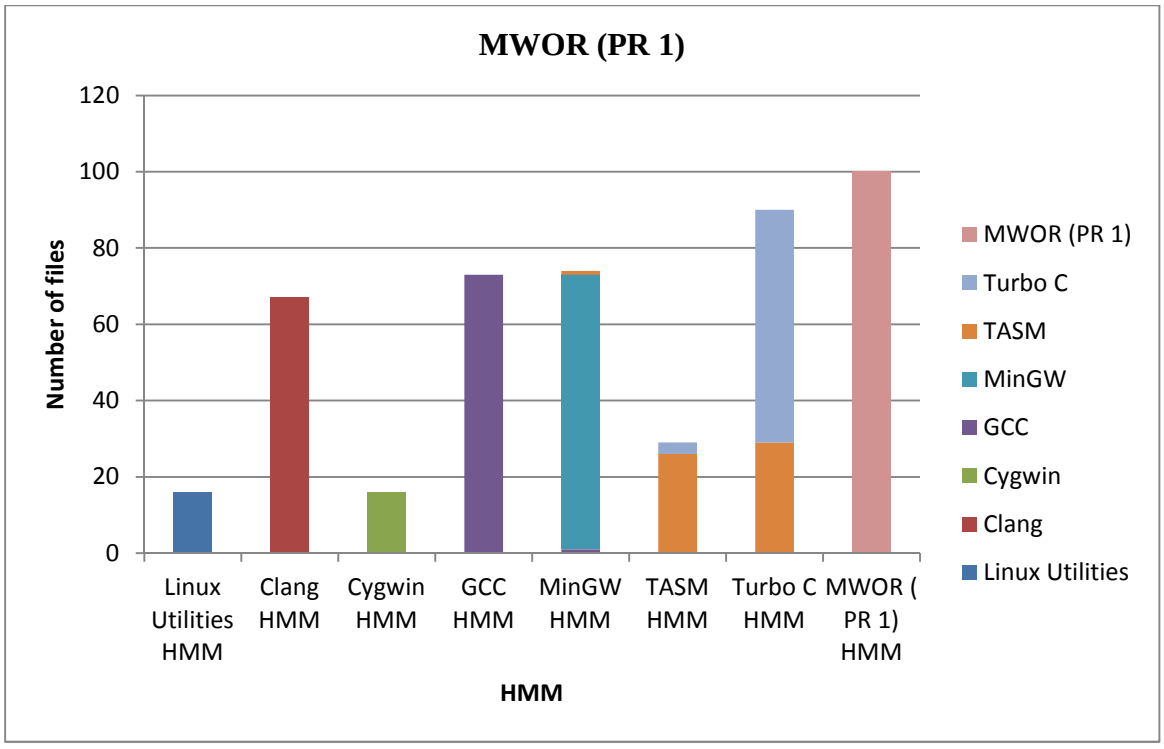

Figure B.4: MWOR (PR 1)

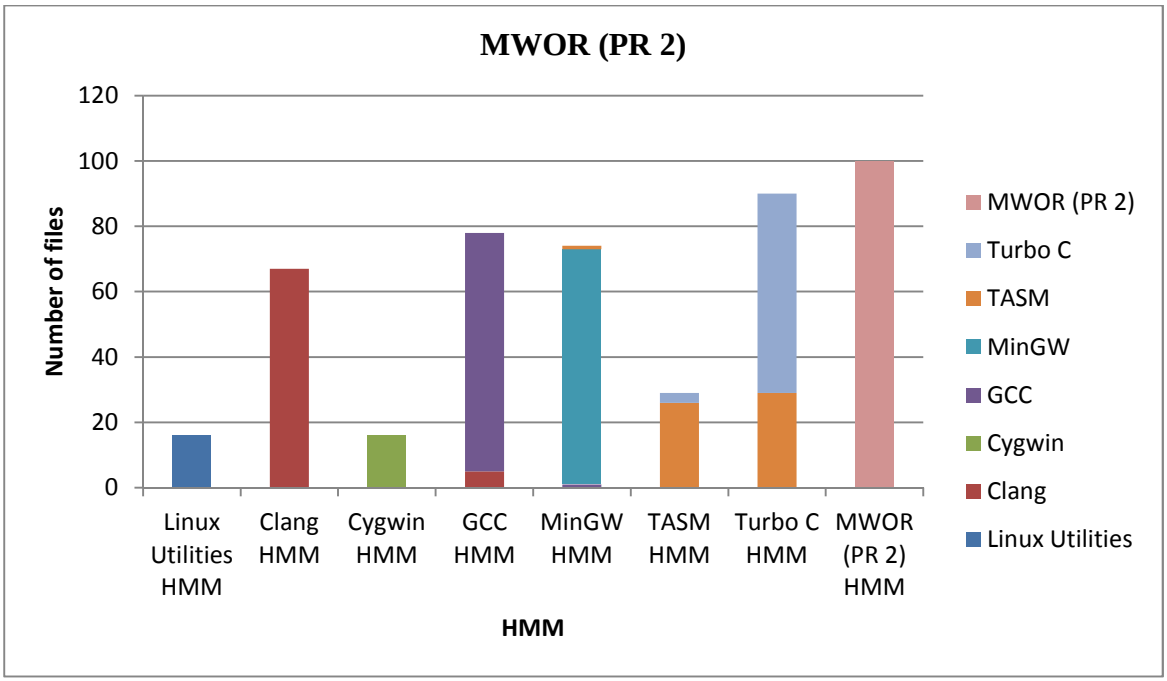

Figure B.5: MWOR (PR 2)

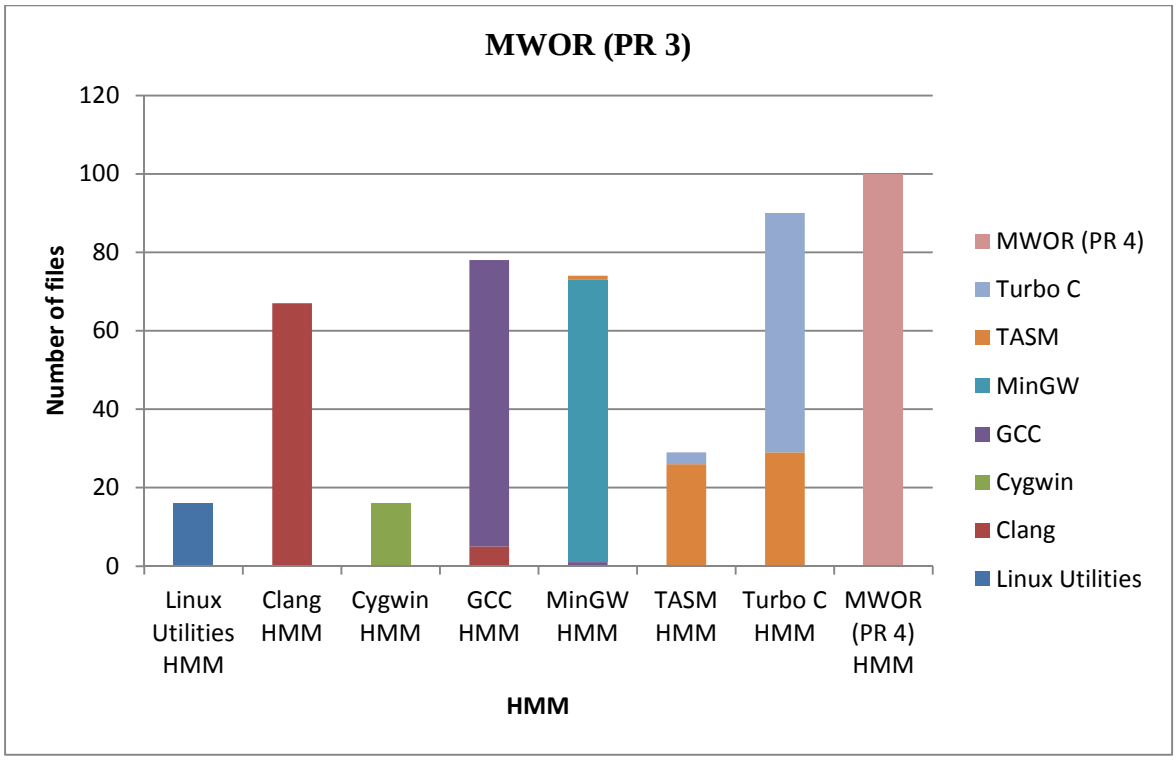

Figure B.6: MWOR (PR 3)

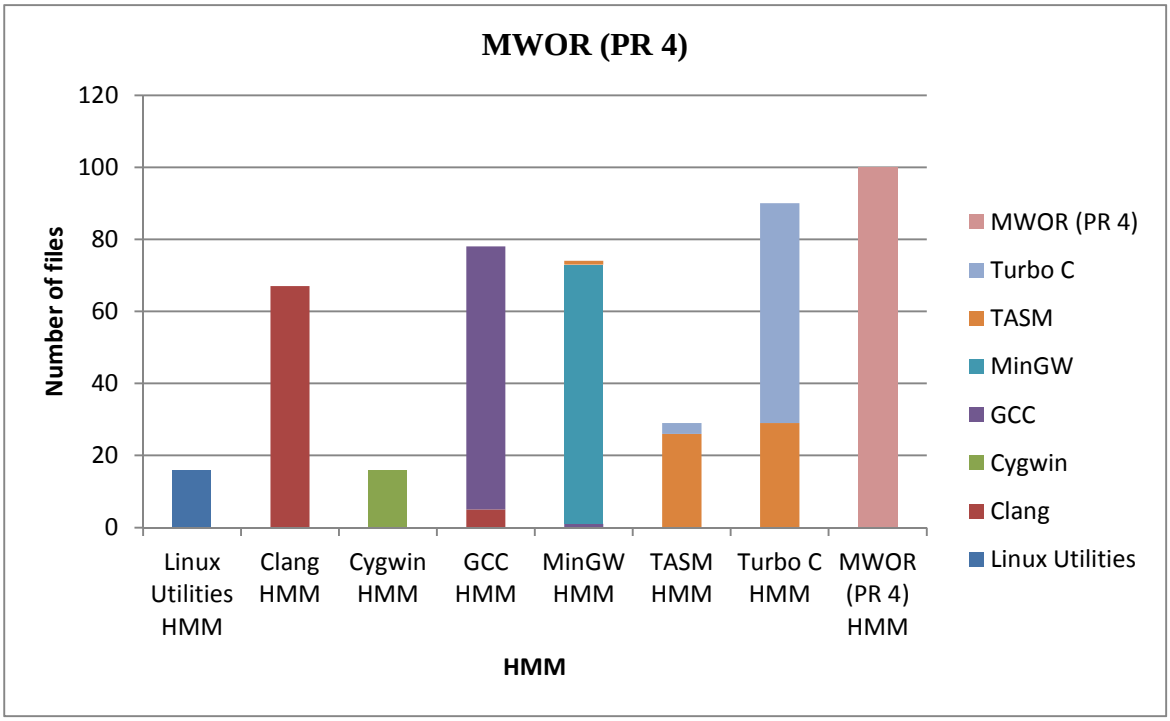

Figure B.7: MWOR (PR 4)

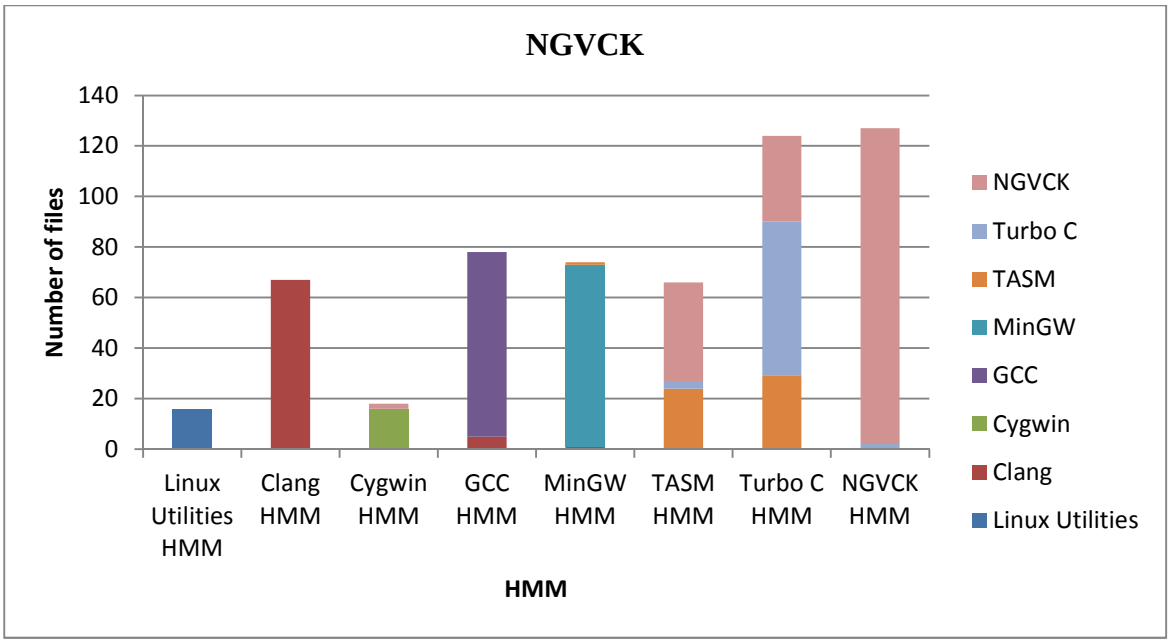

Figure B.8: NGVCK

APPENDIX C

Charts for Tiered Approach

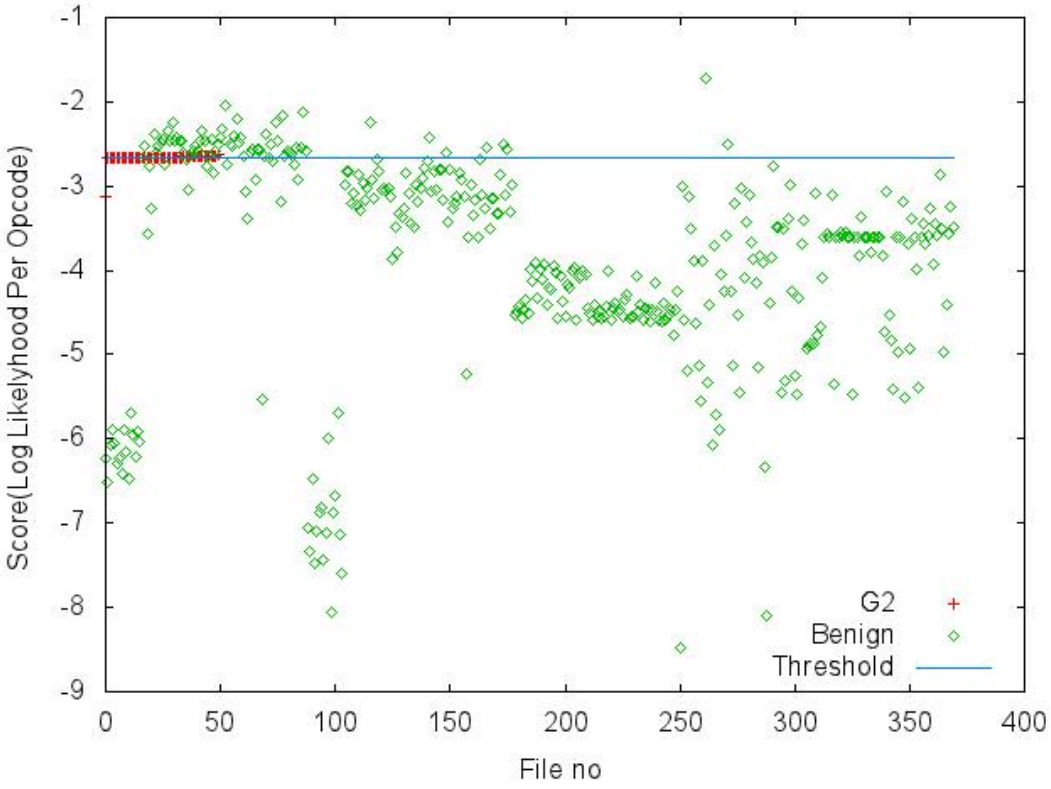

Figure C.1: G2, Tiered approach - Threshold Tier, Threshold = -3.2128274404

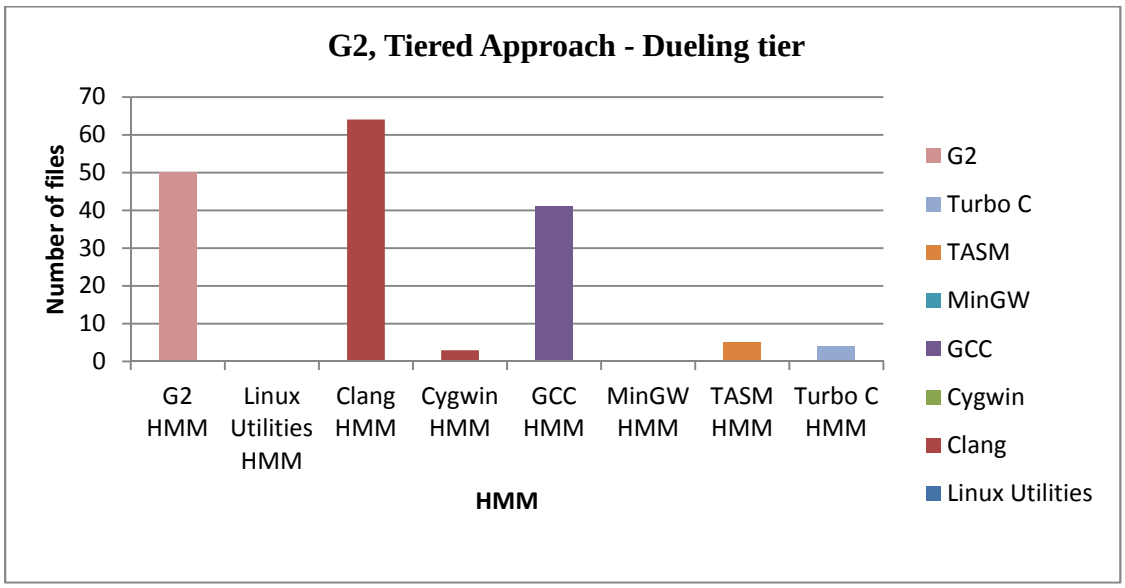

Figure C.2: G2, Tiered Approach – Dueling Tier, Bias =-0.25

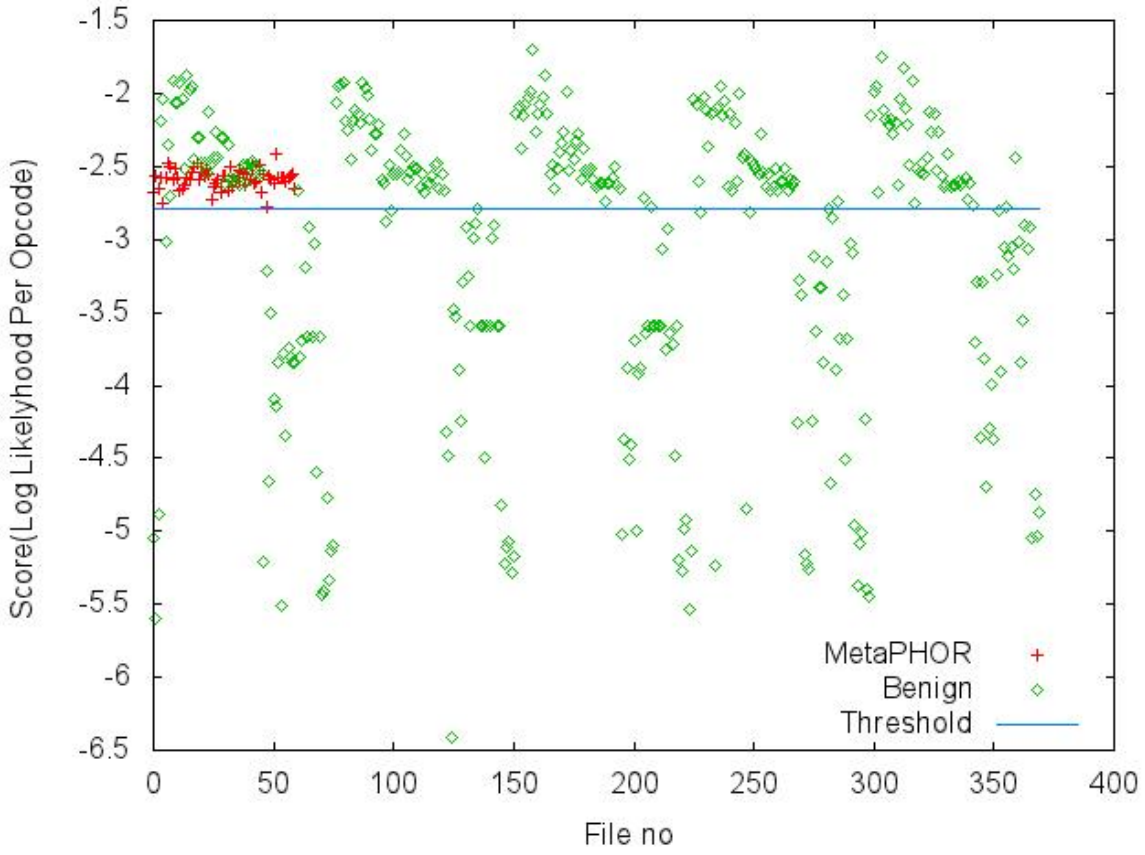

Figure C.3: MetaPHOR, Tiered Approach - Threshold tier, Threshold = -2.791860013

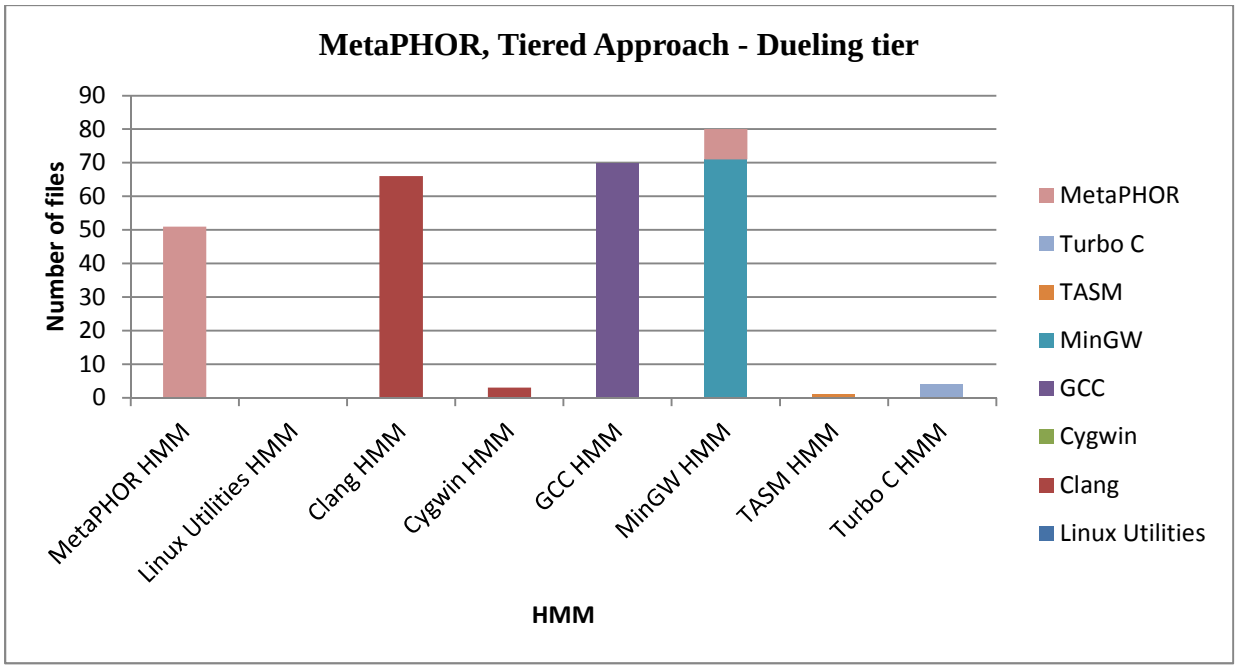

Figure C.4: MetaPHOR, Tiered Approach – Dueling tier

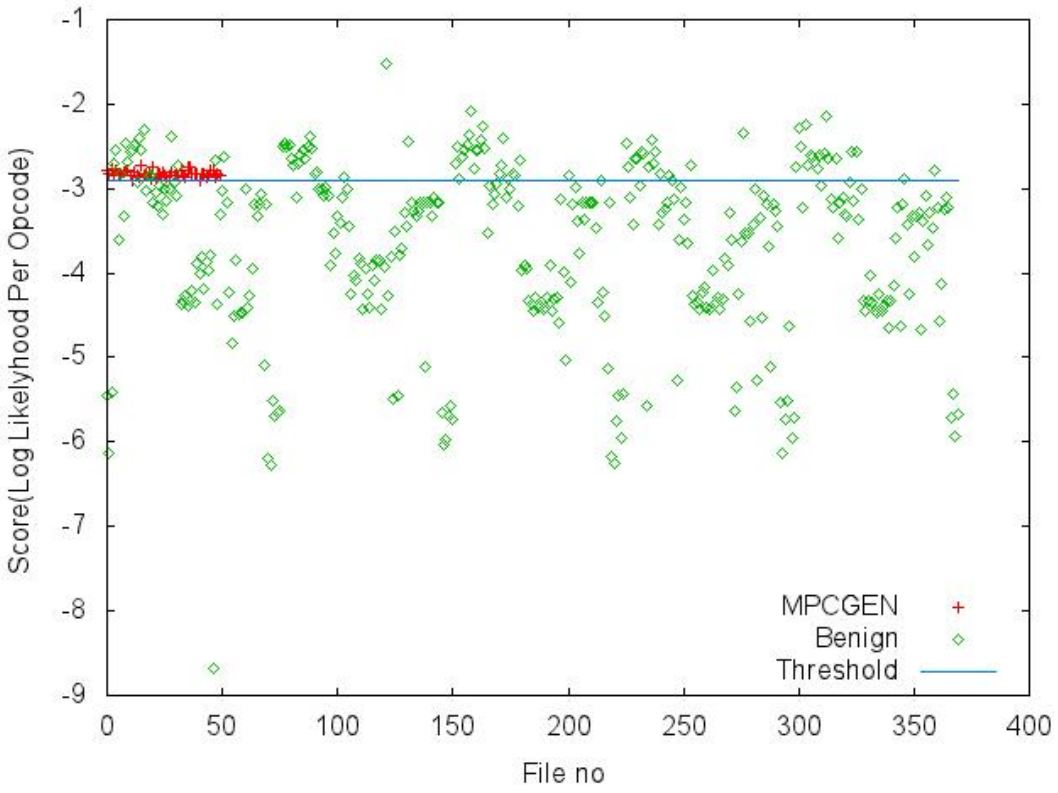

Figure C.5: MPCGEN, Tiered Approach, Threshold tier, Threshold = -2.903975532

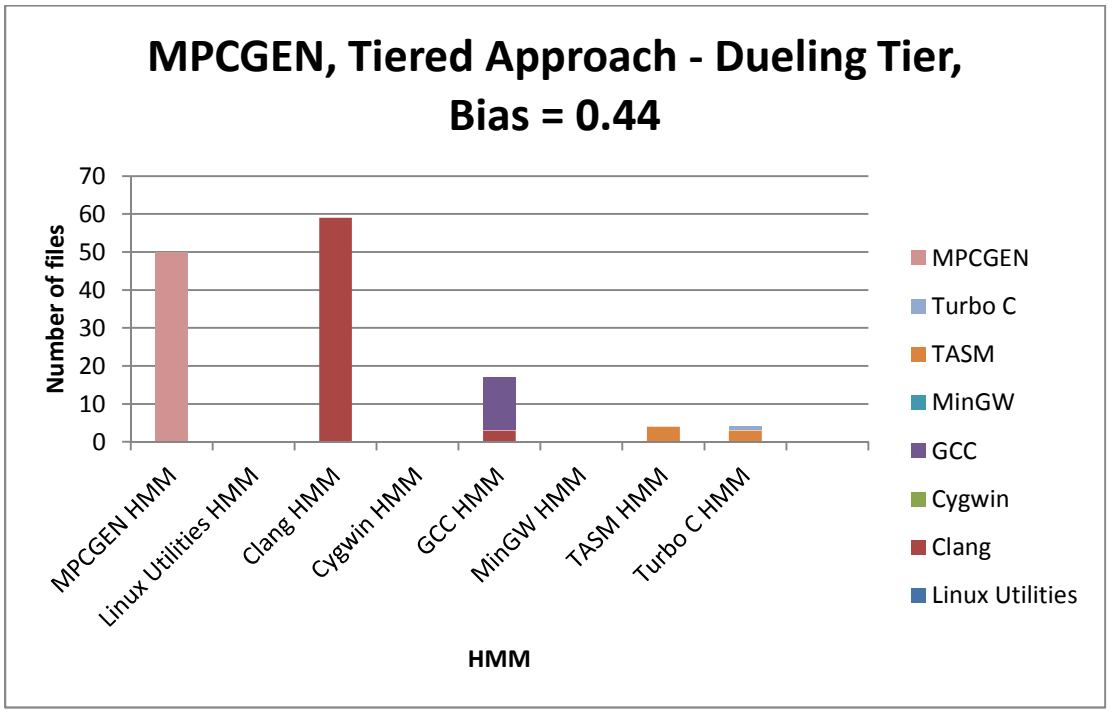

Figure C.6: MPCGEN, Tiered approach - Dueling Tier, Bias = - 0.44

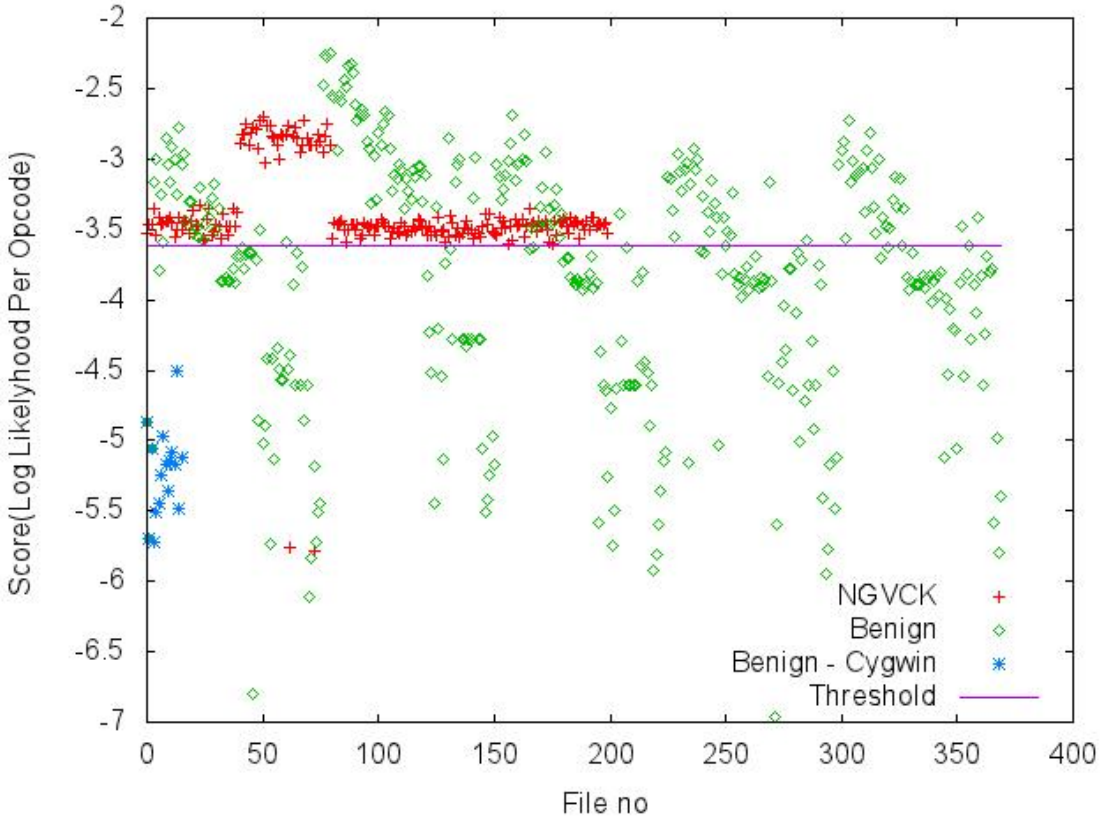

Figure C.7: NGVCK, Tiered Approach - Threshold Tier, Threshold = -3.620342489

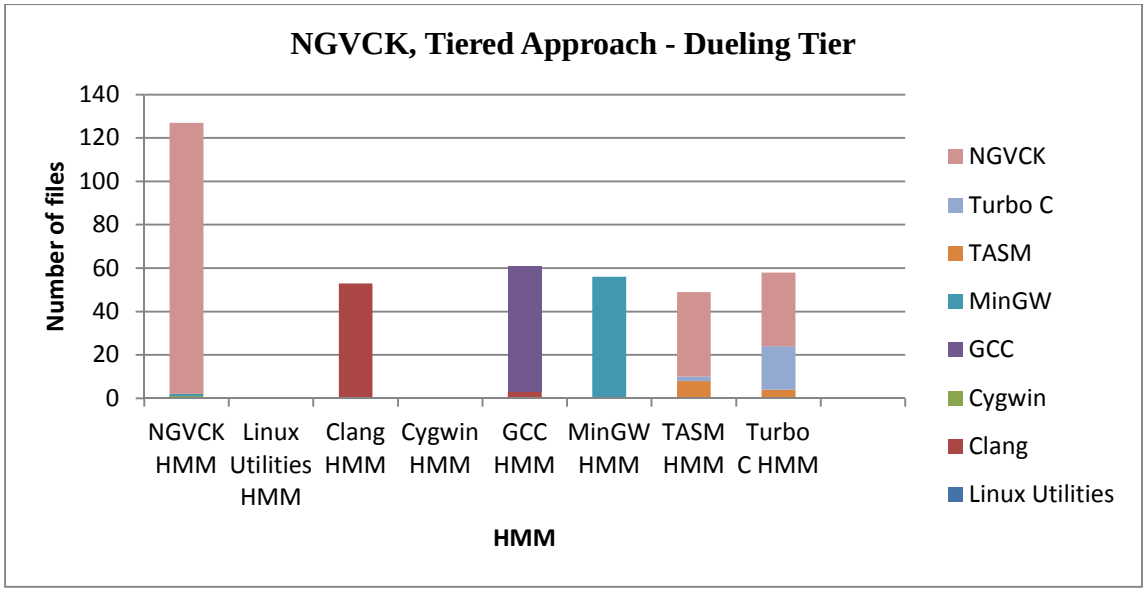

Figure C.8 NGVCK, Tiered Approach - Dueling Tier

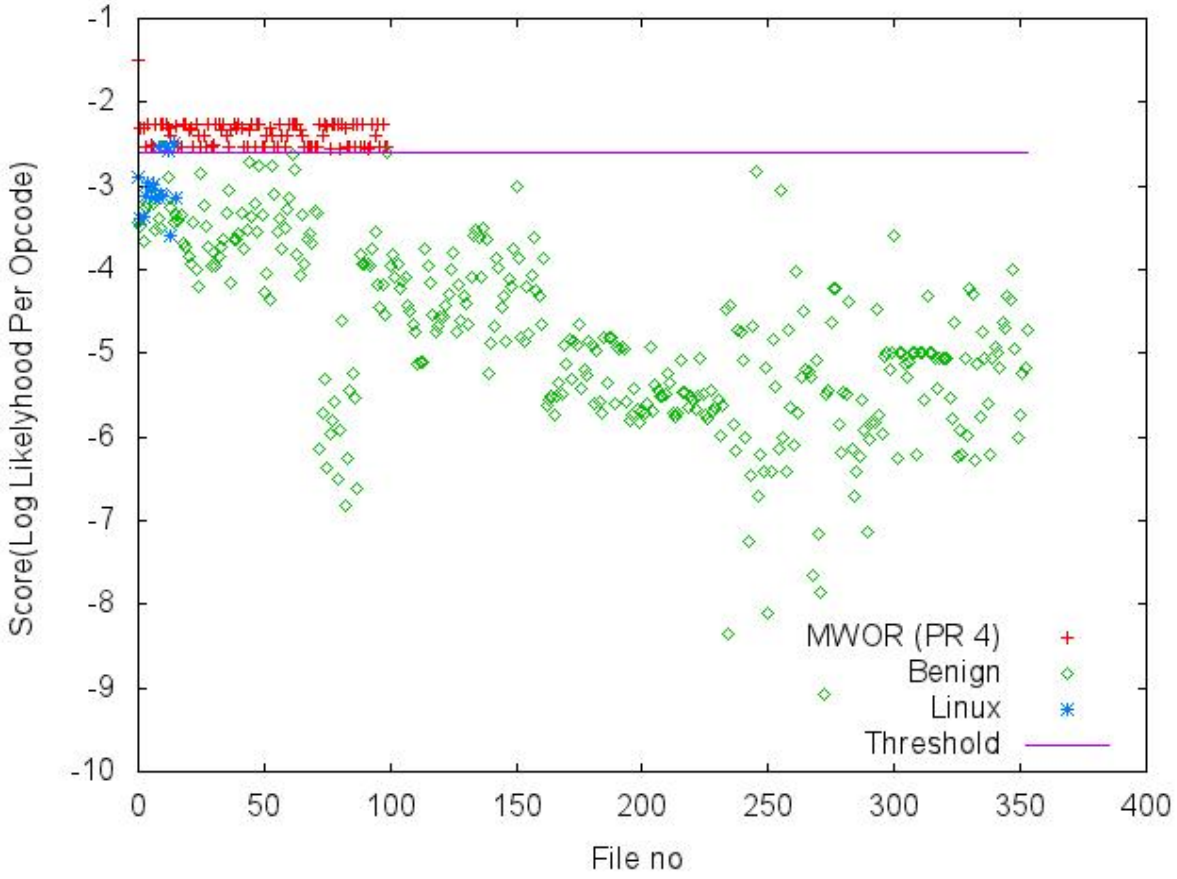

Figure C.9: MWOR (PR 4), Tiered Approach - Threshold Tier, Threshold = -2.539395038

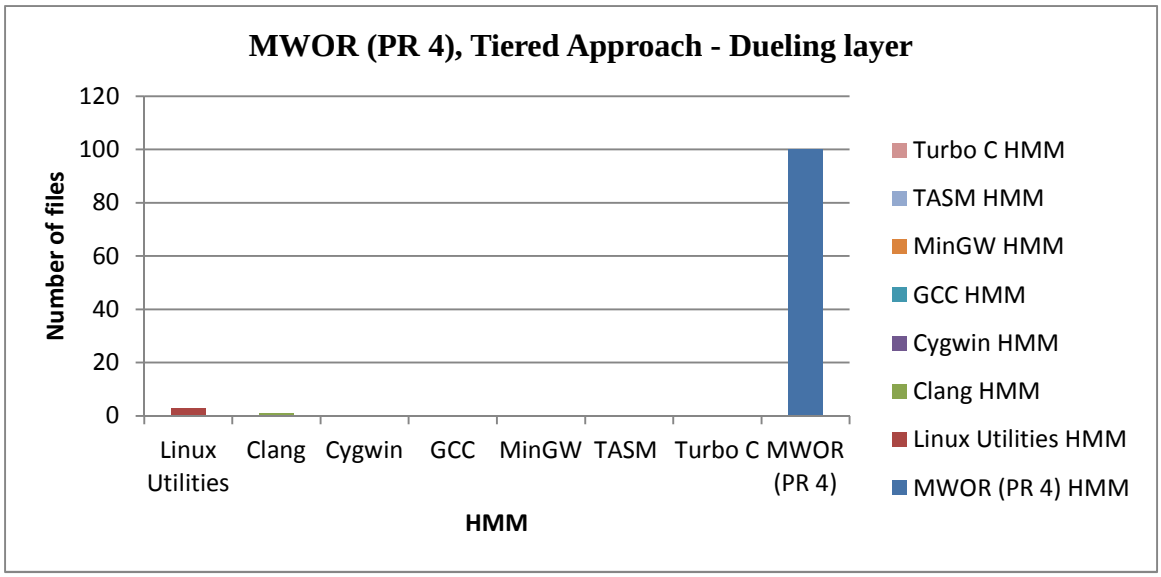

Figure C.10: MWOR (PR 4), Tiered Approach - Dueling Tier

APPENDIX D

K-Means Clustering Results

A. Centroids = 3, Silhouette = 0.625796768

Table D.1: 3 Centroids, Benign - Malware Distribution by Clusters

Cluster -> Type	0	1	2
Benign	3	485	103
Malware	396	15	309

Table D.2: 3 Centroids, Detailed distribution by clusters

	Zero	One	Two	Three
Linux_Utilities	3	0	0	0
Clang	0	266	0	0
GCC	0	192	1	0
MinGW	0	14	73	0
TASM	0	5	18	0
Turbo C	0	8	9	0
G2	0	0	50	0
MetaPHOR	0	11	49	0
MPCGEN	0	0	50	0
MWOR (PR 1)	99	1	0	0
MWOR (PR 2)	99	1	0	0
MWOR (PR 3)	99	1	0	0
MWOR (PR 4)	99	1	0	0
NGVCK	0	0	162	0

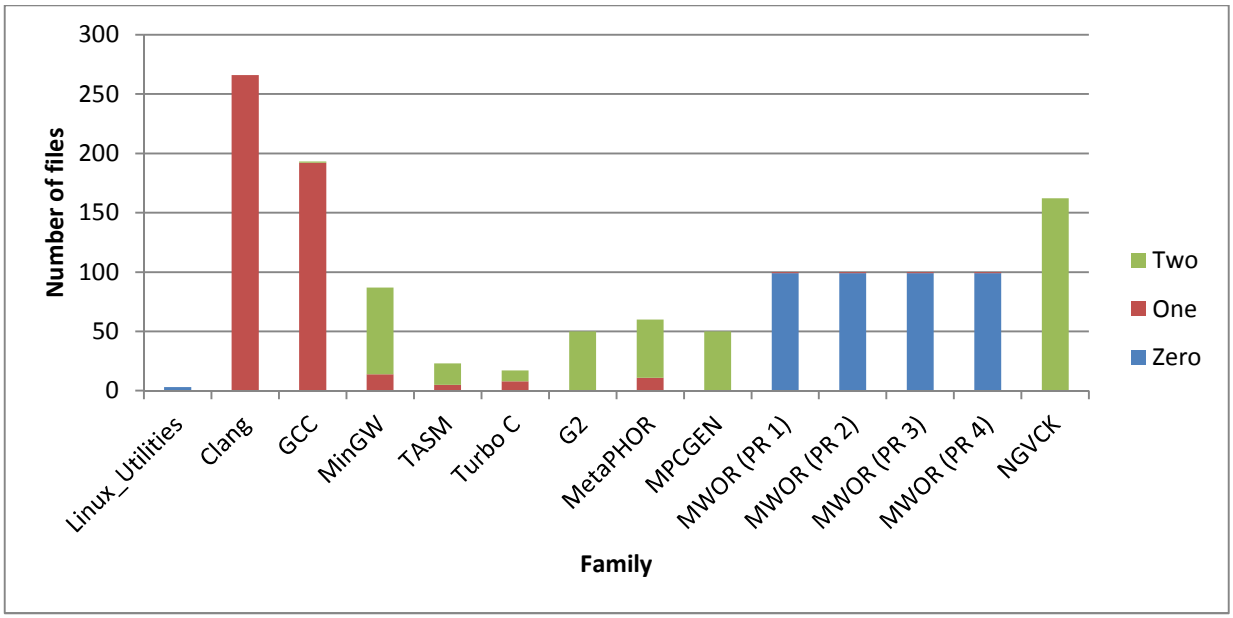

Figure D.1: 3 Centroids, Family classification

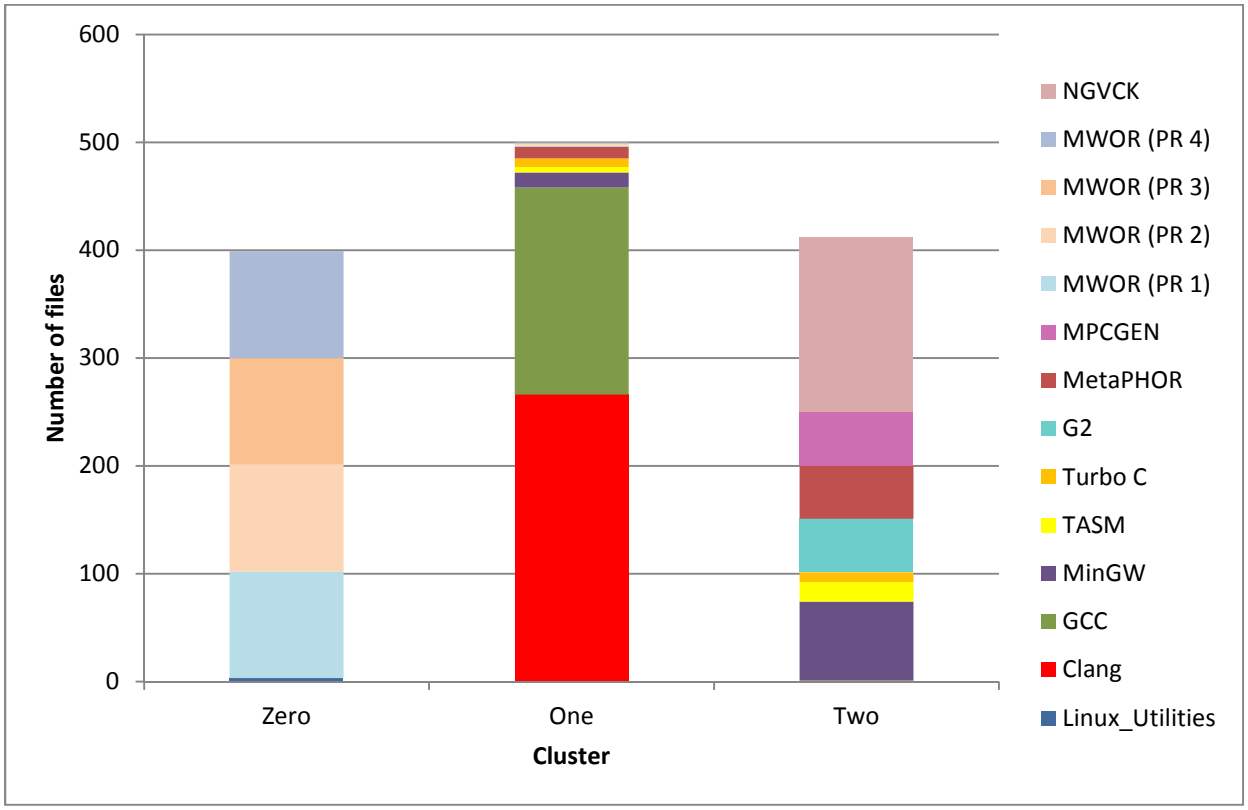

Figure D.2: 3 Centroids, Cluster Composition

B. Centroids = 4, Silhouette = 0.599853077

Table D.3: 4 Centroids, Benign - Malware Distribution by Clusters

Cluster -> Type	0	1	2	3
Benign	145	3	428	15
Malware	60	396	4	260

Table D.4: 4 Centroids, Detailed distribution by clusters

	Zero	One	Two	Three
Linux_Utilities	0	3	0	0
Clang	0	0	266	0
GCC	38	0	155	0
MinGW	87	0	0	0
TASM	8	0	3	12
Turbo C	12	0	4	1
G2	0	0	0	50
MetaPHOR	60	0	0	0
MPCGEN	0	0	0	50
MWOR (PR 1)	0	99	1	0
MWOR (PR 2)	0	99	1	0
MWOR (PR 3)	0	99	1	0
MWOR (PR 4)	0	99	1	0
NGVCK	0	0	0	162

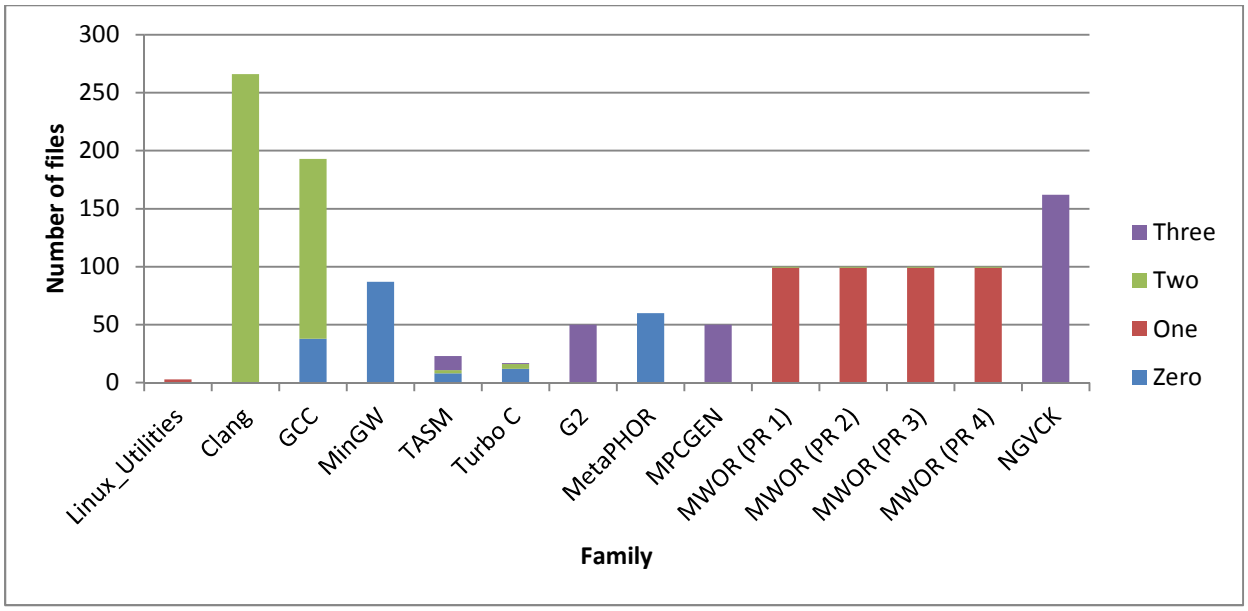

Figure D.3: 4 Centroids, Family classification

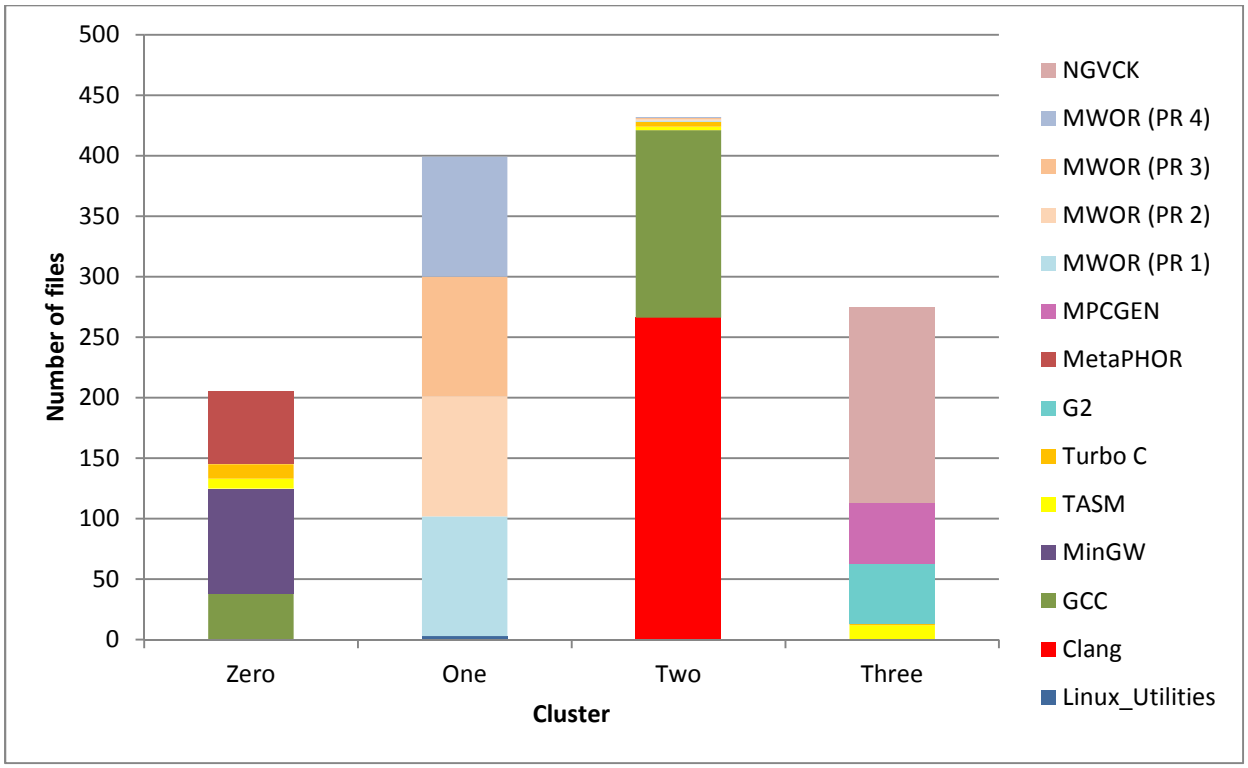

Figure D.4: 4 Centroids, Cluster Composition

C. Centroids = 7, Silhouette = 0.553054408

Table D.5: 7 Centroids, Benign - Malware Distribution by Clusters

Cluster -> Type	0	1	2	3	4	5	6
Benign	5	111	3	234	0	228	10
Malware	162	60	396	4	51	0	47

Table D.6: 7 Centroids, Detailed distribution by clusters

	Zero	One	Two	Three	Four	Five	Six
Linux_Utilities	0	0	3	0	0	0	0
Clang	0	0	0	205	0	61	0
GCC	0	5	0	26	0	162	0
MinGW	0	87	0	0	0	0	0
TASM	4	8	0	3	0	0	8
Turbo C	1	11	0	0	0	5	0
G2	0	0	0	0	1	0	49
MetaPHOR	0	60	0	0	0	0	0
MPCGEN	0	0	0	0	50	0	0
MWOR (PR 1)	0	0	99	1	0	0	0
MWOR (PR 2)	0	0	99	1	0	0	0
MWOR (PR 3)	0	0	99	1	0	0	0
MWOR (PR 4)	0	0	99	1	0	0	0
NGVCK	162	0	0	0	0	0	0

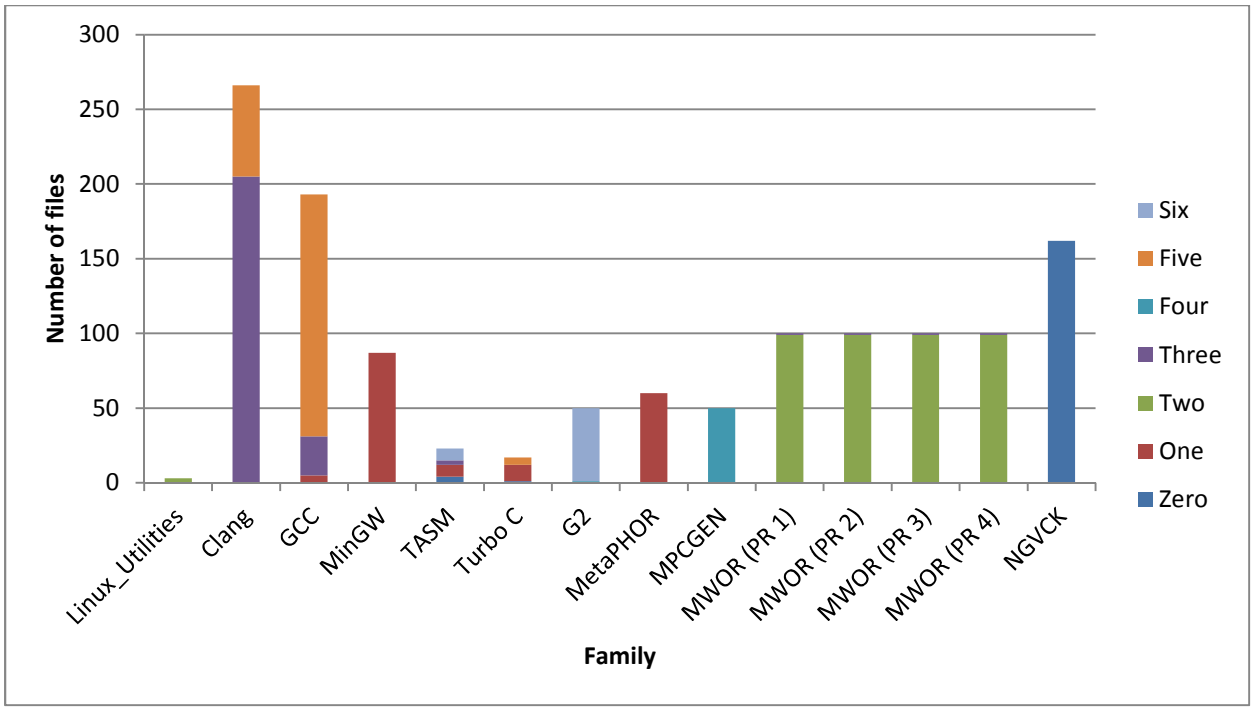

Figure D.5: 7 Centroids, Family classification

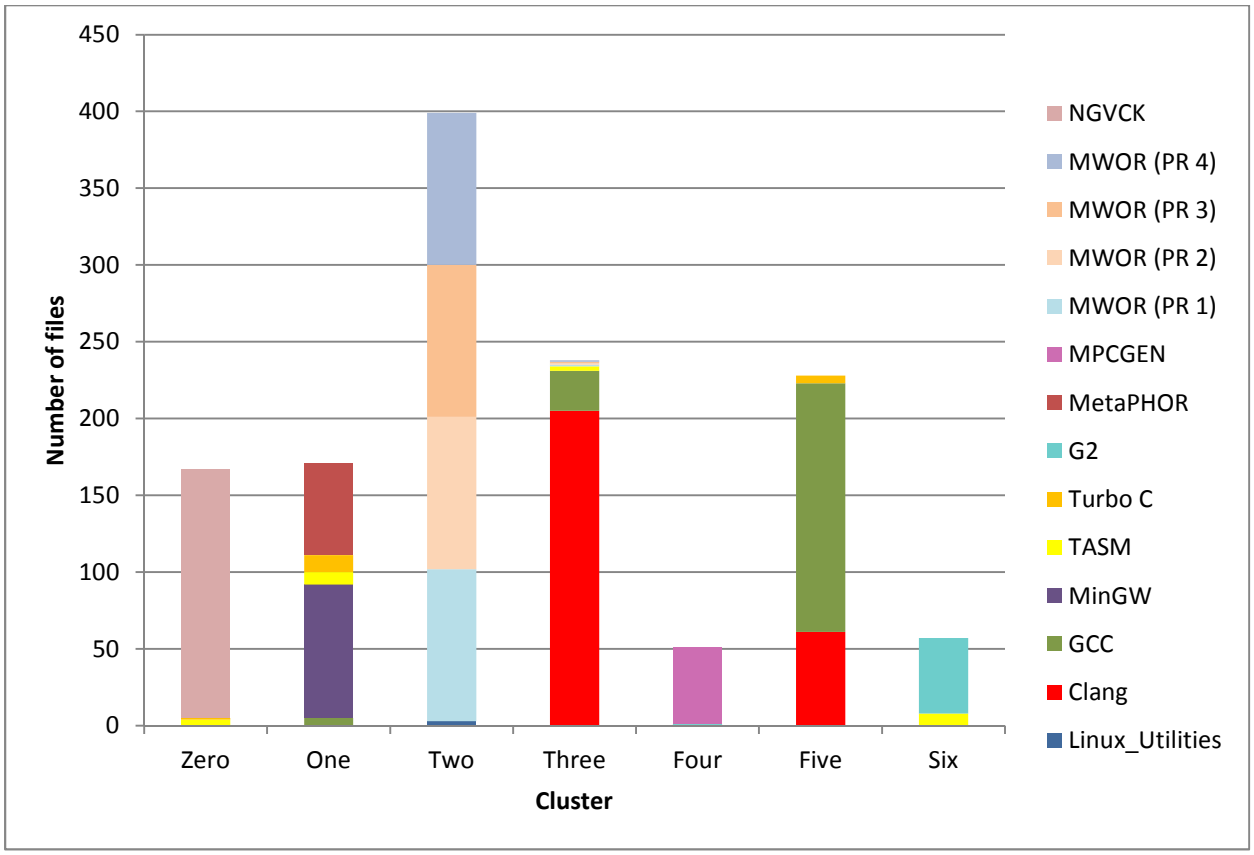

Figure D.6: 7 Centroids, Cluster Composition

D. Centroids = 6, Silhouette = 0.547637992

Table D.7: 6 Centroids, Benign - Malware Distribution by Clusters

Cluster -> Type	0	1	2	3	4	5
Benign	234	3	11	111	228	4
Malware	4	396	162	60	0	98

Table D.8: 6 Centroids, Detailed distribution by clusters

	Zero	One	Two	Three	Four	Five
Linux_Utilities	0	3	0	0	0	0
Clang	205	0	0	0	61	0
GCC	26	0	0	5	162	0
MinGW	0	0	0	87	0	0
TASM	3	0	10	8	0	2
Turbo C	0	0	1	11	5	0
G2	0	0	0	0	0	50
MetaPHOR	0	0	0	60	0	0
MPCGEN	0	0	0	0	0	50
MWOR (PR 1)	1	99	0	0	0	0
MWOR (PR 2)	1	99	0	0	0	0
MWOR (PR 3)	1	99	0	0	0	0
MWOR (PR 4)	1	99	0	0	0	0
NGVCK	0	0	162	0	0	0

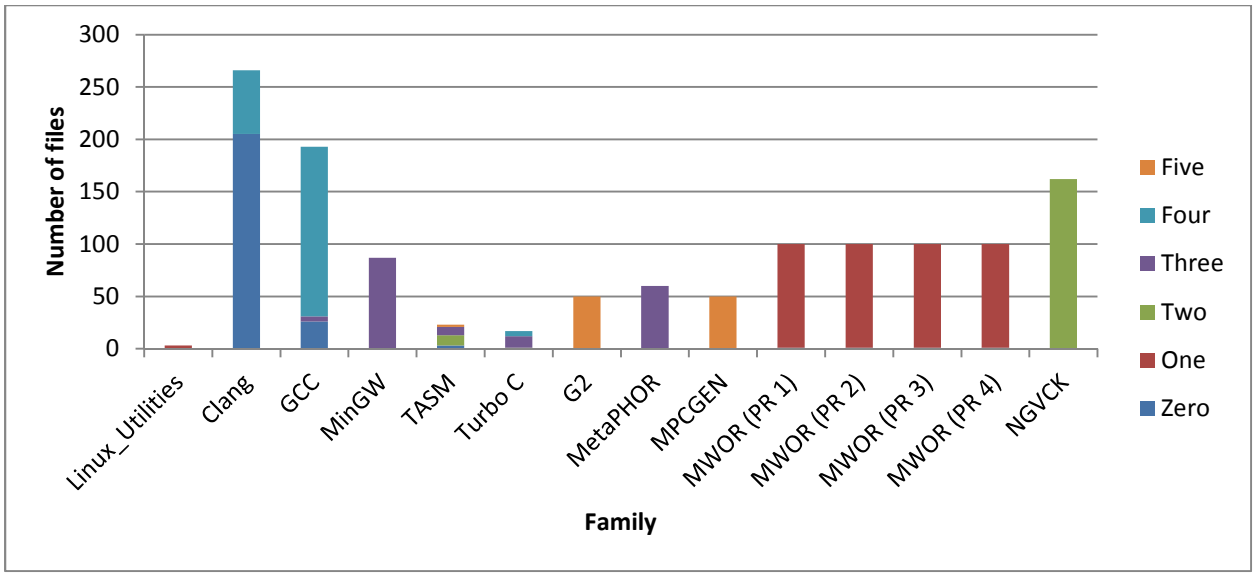

Figure D.7: 6 Centroids, Family classification

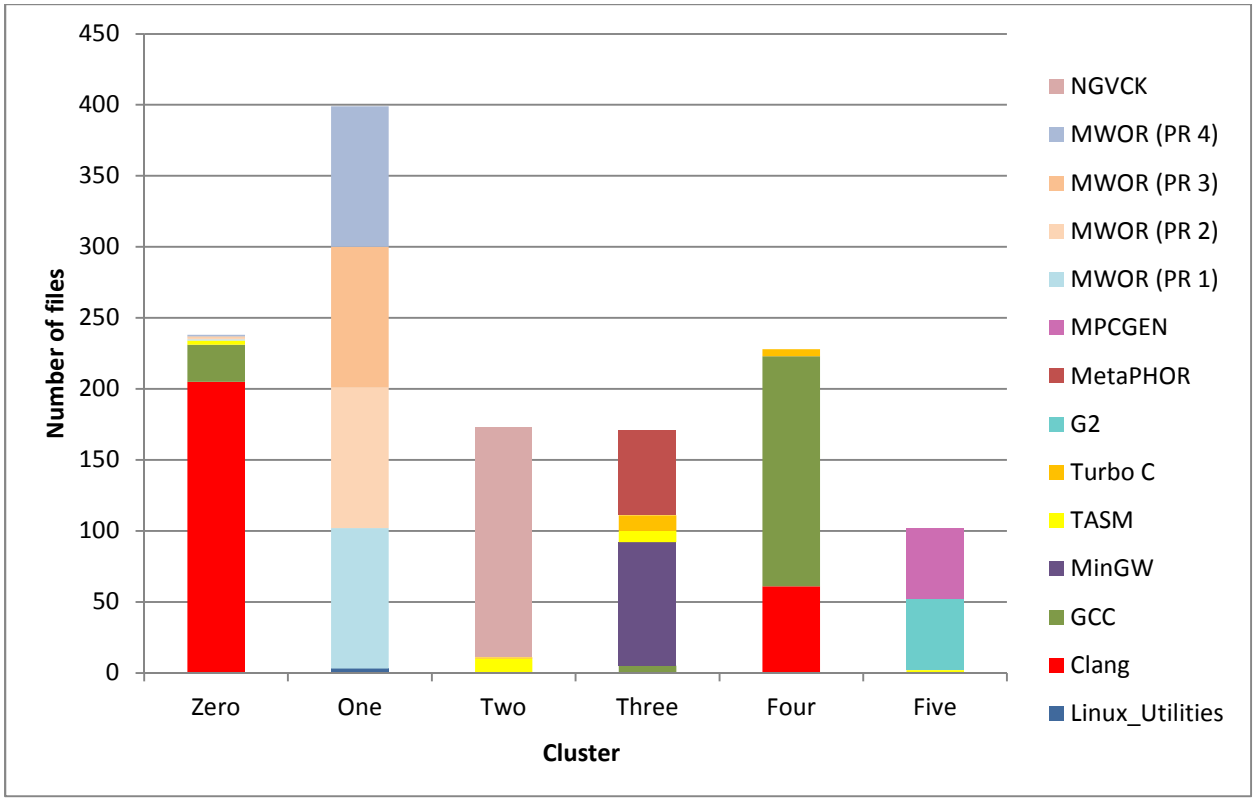

Figure D.8: 6 Centroids, Cluster Composition