CODE OBFUSCATION AND VIRUS DETECTION

A Writing 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
Ashwini Venkatesan

May, 2008

ACKNOWLEDGEMENTS

I am greatly indebted to Dr Mark Stamp not only for his expert guidance, judgment, suggestions and insight without which this thesis could not have been completed but also for his excellent information security class which sparked my interest in the subject and started me on this journey.

I would also like to thank Dr Agustin Arraya and Dr Soon Tee Teoh for graciously consenting to be on my committee and providing me with valuable feedback.

Thanks are also due to all my friends and lab mates for their help and companionship which made graduate school a much more memorable experience.

And to my husband – thank you for putting up with all the long nights and weekends and making my life pleasurable when I was busy and down!

ABSTRACT

Typically, computer viruses and other malware are detected by searching for a string of bits which is found in the virus or malware. Such a string can be viewed as a "fingerprint" of the virus. These "fingerprints" are not generally unique; however they can be used to make rapid malware scanning feasible. This fingerprint is often called a signature and the technique of detecting viruses using signatures is known as signature-based detection [8].

Today, virus writers often camouflage their viruses by using code obfuscation techniques in an effort to defeat signature-based detection schemes. So-called metamorphic viruses are viruses in which each instance has the same functionality but differs in its internal structure. Metamorphic viruses differ from polymorphic viruses in the method they use to hide their signature. While polymorphic viruses primarily rely on encryption for signature obfuscation, metamorphic viruses hide their signature via "mutating" their own code [3].

The paper [1] provides a rigorous proof that metamorphic viruses can bypass any signature-based detection, provided the code obfuscation has been done carefully based on a set of specified rules. Specifically, according to [1], if dead code is added and the control flow is changed sufficiently by inserting jump statements, the virus cannot be detected.

In this project we first developed a code obfuscation engine conforming to the rules in [1]. We then used this engine to create metamorphic variants of a seed virus (created using the PS-MPK virus creation kit [15]) and demonstrated the validity of the assertion

in [1] about metamorphic viruses and signature based detectors. In the second phase of this project we validated another theory advanced in [2], namely, that machine learning based methods—specifically ones based on Hidden Markov Model (HMM) —can detect metamorphic viruses. In other words, we show that a collection of metamorphic viruses which are (provably) undetectable via signature detection techniques can nevertheless be detected using an HMM approach.

TABLE OF CONTENTS

INTRODUCTION..........7

A HISTORY OF VIRUS EVOLUTION FROM A DETECTION AVOIDANCE PERSPECTIVE..........9

- Stealth viruses..........9
- Encrypted and Polymorphic viruses..........9
- Metamorphic viruses..........11
- Obfuscation techniques used in metamorphic viruses..........12
- Metamorphic virus generation toolkits..........13
- Other malware self-defense techniques (Rootkits, Packers etc)..........14
- Current state of virus detection techniques..........16
- String scanning or pattern based detection..........17
- Emulation based detection..........18
- Static analysis based detection..........18

HIDDEN MARKOV MODELS APPLIED TO METAMORPHIC VIRUS DETECTION 20

- The Hidden Markov Model (HMM)..........21
- Training the HMM..........21
- Assembly code comparison and scoring..........22

IMPLEMENTATION OF THE METAMORPHIC CODE GENERATOR..........24

- Background theory..........24
- Implementation details..........25
- Detailed description of the code obfuscation process..........27
- Jump statement insertion..........28
- Dead code insertion..........29
- Block re-ordering..........30

EXPERIMENT SETUP AND RESULTS..........32

Experiment setup....32
Test methodology....32
Results....36
CONCLUSIONS AND FUTURE WORK....38
BIBLIOGRAPHY....39
APPENDIX A: Normalized HMM Scores for Metamorphic Viruses and Normal Files 42
Table1: Scores of files with model file 99_virus_N2_E0.model....42
Table2: Scores of files with model file 99_virus_N2_E1.model....43
Table3: Scores of files with model file 99_virus_N2_E2.model....43
Table4: Scores of files with model file 99_virus_N2_E3.model....44
Table5: Scores of files with model file 99_virus_N2_E4.model....45
APPENDIX B: Scatter graph representation of HMM Training and Testing Results
46
49
49
49

TABLE OF FIGURES

Figure 1: How polymorphic viruses evolve with each generation [4]........................11

Figure 2: Evolution of generations of a metamorphic virus [4]................................12

Figure 3: Instruction reordering and jump statement insertion in Zperm [4]..........13

Figure 4: Difference between a packed and unpacked virus [6]................................15

Figure 5: Approximate breakdown of malware self defense techniques in 2007 [6].16

Figure 6: Stoned virus showing the search pattern 0400 B801 020E 07BB 0002 33C9 8BD1 419C [4]...17

Figure 7: Stages in static analysis of virus binaries [15]..19

Figure 8: Average similarity score comparison for metamorphic viruses and normal files 20

Figure 9: Method used to compare assembly programs (virus families and benign programs) [2]...23

Figure 10: HMM similarity scores for different metamorphic virus families [2]......24

Figure 11: Equation to determine the value of integer 'k'..26

Figure 12: Dead code blocks..27

Figure 13: Code obfuscation process in our metamorphic engine.............................28

Figure 14: Separation of virus code into blocks..28

Figure 15: Example of jump statement insertion..29

Figure 16: Insertion of dead code blocks...30

Figure 17: Rearrangement of blocks after shuffling 31

Figure 18: Seed virus being detected by McAfee VirusScan......................................33

Figure 19: Two metamorphic variants generated by our code morphing engine.....34

Figure 20: McAfee VirusScan fails to detect our metamorphic viruses....................35

Figure 21: N = 2, E = 0..47

Figure 22: N =2, E = 1...47

Figure 23: N =2, E = 2...48

Figure 24: N =2, E = 3...48

Figure 25: N =2, E = 4...49

INTRODUCTION

In today's age, where a majority of the transactions involving sensitive information access happen on computers and over the internet, it is absolutely imperative to treat information security as a concern of paramount importance.
Computer viruses and other malware have been in existence from the very early days of the personal computer and continue to pose a threat to home and enterprise users alike. As anti-virus technologies evolved to combat these viruses, the virus writers too changed their tactics and mode of operation to create more complex and harder to detect viruses and the game of cat and mouse continued.

Both viruses and virus detectors have gone through several generations of change since the first appearance of viruses and this thesis is particularly concerned with a recent stage in virus evolution—metamorphic viruses. These are viruses which employ code obfuscation techniques to hide and mutate their appearance in host programs as a means to avoid detection. The most popular virus detection technique employed today is signature based static detection, which involves looking for a fingerprint-like sequence of bits (extracted from a known sample of the virus) in the suspect file. Metamorphic viruses are quite potent against this technique since they can create variants of themselves by code-morphing and the morphed variants do not necessarily have a common signature. In fact, the paper [1] provides a rigorous proof that metamorphic viruses can bypass any signature-based detection, provided the code obfuscation has been done based on a set of specified rules. These rules include dead code insertion and jump statements to obfuscate the control flow.

For this thesis a code obfuscating engine conforming to the rules specified in [1] has been created and using it we demonstrate that viruses obfuscated with this engine are not detectable by commercial virus scanners employing signature based detection. A second

experiment was then carried out to test the hypothesis in [2] that metamorphic viruses can be detected by machine learning based methods (in this case employing Hidden Markov Models or HMMs). The detection engine in [2] was tested against metamorphic viruses generated by our obfuscation engine to determine the effectiveness of this detection approach.

This thesis is organized in the following manner. Chapter 2 provides background information and some history on how viruses evolved, from the point of view of detection avoidance. We also consider various techniques used by virus writers including encryption and code obfuscation. Some background information on the current state of virus detection is also presented. Chapter 3 provides details on the HMM model and its application to the problem of detecting metamorphic viruses. A complete description of the code obfuscation engine created for this project is provided in Chapter 4. Chapter 5 details the experimental setup used for this project and the various experiments performed with the metamorphic code generation engine. Chapter 6 records the conclusions from the experiments and provides some suggestions for future research.

A HISTORY OF VIRUS EVOLUTION FROM A DETECTION AVOIDANCE PERSPECTIVE

Stealth viruses

Virus writers have been employing techniques to avoid detection from the earliest days of computer viruses. One of the first techniques virus writers employed to evade detection was to keep the last modified date of an infected file unchanged to make it seem like it was uninfected. Virus detectors combated this tactic by maintaining cyclic redundancy check (CRC) logs on files to detect infection. Other viruses tried to hide in memory and maintained copies of infected files, taking over system functions for reading files or disk sectors and redirecting virus detectors to the unaffected copies to evade detection. "Brain", the very first PC virus was an example of such a virus which redirected attempts to read infected boot sectors to the area of the disk where the original boot sector was stored [11]. The catch here was that the virus had to be memory resident to do this and virus detectors began to analyze memory as well for evidence of viruses as a countermeasure. Brain also was the origin of the rule of thumb: starting from a clean trusted disk before checking the status of a system.

Encrypted and Polymorphic viruses

The next stage in virus evolution produced viruses which used encryption as a technique to obfuscate their presence. One of the earliest examples of a virus using encryption as an anti-detection technique was Cascade, a DOS virus [11]. Encrypted viruses typically carry along a decryption engine and thus they have to maintain a small portion of the virus body unencrypted. Virus detectors began to tackle these viruses by looking for the signature bits in this unencrypted portion. Oligomorphic viruses then appeared, where the

viruses employed multiple decryption algorithms (one simple way was to carry along multiple decryption engines and pick one at random) making pattern based detection more difficult [12]. Then came polymorphic viruses which were basically encrypted viruses capable of mutating their decryption engines in each generation. Polymorphic viruses created variants of themselves which used a different encryption mechanism in each generation resulting in different decryption engines and thus effectively countering scanners looking for the signature of the decryptor [12].

Polymorphic viruses necessitated further evolution in anti-virus technology and the answer came in the form of static emulation. In this detection technique, the virus decryption process is executed in a controlled environment and the location of the decrypted virus is captured. After decryption, the virus detector can locate a signature string in the decrypted virus and use that to detect subsequent infections of the same virus just as if the virus were unencrypted. Figure 1 below [4] pictorially illustrates how polymorphic viruses evolve with each generation.

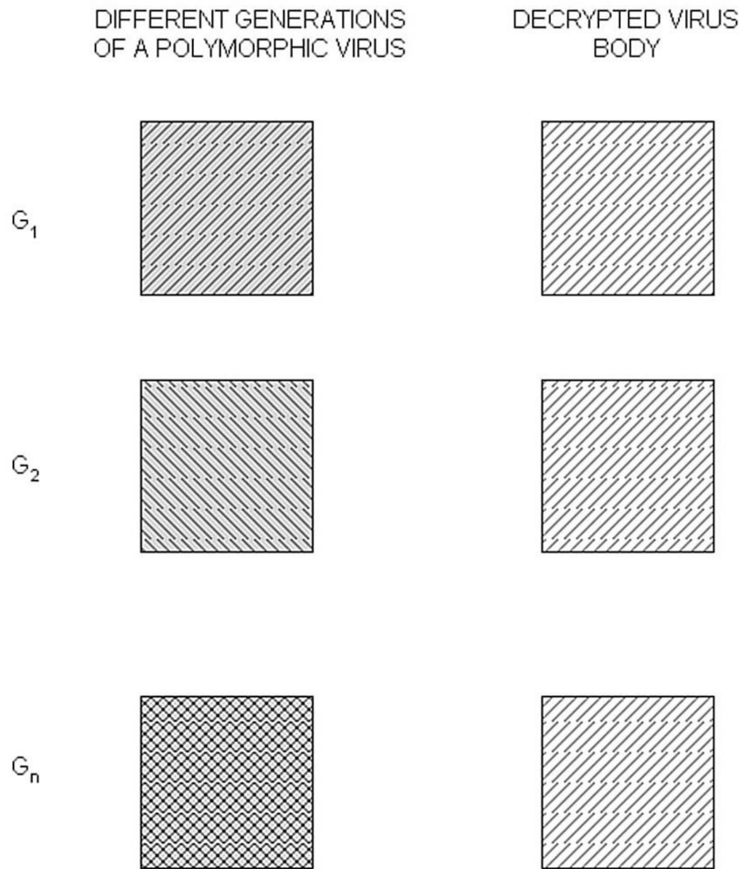


Figure 1: How polymorphic viruses evolve with each generation [4]

Metamorphic viruses

Polymorphic viruses have one major Achilles heel—the virus body is identical in each generation. Therefore, if a polymorphic virus is somehow decrypted it can subsequently be detected by pattern-based detection. Metamorphic viruses were the next stage in virus evolution. These viruses do not rely on encryption as an obfuscation technique but instead mutate their own code structure through operations such as dead code insertion and control flow obfuscation, which yields generational variants that are very different. This is illustrated pictorially in Figure 2 [4]

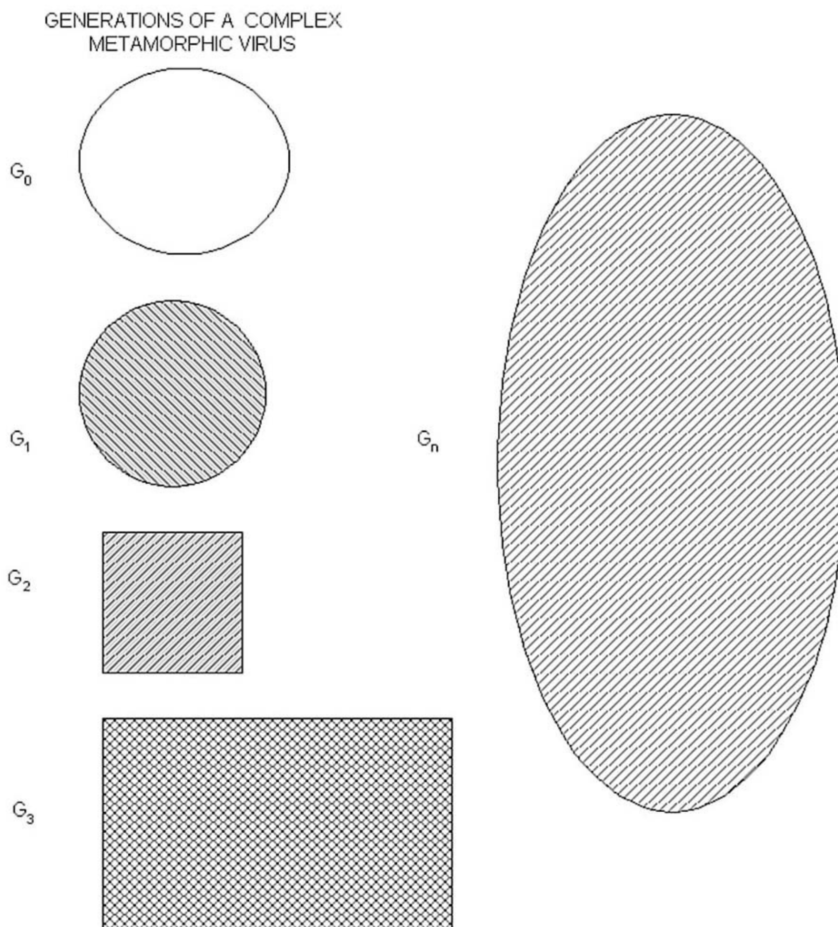


Figure 2: Evolution of generations of a metamorphic virus [4]

Obfuscation techniques used in metamorphic viruses

Metamorphic viruses can obfuscate their data flow by various techniques including register exchange (using different registers in each generation), instruction swap (replacing instructions with other equivalent ones), permutation (subroutine reordering), transposition (reordering instructions which are not order dependant) and dead code insertion (adding nop and other "do nothing" statements).
They can also obfuscate their control flow can by extensive use of jump instructions. Some metamorphic viruses carry their own metamorphic engines. For example, Zperm carries along its own metamorphic engine, which is known as the Real Permuting Engine or RPME [12]. Other metamorphic generators operate "offline", in the sense that the metamorphic engine is independent of the virus itself. Figure 3 [4] illustrates how jump

instructions and instruction reordering are used in the Zperm virus to obfuscate the virus body.

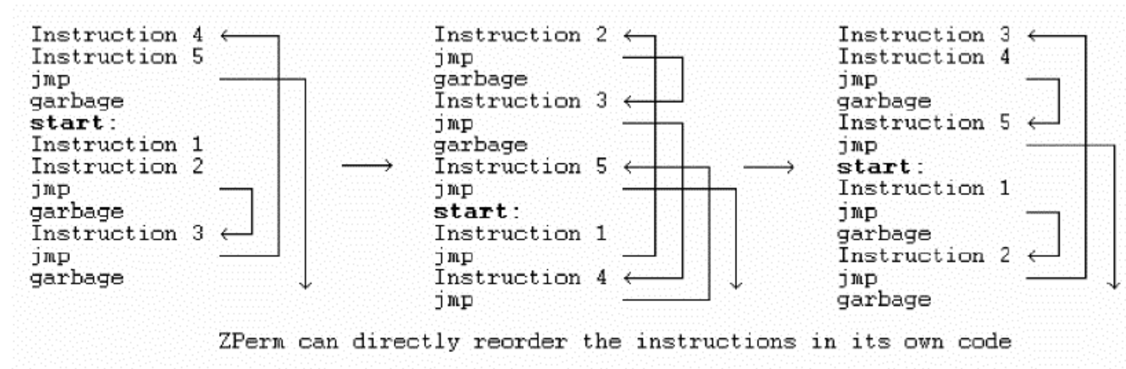


Figure 3: Instruction reordering and jump statement insertion in Zperm [4]

Regardless of the actual technique used to obfuscate the virus body, metamorphic viruses have one shared characteristic which gives them their potency and makes them difficult to detect—they do not provide any moment in their evolution when a constant code body is completely observable. Note that this is in contrast to polymorphic viruses.

Metamorphic virus generation toolkits

Virus writing used to be the purview of a few dedicated “enthusiasts”. However, the past several years have seen the emergence of several virus generation toolkits which has made creating a potent virus very easy. These toolkits range from rudimentary ones to very elaborate tools with GUIs which can generate polymorphic and metamorphic viruses. Some of the more sophisticated toolkits come complete with anti-debugging and emulation resistant techniques built in. VX Heavens [14], which is a resource for virus creators and researchers, lists well over a hundred virus generation toolkits. Some of the more advanced toolkits include the Next Generation Virus Creation Kit (NGVCK), Phalcon/Skism Mass Produced Code Generator (PS-MPC), Mass Code Generator (MPCGEN), etc.

For the purposes of this project the PS-MPC toolkit [17] has been used to generate sample viruses. According to Szor [3], PS-MPC generates viruses that are not only polymorphic but have different decryption routines and structures in variants.

Other malware self-defense techniques (Rootkits, Packers etc)

In addition to the techniques discussed earlier in this section there are several other techniques employed by virus writers to avoid being detected by anti-virus programs. Some of the more common ones include Rootkits, Packers and anti-debugging techniques.

Rootkits are programs that reside in a computer system without authorization and take control of the operating system [6]. They are designed to conceal malicious programs in the system to make it very difficult to detect the malicious programs using antivirus or other security software. Execution Path Modification (modifying a chain of system calls and using API level hooks to hijack system functions) and Direct Kernel Object Modification (modifying information or commands directly in the kernel source) are some common techniques used by Rootkit technologies. The deeper these Rootkits are located in the system the more difficult it is to find them. Newer trends in Rootkits include Firmware rootkits which attack the firmware supplied with devices and Virtualized rootkits which modify the boot sequence, load themselves instead of the original OS and then load the original OS as an enslaved virtual machine [18].

Packers are programs that compress viruses making them difficult to be detected. When virus writers try to create new viruses by building on or modifying existing viruses the heart of the virus remains the same with some extra lines of code. Viruses created in this manner are hence easily detected by many virus scanners using pattern based detection. By packing the files virus creators bypass the problem as changing even one byte in the unpacked executable results in a very differently byte sequenced packed file. Figure 4 [6] below illustrates the difference between a packed and unpacked virus executable.

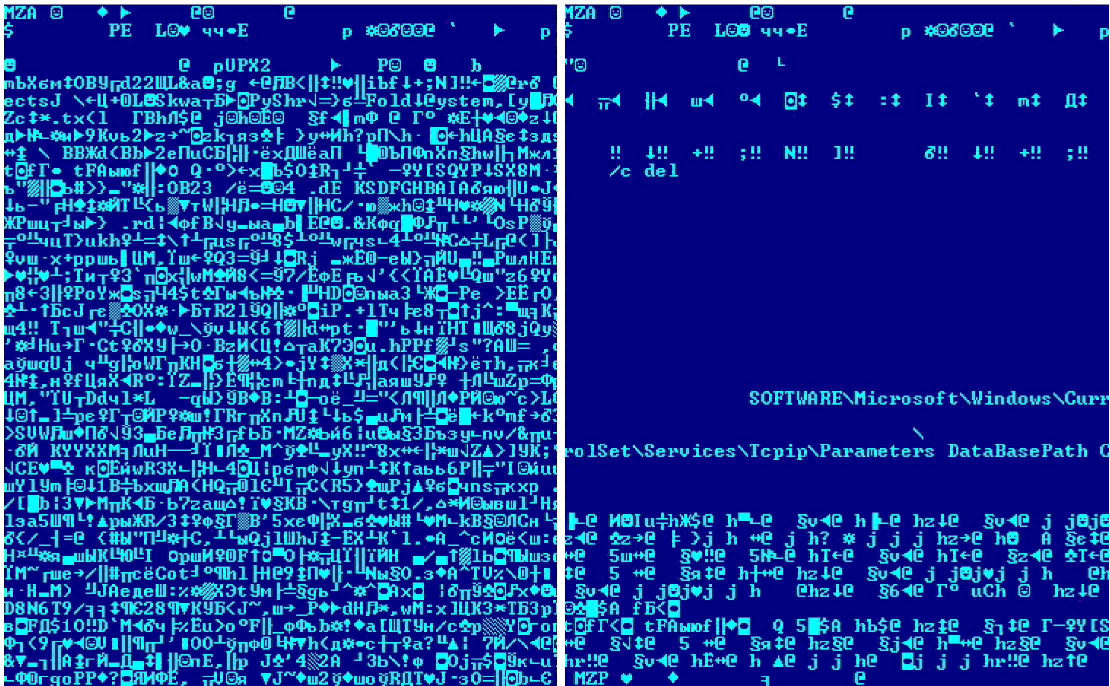


Figure 4: Difference between a packed and unpacked virus [6]

Figure 5 [6] provides a graphical breakdown of the various self defense techniques used by malware writers in the year 2007. We can see that packing was the most popular technique (possibly due to the large return on investment virus writers derive by employing this technique and its simplicity). Encryption and code obfuscation tied for second place with Rootkits. One possible reason Metamorphism was less commonly seen could be because the technique is harder to implement in practice than some of the others. This might however change in the future with the proliferation of metamorphic virus generation toolkits.

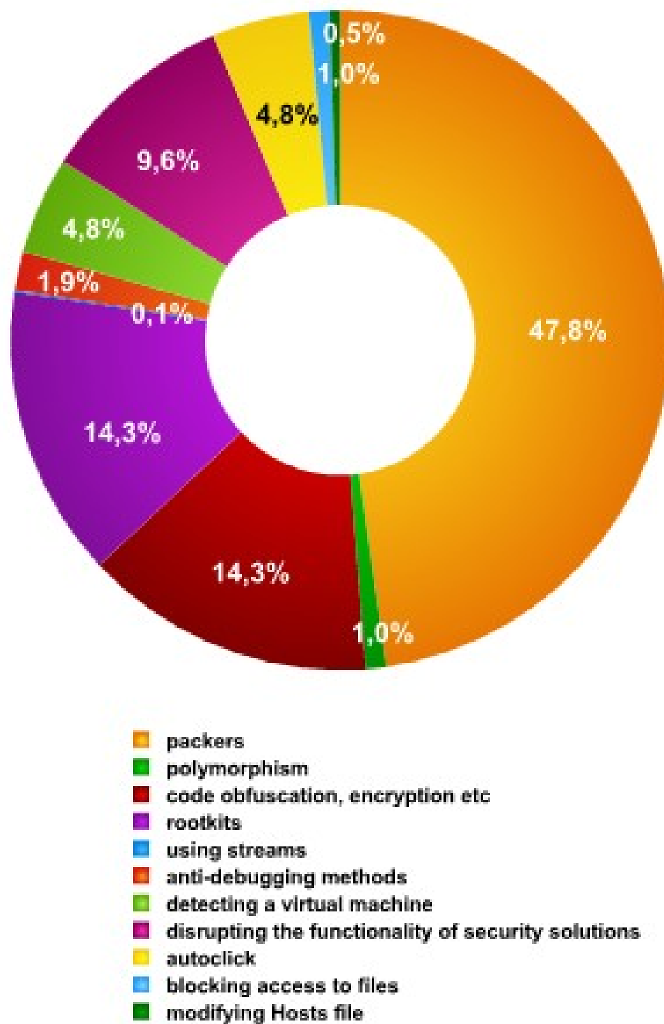


Figure 5: Approximate breakdown of malware self defense techniques in 2007 [6]

Current state of virus detection techniques

Anti-virus technologies today use a variety of techniques to detect viruses. The objectives of these technologies are to detect viruses with a high degree of accuracy, produce very few false positives, and accomplish the detection process in a reasonable amount of time. Some of the different detection techniques employed today includes:

- Pattern based detection

- Emulation based detection
- Static analysis based detection
- Heuristics and statistical methods

Below, we briefly discuss each of these techniques.

String scanning or pattern based detection

The most popular technique in anti-virus scanners today is pattern based detection. It is not as effective as some other techniques but it can be performed more quickly. This technique involves extracting a unique sequence of bits from a known virus and this sample is subsequently used like a fingerprint to match against while scanning for existence of the virus. Care has to be taken when choosing the bit sequence to minimize the number of false positives and at the same time match the virus and (ideally) possible variants. Sometimes statistical techniques are also used to extract these patterns. Figure 6 [4] shows an example of a search pattern for the "Stoned" boot sector virus. In this case, the bit sequence selected was chosen by observing a behavioral peculiarity of the virus (it reads the boot sector of the diskette four times, resetting the disk between each try).

```
seg000:7C40 BE 04 00               mov     si, 4            ; Try it 4 times
seg000:7C40                                                 ;
seg000:7C43
seg000:7C43             next:                               ; CODE XREF: sub_7C3A+27↓j
seg000:7C43 B8 01 02               mov     ax, 201h         ; read one sector
seg000:7C46 0E                     push    cs
seg000:7C47 07                     pop     es
seg000:7C48                        assume es:seg000
seg000:7C48 BB 00 02               mov     bx, 200h         ; to here
seg000:7C4B 33 C9                  xor     cx, cx
seg000:7C4D 8B D1                  mov     dx, cx
seg000:7C4F 41                     inc     cx
seg000:7C50 9C                     pushf
seg000:7C51 2E FF 1E 09 00         call    dword ptr cs:9   ; int 13
seg000:7C56 73 0E                  jnb     short fine
seg000:7C58 33 C0                  xor     ax, ax
seg000:7C5A 9C                     pushf
seg000:7C5B 2E FF 1E 09 00         call    dword ptr cs:9   ; int 13
seg000:7C60 4E                     dec     si
seg000:7C61 75 E0                  jnz     short next
seg000:7C63 EB 35                  jmp     short giveup
```

Figure 6: Stoned virus showing the search pattern 0400 B801 020E 07BB 0002 33C9 8BD1 419C [4]

Second generation pattern based detectors use more advanced techniques such as "smart scanning" (ignoring nop instructions), using wildcards (allowing skipping of bytes and byte ranges), generic matching (using a single string to potentially match a family of viruses), near exact identification (using two search strings instead of one), using a checksum of a constant range found in the virus body and, finally, the most accurate method—exact identification (using checksums of all the constant bits found in the virus).

Emulation based detection

Emulation based detection is a powerful anti-virus technique where the virus is executed in a controlled environment (a virtual machine, or VM, emulating the instructions of the real processor and the interface of the operating system) and the behavior of the virus is observed. This technique is particularly useful with polymorphic and encrypted viruses where the virus is allowed to decrypt itself and then a snapshot of the decrypted virus can be captured for analysis from the virtual machines memory structures.

One drawback of emulation-based detection is that the virus execution in the VM environment can sometimes take relatively long, especially when the virus has many garbage instructions in a loop. Code optimization techniques are sometimes applied in such cases for faster execution.

Static analysis based detection

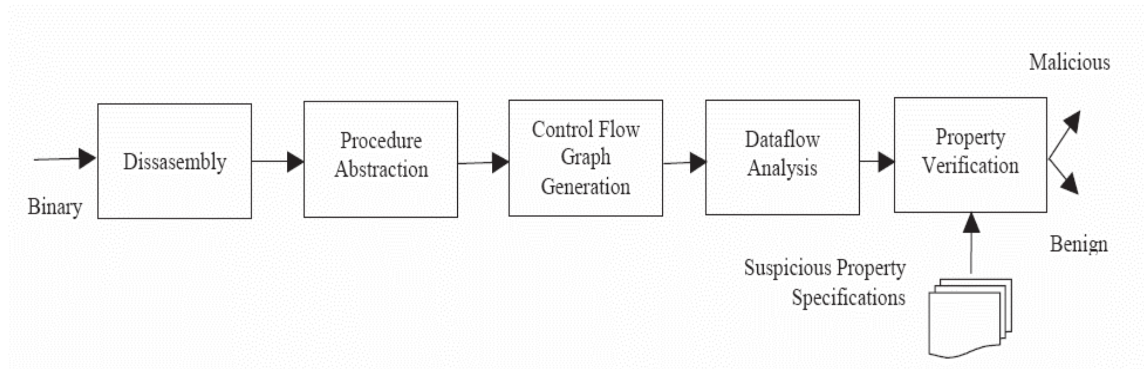


Figure 7: Stages in static analysis of virus binaries [15]

In this detection method, heuristic and formal analysis techniques are used to analyze the virus after it has been taken through several stages of information recovery. The stages in static analysis are depicted in Figure 7 [15]. The first stage involves disassembling he virus binary. The most common technique in this step is the linear sweep approach used in interactive debuggers like IDA Pro. Once the assembly level instructions have been recovered, the next stage involves determining procedural boundaries and obtaining a control flow graph (CFG) representation of the program. After this data flow analysis is performed on the CFG to find out instructions which modify the memory locations or registers used by other instructions.

Finally in the property verification stage, a directed graph based on the code is compared with a formal representation of suspicious activities/properties and a determination is made on whether the program is malicious or benign. Model checking against a finite state machine representation of the suspicious properties is a common static analysis approach.

In addition to the detection methods discussed in detail in this chapter, other methods like statistical analysis and machine learning based methods have also been used. One such technique (HMMs) will be discussed in detail in the next chapter.

HIDDEN MARKOV MODELS APPLIED TO METAMORPHIC VIRUS DETECTION

Metamorphic viruses have an interesting property which make behavioral analysis based approaches a viable option for detecting these viruses [2]. Specifically – the generational variants of the same metamorphic virus family despite their differences do share a high degree of similarity especially when compared to normal files because they tend to differ a lot from normal files. This can be seen from Figure 8 [2] which shows a comparison of the average similarity scores computed using HMM for a family of metamorphic viruses and a set of normal files.

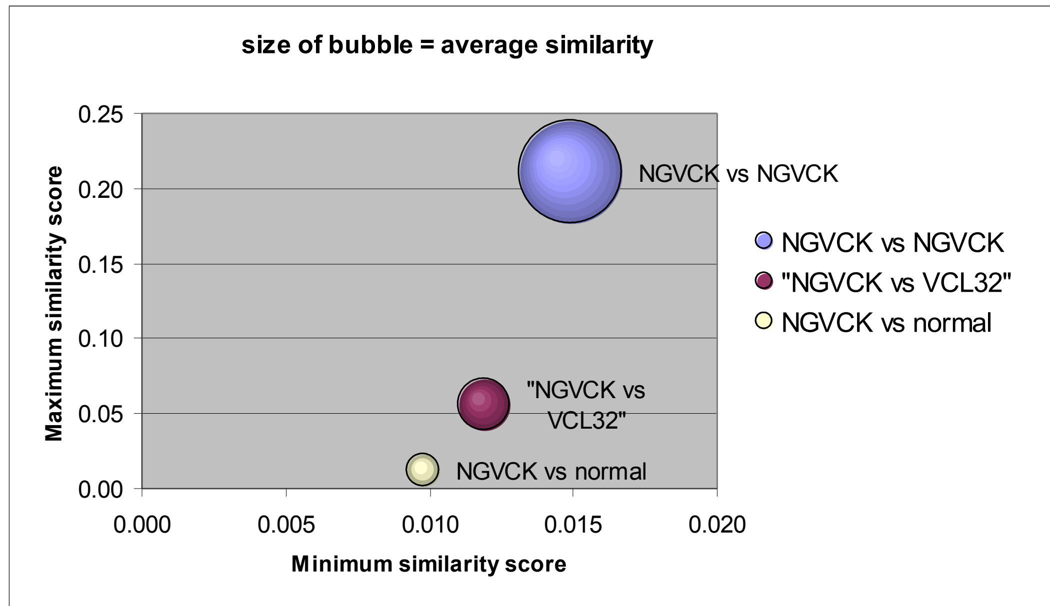


Figure 8: Average similarity score comparison for metamorphic viruses and normal files

Wing Wong and Mark Stamp propose in [2] the application of Hidden Markov Model (HMM) based statistical analysis to the detection of metamorphic viruses to take advantage of this property. Their idea is to use a two step approach - HMM based modeling is first used to represent the statistical properties of a family of metamorphic viruses (i.e. the model is trained on a metamorphic virus family) and then later the trained model is used to determine whether a given program is similar to the virus or different.

The second phase of this project aims to demonstrate that viruses created by our code obfuscation engine can be identified by this HMM based method described in [2].

The Hidden Markov Model (HMM)

Hidden Markov models (HMMs) are state machine based statistical models which can be used to describe a set of observations generated by a stochastic process. Such processes (also called Markov processes) can be modeled as a sequence of states, where the progression to the next state depends solely on the present state but not on the past states. The underlying stochastic process modeled in a HMM is "hidden" and all we can see is the sequence of observations associated with the states. The idea here is to make use of the information observed about the process to gain an understanding of the underlying Markov process [18]. HMMs are well suited for statistical pattern analysis and have been applied to solve various problems of this nature including speech pattern analysis and biological sequence analysis.

Training the HMM

1.1.1.

When a HMM is trained on a particular data set the states in the model represent features of the data set under observation and are associated with a probability distribution for the set of symbols under observation. The state transitions represent the transition probabilities between the observed states and have fixed values.

In [2] where HMM was applied to the problem of recognizing metamorphic viruses, the HMM states corresponded to features of the virus code, while the observations about the data (in this case metamorphic viruses) were instructions or opcodes making up the virus

program. The idea here was that the HMM should after training be able to detect similarities between (and assign high probabilities to) the viruses from the same metamorphic family the model was trained on.

1.1.2.

Assembly code comparison and scoring

The comparison process used in [2] is graphically depicted in Figure 9 [2]. The process was first outlined by Mishra in [16] and is based on finding identical opcode sequences in the two programs. The first step is to extract opcodes from the program (comments, labels etc are excluded). Each opcode is then assigned a number and the sequence of opcodes in the two programs is compared to find common subsequences of size three. The match locations in the code in one program X are then plotted against match locations in the other program Y. Identical code segments thus appear as line segments parallel to the main diagonal (for the case where the programs have identical sizes the main diagonal is the 45 degree line).

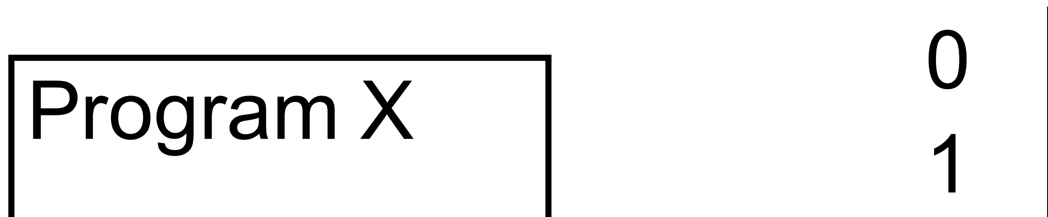


Figure 9: Method used to compare assembly programs (virus families and benign programs) [2]

In paper [2] Wing Wong presented the results for the above comparison performed on four different families of viruses (created using 4 different metamorphic virus generation kits: NGVCK, G2, VCL32, MPCGEN) and a set of normal files. These results are shown in Figure 10 below. We can see that viruses from the same family score very similar and the scores are noticeably different from those for the normal files. The MPCGEN and VCL32 families share some overlap in their scores indicating that the generators create similar viruses and probably perform similar morphing operations. NGVCK clearly performs much better than the other virus generator kits in creating viruses which look very different from other viruses and normal files. Interestingly enough it is this exceptional ability to look different which helps the HMM recognize viruses from this family

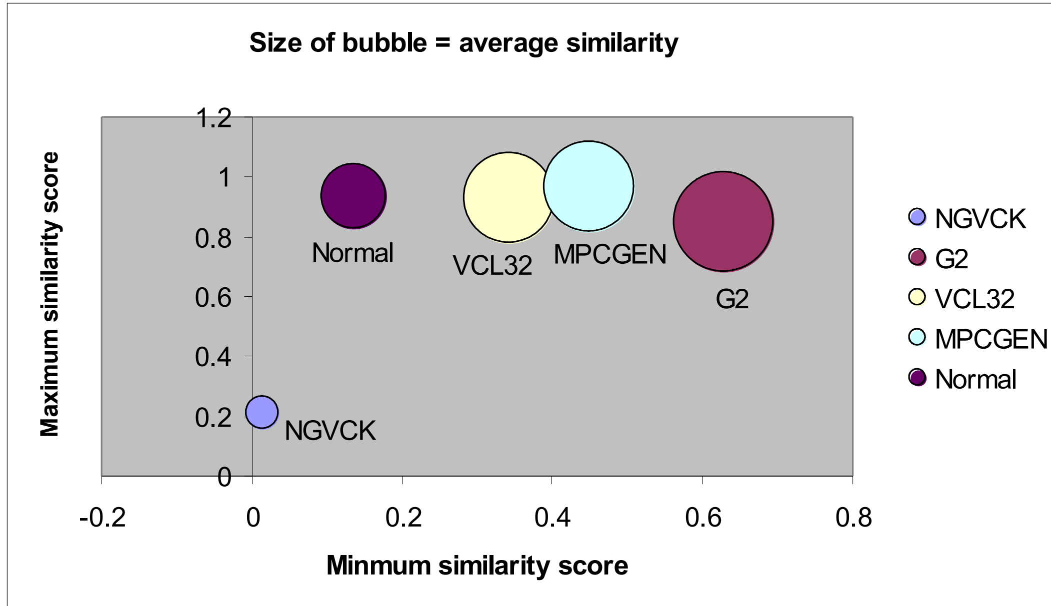


Figure 10: HMM similarity scores for different metamorphic virus families [2]

In phase three of our experiments we trained the HMM model described in [2] on metamorphic viruses created by our code obfuscation engine and then determined the similarity scores for other variants from the same family and also normal files. The experiment details and results are presented in chapter 5.

IMPLEMENTATION OF THE METAMORPHIC CODE GENERATOR

For this project we implemented a code morphing engine in Perl confirming to the specifications in [1]. This engine was intended to work with any given block of assembly code.

Background theory

The authors in [1] advance formal proofs for their specific code morphing suggestions. Their contention is that the assembly code of the original virus should first be separated into small blocks of code based on two basic conditions. The first condition being that no block should end with any kind of jump instruction (JMP, JNZ, JGE etc). The second condition being that no block should end with a NOP operation. They also require that the virus carry its own metamorphic engine (i.e. the virus should know how to strip out the garbage code and re-order the blocks without outside assistance). From a virus detection point of view it is even harder to detect metamorphic viruses which do not carry their own metamorphic engine, hence we ignored this restriction in [1] and made the code morphing engine a separate entity.

After the code is separated into blocks the order of the code blocks has to be randomly shuffled. After the blocks are shuffled, small blocks of dead code (also known as garbage code) have to be inserted between the blocks of original code. Dead code is a block of code which is syntactically correct but semantically irrelevant to the program being executed. Once the dead code is added, the correct flow of the virus code is controlled by the result achieved from a mathematical equation which always computes to the same value. The idea is to use an equation which always results in the same result (condition always true or always false) but at the same time is a sufficiently complex expression that it is difficult analyze from assembly code.

Implementation details

For our project we chose a fixed block size of three for simplicity. Care was taken while splitting the code into blocks to make sure that none of the blocks ended with a jump instruction or a NOP instruction. If either of these types of instructions happened to be the last instruction of the block then we included the instruction succeeding the jump/NOP into the same block.

After the blocks were created, the starting address of each block was stored in an array and a conditional jump instruction pointing to the next block was added at the end of each block. This jump instruction was constructed depend on the result of a relatively complex mathematical equation. Complexity here implies that by manually reading the equation it is not apparent that the result is always the same for a set of given values. Since the equation always gives the same result, in all versions of the virus the jump instruction will always point to the logically correct sequence of blocks. Once these jump statements were inserted the blocks were randomly shuffled and blocks of dead code were inserted between blocks.

This project was implemented in three principal modules. The first module was designed to count the number of lines in the entire block of code and divide the program into smaller blocks of code. After that the second module stored the program in an array and appended conditional jump instructions to the end of each code block. The condition used to determine the value of integer ‘k’ is as follows:

```
If (((a+1+a) %2 * (a^2 / a)) % 2 == 0) {
k=1;
} else {
k=0;
}
```

Figure 11: Equation to determine the value of integer 'k'

The letter ‘a’ in the above equation refers to any integer value. This equation will always result in k=1 for even values of ‘a’ and k=2 for odd values of ‘a’. Here the integer k determines the jump condition. The third module performed the process of obfuscation. This was achieved in two steps. In the first step some small blocks of dead code were added at the end of the array storing the generated code blocks. The dead code blocks used were also the same as the ones mentioned in [1]. They are as follows:

G_1	xor ebx, ebx
	mov ebp, 24h
	mov eax, 26h

G_2	mov ebp, esp
	xor edi, edi
	mov eax, 1F03

Figure 12: Dead code blocks

In the second step the small blocks of codes were randomly shuffled, in other words the logical order was changed and the results were stored in a text file. The above process i.e. the second step of third module was repeated multiple times (124 times in the case of this project) and the result is stored in different text files.

Care was taken while changing the logical order of the block to ensure that the first block was the same as that in the original code. According to authors in [1], all metamorphic viruses created by this engine always have the same entry and exit points/blocks. Hence the virus was not parsed once it has reached the end of the last block. Though the blocks were linked using the conditional jumps, the original logical sequence could not be achieved unless the first block was parsed first. The following sections provide more detailed descriptions of the code obfuscation process performed in our engine.

Detailed description of the code obfuscation process

The sequence of transformations performed by our code obfuscation engine is shown in Figure 13. The virus code is first broken down into fixed size blocks. Blocks of dead code are then inserted followed by jump statement insertion and reordering of the blocks. Each step in the transformation will be explained in detail in subsequent sections.

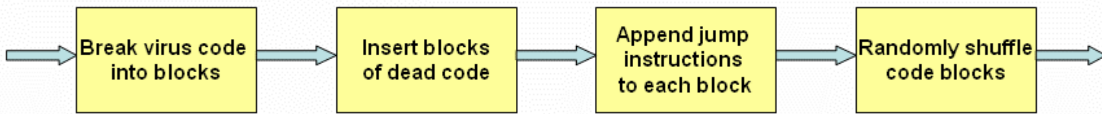


Figure 13: Code obfuscation process in our metamorphic engine

The first step in the obfuscation was breaking the code into fixed size blocks (Figure 14). One important thing that we had to take care of in this stage was to make sure some sections of the assembly code, which needed to remain together like the .stack and .data sections, did not get split into different blocks.

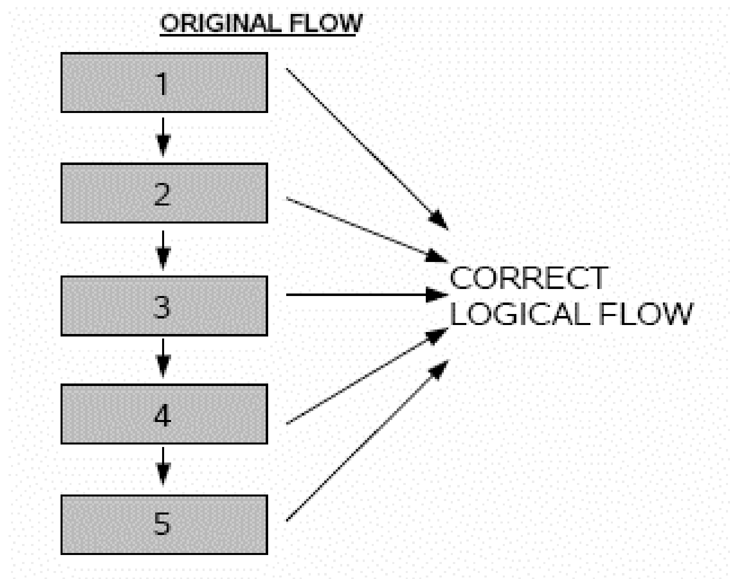


Figure 14: Separation of virus code into blocks

Jump statement insertion

At the end of the first step the blocks were still all in logically correct order.

The next step after chopping the code into these blocks involved the insertion of jump statements and this is depicted in the Figure 15 below.

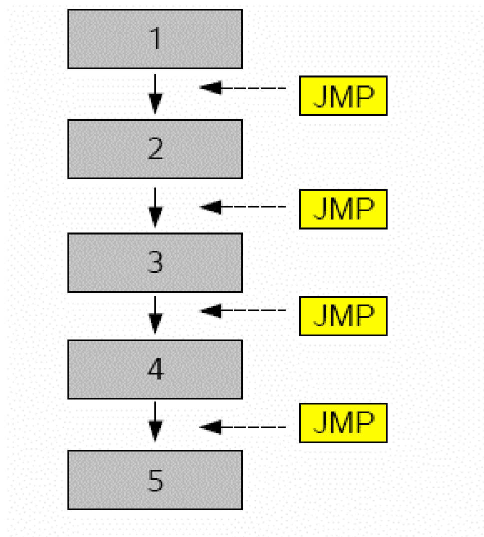


Figure 15: Example of jump statement insertion

Dead code insertion

Once the block of conditional jump instructions were attached at the end of each block (Figure 16). The blocks were stored in an array where each element in the array is a set of instructions and at the end of the array more dead code blocks were added. Each dead code block was stored in a singly array element. This increased the size of the array by the total number of garbage blocks.

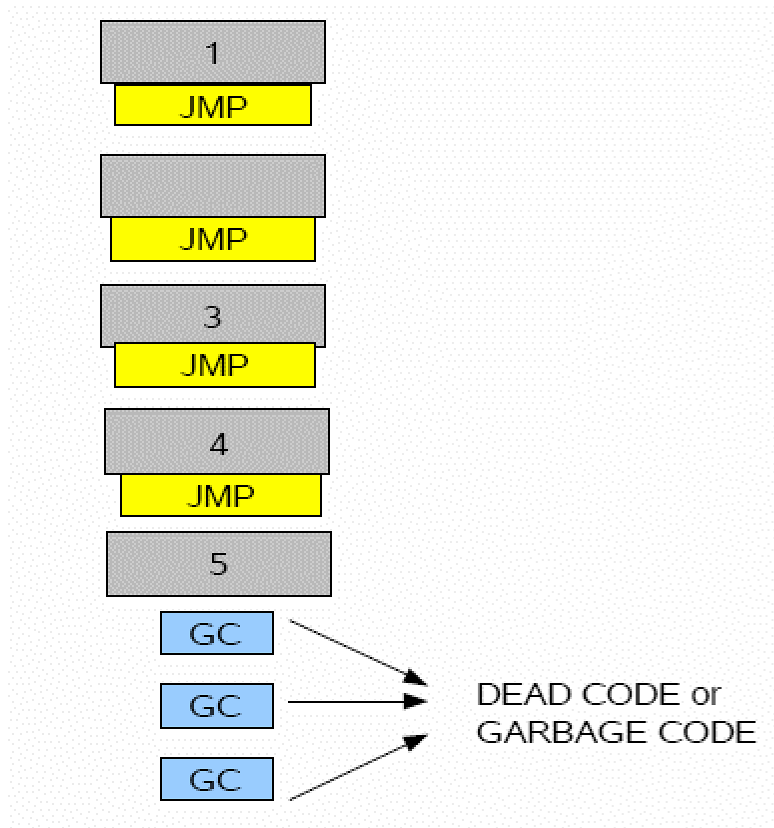


Figure 16: Insertion of dead code blocks

Block re-ordering

After the garbage code insertion the blocks were randomly shuffled. Figure 17 shows the control flow after this shuffle and this can be compared to the original code in Figure 14. The thing to note here is that the entry point for the virus always needs to be the original starting block. Thus block 1 being the starting block remains the same for all versions of the metamorphic virus. Similarly the program always ends with the end of last block and there is no garbage code or jump introduced after that.

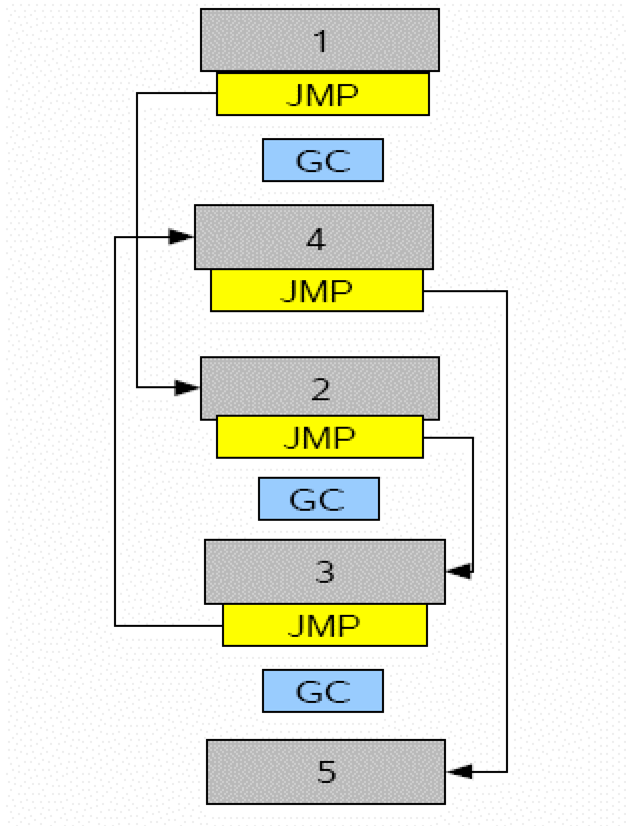


Figure 17: Rearrangement of blocks after shuffling

EXPERIMENT SETUP AND RESULTS

Experiment setup

Experiment platform:	Windows XP, VMware virtual machine
Programming language:	Perl5
Dis-assemblers:	OllyDbg and IDA pro. (Both free download versions)
Assembler:	MASM
Linker:	Tlink
Virus generator:	PS-MPC Phalcon/Skism mass produced code generator
Virus scanner (for baseline check):	McAfee VirusScan

Test methodology

The experiments in this project consisted of three major phases. The first phase involved creating the seed virus required for this project and running baseline checks on the seed virus using pattern based detectors. Phase two involved running our code obfuscation engine on the seed virus to generate a family of metamorphic variants of the seed virus. The final phase involved testing the metamorphic viruses created by our engine using pattern based detectors and a HMM based detector.

1.1.3. ***Creation of the seed virus***

The virus generator used for creating the seed virus for this project was the Phalcon/Skism Mass Produced Code Generator (**PS-MPC**) from vxheavens.com [15]. For this experiment the viruses we created were unencrypted. The PS-MPC virus creator generated the assembly language code for the virus which we assembled using **MASM** assembler and converted into an executable using the **Tlink** linker. After this the virus executable was scanned using the McAfee VirusScan scanner which recognized it as a virus and flagged a warning. Figure 18 shows a screenshot of the result we obtained when we ran McAfee VirusScan on the virus created using PS-MPC.

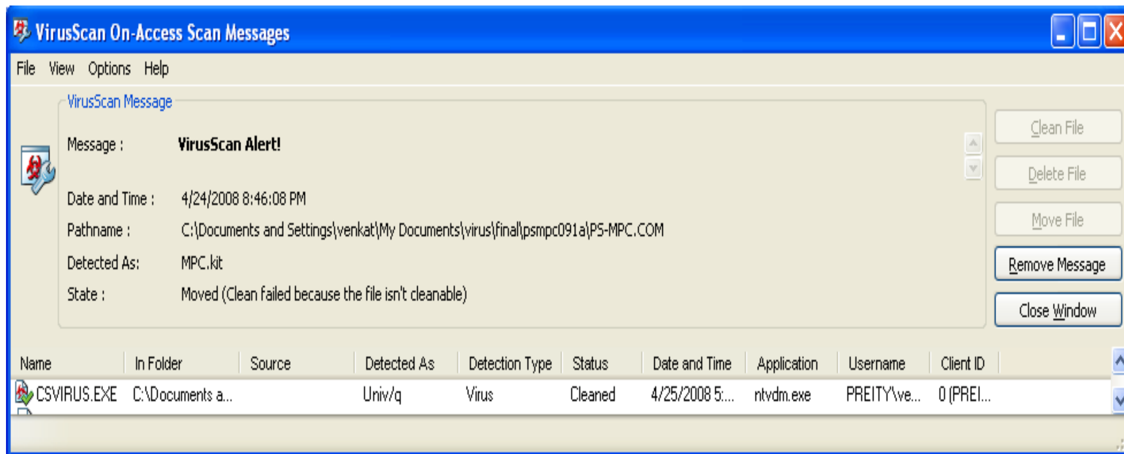


Figure 18: Seed virus being detected by McAfee VirusScan

1.1.4. ***Creation of metamorphic variants***

After making certain that the seed virus was detected by a pattern based scanner it was run through our code morphing engine to create metamorphic versions. For the purposes of our experiment 120 variants of the seed virus were created. The code morphing engine reads the assembly code for the virus divides the code into blocks and then randomly shuffles the block order while simultaneously inserting some dead code blocks.

Figure 19 shows a side by side comparison of two variants created by our code obfuscation engine (VIRUS1.asm and VIRUS2.asm) and illustrates the difference in code between the metamorphic variants. We can see the labels and the jump instructions inserted between the blocks and the differences in the block order. It is also evident that we keep the starting block in the same place.

VIRUS1.ASM - Notepad

File Edit Format View Help

```
; csvirus.asm : [Skeleton] by Deke
 ; Created wik the Phalcon/Skism Mass-Proc
 ; from the configuration file SKELETON.CF

 .model tiny                                   ;
 .code                                         ;
           org     100h                        ;
 jmp labelvirus1


labelvirus22:
           call attributes                     ;

           mov   ax,3d02h                      ;
           int   21h
 jmp labelvirus23


labelvirus28:
           add   cx,heap-startvirus+3          ;
           cmp   ax,cx                         ;
           je    find_next
 jmp labelvirus29


labelvirus42:
           sbb   dx, 0                         ;

           mov   cx, 10h                       ;
           div   cx                            ;
 jmp labelvirus43


labelvirus4:
           je    restoreEXE
 restoreCOM:
           lea   si,[bp+offset save3]
           mov   di,100h
 jmp labelvirus5


labelvirus10:
           mov   ah,1Ah                        ;
           lea   dx,[bp+offset newDTA]         ;
           int   21h

 jmp labelvirus11
```

VIRUS2.ASM - Notepad

File Edit Format View Help

```
; csvirus.asm : [Skeleton] by Deke
 ; Created wik the Phalcon/Skism Mass-Prod
 ; from the configuration file SKELETON.CF

 .model tiny                                   ;
 .code                                         ;
           org     100h                        ;
 jmp labelvirus1


labelvirus2:

 startvirus:                                   ;
           call next                           ;
 next:
 jmp labelvirus3


labelvirus34:
           jnz   find_next
           pop   ax                            ;
           jmp   done_infections

 jmp labelvirus35


labelvirus6:
           push ds
           push es
           push cs                             ;
           pop  ds
 jmp labelvirus7


labelvirus19:
 creator   db '[MPC]',0                        ;
 virus     db '[Skeleton]',0
 author    db 'Deke',0

 jmp labelvirus20


labelvirus11:
           lea   dx,[bp+offset exe_mask]
           call infect_mask
           lea   dx,[bp+offset com_mask]
           call infect_mask
 jmp labelvirus12
```

Figure 19: Two metamorphic variants generated by our code morphing engine

After creating the metamorphic variants of the original virus we assembled and linked these variants using the MASM assembler and Tlink linker and created executables for each of them (a Perl script was used to automate this process)

1.1.5. ***<u>Testing metamorphic variants with commercial virus scanners</u>***

In the third phase of our experiments the metamorphic viruses created in the second phase were tested with an off-the-shelf scanner and a HMM based detector.

First the metamorphic viruses were scanned using the same scanner (McAfee VirusScan) used for checking the seed virus and it failed to detect the presence of any virus. Figure 20 shows a screenshot capture of McAfee VirusScan after it was run on the folder containing the 120 metamorphic virus executables generated by our code obfuscation engine.

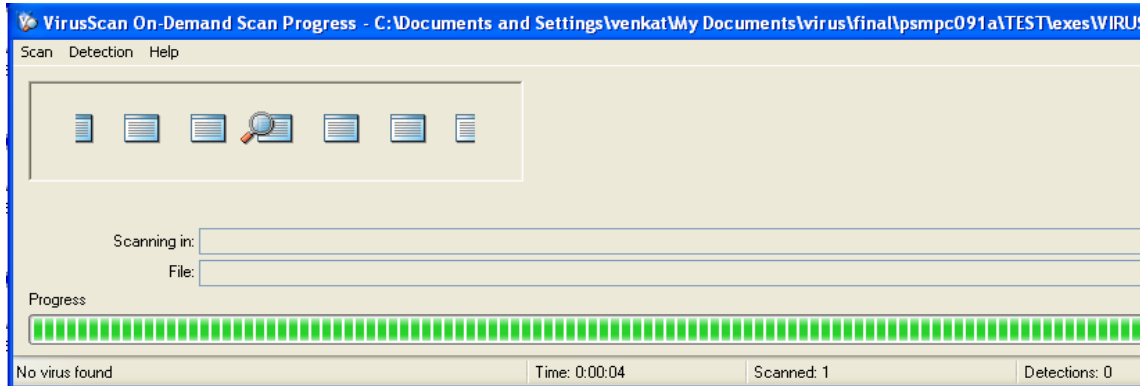


Figure 20: McAfee VirusScan fails to detect our metamorphic viruses

1.1.6. ***Testing metamorphic variants using HMM based detection***

Next our metamorphic viruses were tested against the HMM detector. First the executables were disassembled using the IDA pro disassembler and these assembly files were used for training the HMM.

Our naming convention was to name all the files containing virus assembly code with the prefix "IDAN" and to name all the files containing benign (normal) assembly code with the prefix "IDAR". Prior to HMM training the all the training files were in passed through the train-test module to create the alphabet and input files. The alphabet file contains the different observation symbols present in the training files and the input file contains the frequency of the observation symbols in the training files. We divided the 124 virus files into 5 sets of 24 files each as we performed k-fold HMM validation (in our case 5-fold validation). For each fold we used 4 sets different metamorphic versions of the original virus for training and creating a model file and used the 5th set for testing. For this experiment after running the four sets of IDAN files through the module the value for number of observation symbols were between 42 ~ 44 and the total number of

observation symbols present were between 21739 ~ 21959. We trained the files and created the model for 2 states. The number of iterations was set to be maximum of 800.

Results

Once the HMM was trained the scores against the model file were obtained for both normal files and files belonging to the virus family. The scores for the viruses ranged between -2 to -8 (Appendix A) but for the normal files the scores ranged between -37 to -190 hence clearly separating the viruses from the normal files.

Figure 21: Scatter graph of scores of normal and virus files with one of the model files

Figure 21 shows a scatter graph comparison of the HMM scores for the files from the metamorphic virus family we created with our engine and the scores for normal (benign) files. The figure provides a clear validation of the hypothesis in [2] about the property of metamorphic viruses being very similar to each other and very different from normal files. Similar results were obtained in all 5-fold validation with their respective model and test files. The complete results are presented in appendices A and B.

CONCLUSIONS AND FUTURE WORK

The principal aim of this project was to show that viruses that are provably undetectable using signature-based scanning, can nevertheless be reliably detected using machine learning techniques.. To this end we created a code obfuscation engine conforming to the rules in [1]. According to a proof given in [1], these viruses cannot be detected using signature-based scanning. This was validated, since the metamorphic viruses created by our engine were not detected by the same signature-based detectors that had successfully identified the seed virus the metamorphic variants were created from.

We then demonstrated that the metamorphic viruses created using our code obfuscation engine could be detected by the HMM based detector described in [2]. This was done by performing five-fold HMM validation on 120 different metamorphic viruses and comparing the normalized similarity scores for viruses and normal programs. In all the cases the score ranges for the viruses were markedly different from those for the normal files, hence the viruses were identifiable in the HMM method by their similarity scores alone. In this way we were able to provide empirical proof that metamorphic viruses

undetectable by pattern based scanners can be detected by machine learning based methods.

A good future research project would be to design a metamorphic virus-creating engine that can evade both signature-based detection and HMM-based detection. This however is not a trivial task since it means the virus would have to be highly metamorphic to avoid signature based detection and at the same time it would also need to look like normal code (in terms of the statistical signature of its instruction sequence) to evade HMM-based detection.

BIBLIOGRAPHY

[1] Jean-Maries Borello and Ludovic Me, "Code Obfuscation Techniques for Metamorphic Viruses", Feb 2008, <http://www.springerlink.com/content/233883w3r2652537/>

[2] Wing Wong and Mark Stamp, “Hunting for Metamorphic Engines”, September 2006

[3] Peter Ferrie and Frederic Perriot, “Detecting Complex Viruses”, December 2004, <http://www.securityfocus.com/infocus/1813>

[4] Peter Szor, “The art of computer virus research and defence”, February 2005, Symantec press

[5] Peter Szor and Peter Ferrie, “Hunting for Metamorphic”, September 2001 <http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf>

[6] Alisa Shevchenko, “The Evolution of Self-Defense Technologies in Malware”, July 2007, <http://www.net-security.org/article.php?id=1028&p=1>

[7] Arun Lakhotia, Aditya Kapoor and Eric Uday Kumar, “Are metamorphic viruses really invincible?”, December 2004, <http://www.cacs.louisiana.edu/~arun/papers/invincible-part-1-vbtn-dec2004.pdf>

[8] J. Kephart, A. William, “Automatic Extraction of Computer Virus Signatures”, Proceedings of the 4th International Virus Bulletin Conference, R. Ford, ed., Virus Bulletin Ltd., Abingdon, England, pp. 178-184, 1994. <http://www.research.ibm.com/antivirus/SciPapers/Kephart/VB94/vb94-node1.html>

[9] Computer knowledge virus tutorial, “Stealth viruses and Rootkits”, <http://www.cknow.com/vtutor/StealthVirusesandRootkits.html>

[10] M. Stamp, “A Revealing Introduction to Hidden Markov Models”, January 2004, <http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf>

[11] virus-scan-software.com, “A history of computer viruses”, <http://www.virus-scan-software.com/virus-scan-help/answers/the-history-of-computer-viruses.shtml>

[12] Peter Szor, “Advanced code evolution techniques and computer virus generation toolkits”, March 2005, <http://www.informit.com/articles/article.aspx?p=366890>

[13] IDA Pro Disassembler, <http://www.datarescue.com/idabase>

[14] Myles Jordan, “Dealing with metamorphism”, Virus Bulletin, October 2002, <http://ca.com/us/securityadvisor/documents/collateral.aspx?cid=48051>

[15] VX Heavens, <http://vx.netlux.org/>

[16] P. Mishra, “A taxonomy of software uniqueness transformations”, master’s thesis, San Jose State University, Dec. 2003, <http://home.earthlink.net/~mstamp1/mss_v.html#masters>

[17] Anti-virus test center, University of Hamburg, Germany, “Profile of Phalcon/Skism Mass Produced Code Generator”, January 1993, <http://www.informatik.uni-hamburg.de/AGN/catalog/msdos/html/ps-mpc.htm>

[18] Wikipedia, “Rootkits”, < http://en.wikipedia.org/wiki/Rootkit>

APPENDIX A: Normalized HMM Scores for Metamorphic Viruses and Normal Files

Ta*ble1*: Scores of files with model file 99_virus_N2_E0.model

SCORES OF FILES WITH N=2	
Virus Files	**Normal Files**
-5.5029656	**-69.83376868**
-2.4325415	**-37.87830767**
-2.4381414	**-73.26371666**
-2.4399423	**-57.42153619**
-2.421805	**-47.56689641**
-2.452382	**-43.65816532**
-8.4422612	**-48.48792908**
-2.4204001	**-46.23059133**
-2.4159194	**-74.50382075**
-2.4283433	**-44.05335671**
-2.4175014	**-93.58188728**
-2.4455148	**-57.92270389**
-2.5358185	**-88.1007371**
-2.428392	**-191.0725346**
-2.4169905	**-65.19475889**
-2.4258693	**-34.2368562**
-2.423987	**-43.03456294**
-2.5501224	**-47.10429071**
-2.4327782	**-83.10736693**
-2.4156856	**-76.51210052**
-2.4328526	**-56.64371914**
-2.4408223	**-64.4991311**
-2.4134945	**-61.56445913**
-5.5318651	**-103.5941142**

Table2: Scores of files with model file 99_virus_N2_E1.model

SCORES OF FILES WITH N=2	
Virus Files	Normal Files
-2.42929	-43.0274
-2.42501	-47.0966
-5.49969	-83.1067
-2.4176	-76.5089
-2.41623	-56.6362
-2.42202	-65.2135
-2.41515	-63.3242
-5.53471	-103.593
-2.4268	-79.5003
-2.55107	-75.1983
-2.4392	-70.4338
-2.41701	-42.7951
-2.4177	-50.9171
-2.39951	-62.3496
-2.44303	-41.869
-2.42031	-81.0822
-2.40882	-185.725
-2.42636	-69.1235
-2.41216	-95.3118
-2.43944	-52.4204
-2.42554	-46.5878
-2.41714	-201.674
-2.41746	-102.636
-5.46022	-61.9248

Ta*ble3*: Scores of files with model file *99_virus_N2_E2*.model

SCORES OF FILES WITH N=2

Virus File	Normal File
-2.4244	-48.3463
-2.45852	-46.1009
-2.44365	-74.423
-2.42288	-43.9442
-2.45275	-93.4728
-2.46126	-57.7988
-2.58164	-88.0005
-2.47875	-190.972
-2.43661	-65.1368
-2.45632	-34.1023
-2.44438	-42.9306
-2.46801	-46.9584
-5.48544	-83.0168
-2.56762	-73.1635
-2.43439	-56.5274
-5.55586	-64.424
-2.45565	-61.4528
-2.45591	-103.484
-2.46459	-78.8726
-2.44951	-72.1967
-5.53626	-70.3457
-2.49845	-42.6742
-2.43378	-50.7948
-2.45301	-61.2119

Table4: Scores of files with model file 99_virus_N2_E3.model

SCORES OF FILES WITH N=2	
Virus Files	Normal Files
-2.47524	-37.7559

-2.46186	-73.0907
-2.47741	-57.2996
-2.46358	-47.4359
-2.4529	-43.5317
-5.56455	-48.3407
-2.45472	-46.0959
-2.45751	-74.4178
-5.55153	-43.9443
-2.42727	-93.4689
-5.57162	-57.7988
-2.45641	-87.9972
-2.46027	-190.97
-2.46253	-65.1378
-2.43879	-34.1018
-2.46136	-42.9281
-2.43016	-46.9541
-5.56507	-83.0078
-2.45594	-73.1561
-5.60275	-56.5262
-2.43937	-64.4218
-2.43402	-61.45
-2.4696	-103.48
-2.46457	-78.875

Table5: Scores of files with model file 99_virus_N2_E4.model

SCORES OF FILES WITH N=2	
Virus File	Normal File
-2.55592	-65.1906
-2.4061	-34.2304

-2.41149	**-43.0272**
-2.43066	**-47.0969**
-2.55067	**-83.1038**
-2.41977	**-73.2777**
-2.41104	**-56.6362**
-5.45948	**-64.4872**
-2.39883	**-61.5634**
-2.44326	**-103.593**
-2.43392	**-78.9375**
-2.41485	**-72.2963**
-2.45013	**-70.4344**
-2.41202	**-42.7947**
-2.41433	**-50.9159**
-2.42154	**-61.3336**
-2.42373	**-41.8691**
-2.41044	**-81.0816**
-2.42367	**-185.724**
-2.398	**-67.8557**
-2.40483	**-94.029**
-2.39655	**-52.4195**
-2.42012	**-46.5861**
-2.46457	**-201.673**

APPENDIX B: Scatter graph representation of HMM Training and Testing Results

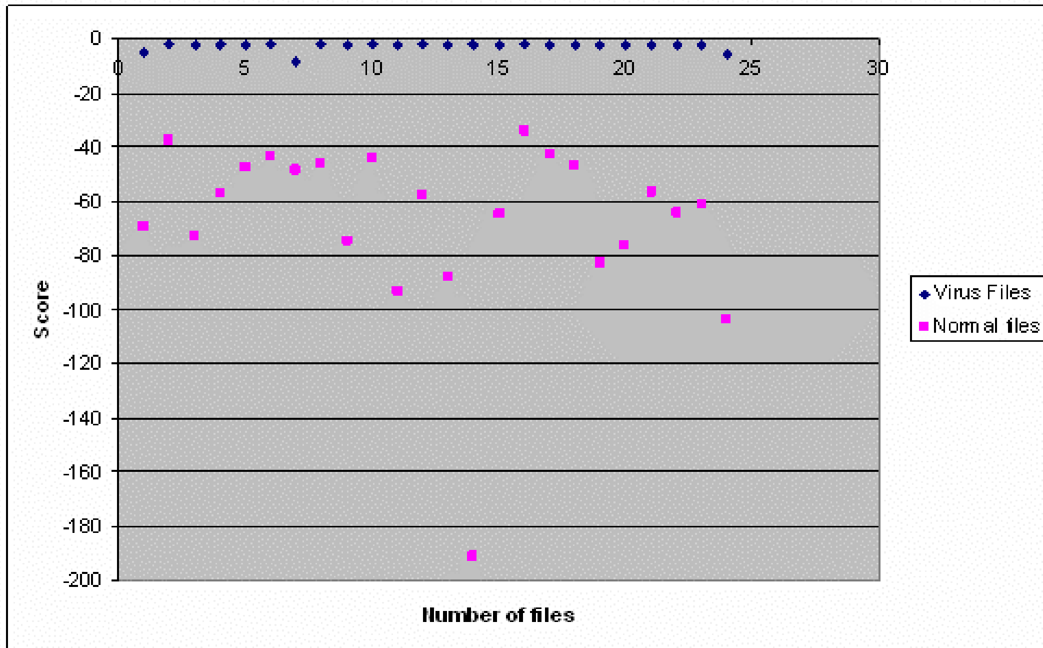
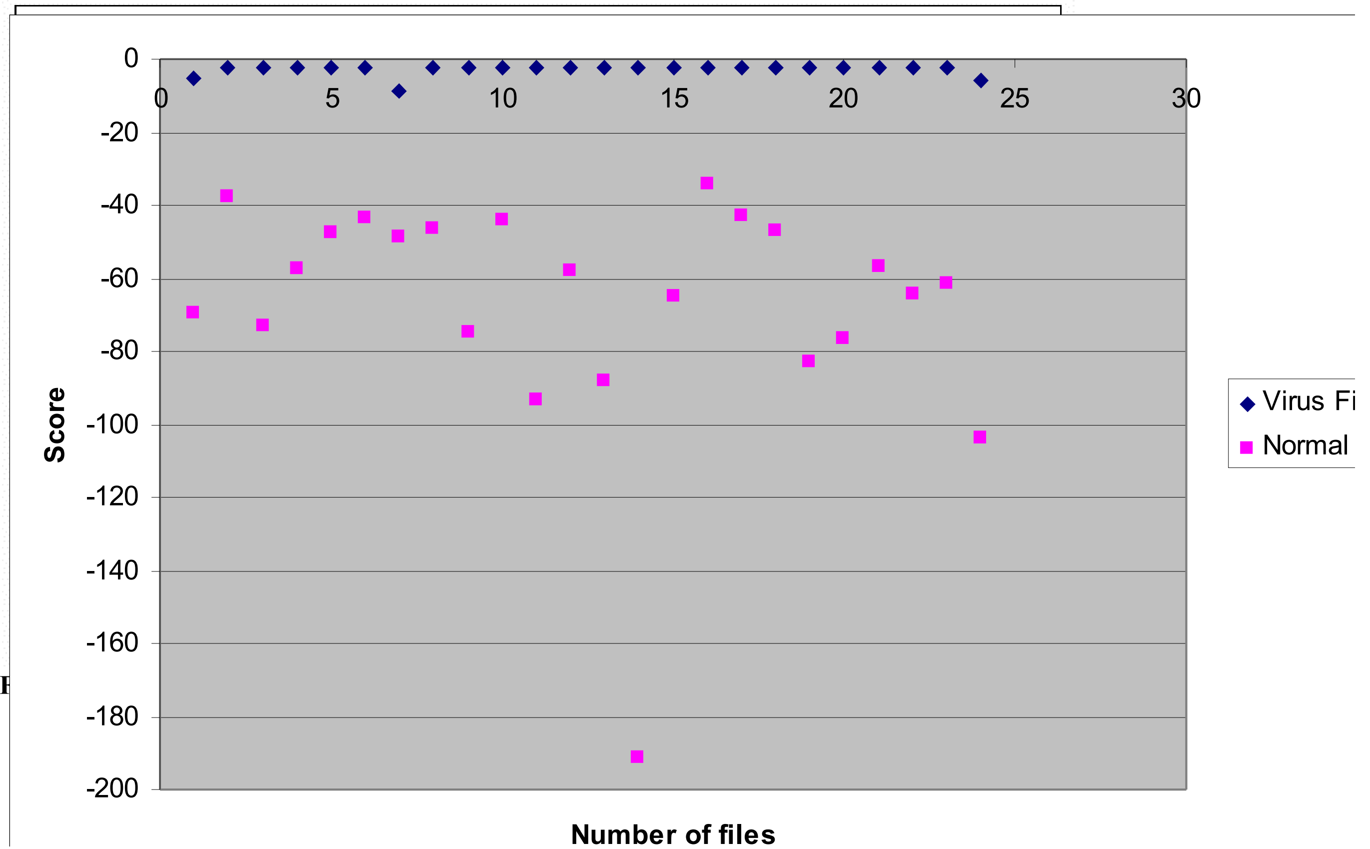


Figure 21: N = 2, E = 0



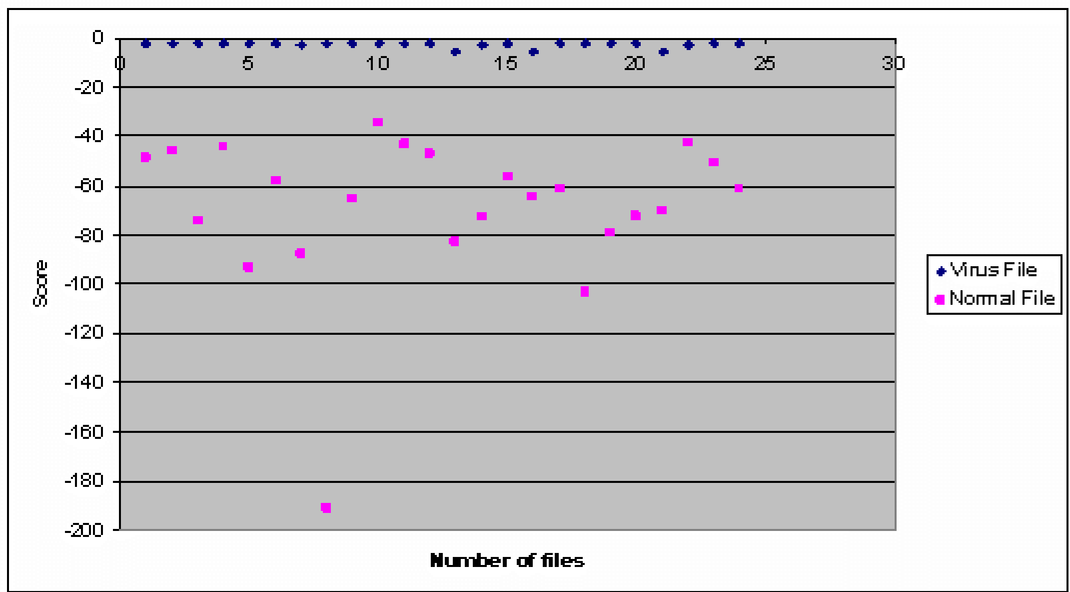


Figure 23: N =2, E = 2

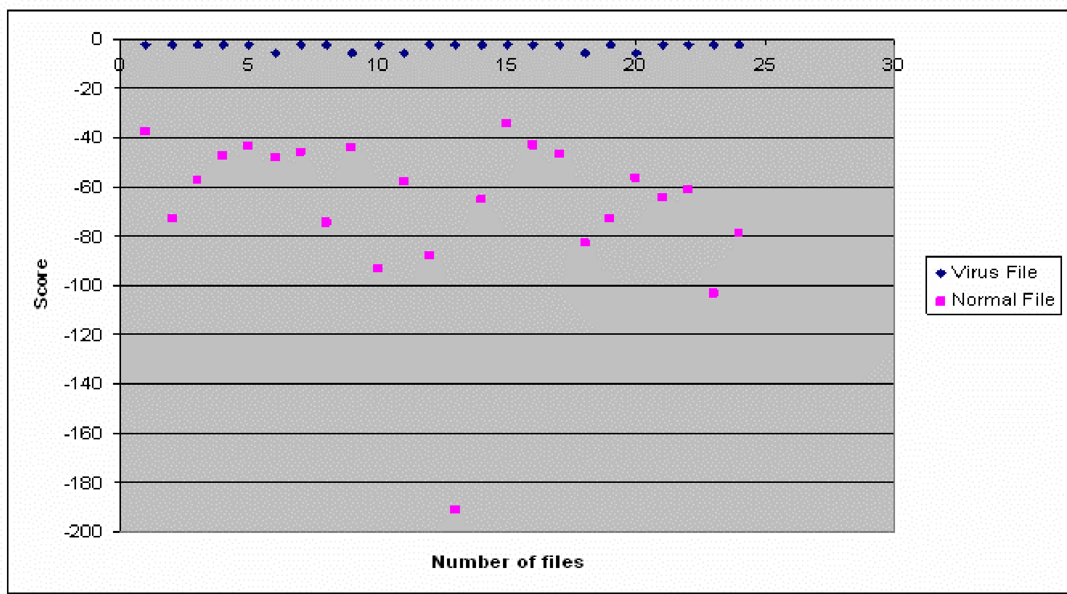


Figure 24: N =2, E = 3

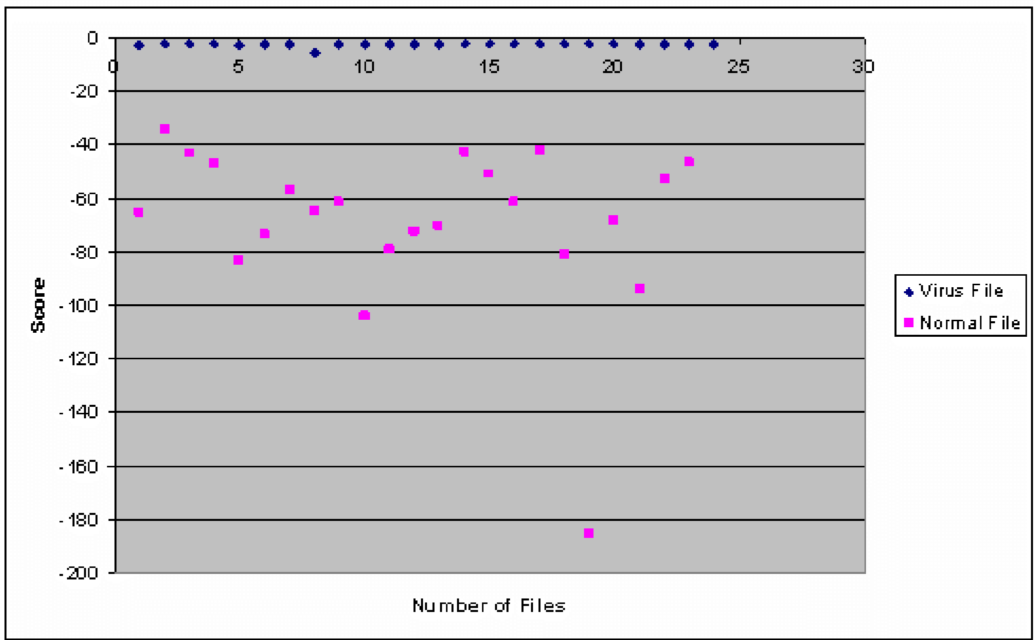


Figure 25: N =2, E = 4