

EVASION AND DETECTION OF METAMORPHIC VIRUSES

Rana Yashveer



Department of Computer Science and Engineering,

National Institute of Technology Rourkela,

Rourkela – 769008, Orissa, India.

EVASION AND DETECTION OF METAMORPHIC VIRUSES

Thesis submitted in partial fulfillment of the requirements for the degree of
Bachelor of Technology

in

Computer Science and Engineering

by

Rana Yashveer
(Roll: 108CS050)

Under the guidance of

Prof. S.K. Jena
NIT Rourkela



Department of Computer Science and Engineering,
National Institute of Technology, Rourkela
Rourkela- 769008, Orissa, India.



Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, India.

CERTIFICATE

This is to certify that the work in the thesis entitled ***Evasion and Detection of Metamorphic Viruses*** submitted by ***Rana Yashveer***(Roll No. **108CS050**) in fulfillment of the requirements for the award of Bachelor of Technology Degree in Computer Science and Engineering at NIT Rourkela is an authentic work carried out by them under my supervision and guidance. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Date: 14-05-2012

Place: Rourkela

S. K. Jena
Professor
Department of Computer Science and Engineering
National Institute of Technology Rourkela

Acknowledgment

I express my sincere gratitude to Prof. S. K. Jena for his motivation during the course of the project which served as a spur to keep the work on schedule. I also convey my heart-felt sincerity to Prof. S.Panigrahi for his constant support and timely suggestions, without which this project could not have seen the light of the day. I convey my regards to all the other faculty members of Department of Computer Science and Engineering, NIT Rourkela for their valuable guidance and advices at appropriate times. Finally, I would like to thank my friends for their help and assistance all through this project.

Rana Yashveer

Table of Contents

Chapter 1 Introduction	1
Chapter 2. Related Work	3
Chapter 3. Evolution of Virus – The Stages	4
3.1 Stealth viruses	5
3.2 Encrypted and Polymorphic viruses	5
3.3 Metamorphic Viruses	7
3.1.1 Anatomy of a Metamorphic Virus.....	8
Chapter 4. Virus Detection Strategies	10
4.1 Signature Based Detection	10
4.2 Heuristic Analysis	11
4.3 Code Emulation.....	12
Chapter 5. Code Obfuscation Techniques	14
5.1 Code Morphing Techniques.....	14
5.1.1 Dead Code Insertion	14
5.1.2 Register Exchange (Register Renaming).....	15
5.1.3 Equivalent Code Substitution.....	16
5.1.4 Transposition.....	17
5.1.5 Subroutine Permutation	17
5.1.6 Instruction Reordering via Jump Statements	18
5.1.7 Subroutine Inlining and Outlining.....	19
5.2 Anti-Heuristic Techniques	20
5.2.1 Call by API Hashing.....	20
5.2.2 Delay Routine Insertion.....	21
5.2.3 Obfuscating suspicious elements	22
Chapter 6. Project Implementation	23
6.1 Creation of Base Virus	23
6.2 Applying Metamorphic Engine	24
6.2.1 Engine Algorithm.....	24
6.3 Applying Anti-Heuristic Techniques.....	25
Chapter 7. Results	28

7.1	Base Virus	28
7.2	Base Virus with API Name Hashing.....	29
7.3	Virus with Metamorphic Code Obfuscations	29
7.4	Metamorphic Virus with Delay Routines	30
7.5	End Virus with Encrypted String Constants	31
Chapter 8. Conclusion.....		34
References		35
Appendix A: Equivalent instruction substitution [2].....		37
Appendix B: Dead code instructions [2].....		40

List of Figures

FIGURE 1 MODULES OF A COMPUTER VIRUS	5
FIGURE 2 POLYMORPHIC VIRUS PE LAYOUT.....	6
FIGURE 3 GENERATIONS OF A METAMORPHIC VIRUS.....	8
FIGURE 4 STONED VIRUS SHOWING THE SEARCH PATTERN	11
FIGURE 5 METAMORPHIC VIRUSES AND CODE OBFUSCATION TECHNIQUES.....	14
FIGURE 6 DEAD CODE INSERTION IN EVOL VIRUS	15
FIGURE 7 TWO DIFFERENT GENERATIONS OF REGSWAP.....	16
FIGURE 8 INSTRUCTION SUBSTITUTION IN METAPHOR VIRUS.....	16
FIGURE 9 SUBROUTINE PERMUTATION.....	18
FIGURE 10 CODE REORDERING THROUGH JUMP STATEMENTS	19
FIGURE 11 SUBROUTINE INLINING	19
FIGURE 12 SUBROUTINE OUTLINING.....	20
FIGURE 13 IMPORTED API FUNCTIONS IN IPMSG.EXE.....	21
FIGURE 14 NGVCK USER INTERFACE.....	24
FIGURE 15 : PROCESS FLOW DIAGRAM	26
FIGURE 16 SCAN RESULTS FOR BASE VIRUS	28
FIGURE 17 SCAN RESULTS FOR BASE VIRUS WITH API NAME HASHING	29
FIGURE 18 SCAN RESULTS FOR VIRUS WITH METAMORPHISM	30
FIGURE 19 SCAN RESULTS FOR METAMORPHIC VIRUS WITH DELAY ROUTINES.....	31
FIGURE 20 SCAN RESULTS FOR END VIRUS.....	32
FIGURE 21 DETECTION RATE OF DIFFERENT VIRUS SAMPLES	33
FIGURE 22 FEATURE COMPARISON OF ANTI-VIRUS ENGINES	33

List of Tables

TABLE 1 OVERVIEW OF DETECTION METHODS	13
TABLE 2 SUMMARY OF TOOLS USED	27
TABLE 3 EQUIVALENT INSTRUCTION SUBSTITUTIONS	39
TABLE 4 DEAD CODE INSTRUCTIONS	41

Abstract

Metamorphic viruses mutate their own code to produce viral copies which are syntactically different from their parents, but functionally equivalent. The viral copies thus produced, may have different signatures, rendering signature-based virus scanners unreliable. New age anti-virus products employ a combination of signature scanning and heuristic techniques to defeat such viruses.

In this project, a metamorphic engine, which uses code obfuscation techniques, is implemented to bypass commercial scanners. A set of anti-heuristic strategies are used to evade code emulation and heuristic detection. Using a combination of the above techniques, the detection rate of a well known sample virus is reduced significantly. Finally, a brief comparative study of major commercial anti-virus software is performed with respect to their detection capability.

Chapter 1 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 existed 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.

In general, a virus performs activities without permission of users. Certain viruses can perform damaging activities on a host machine, such as hard disk data corruption or crashing the computer. Other viruses are harmless and might, as an instance, print annoying messages on the screen. In any case, viruses are undesirable for users, regardless of their nature [4]. Modern viruses also take advantage of the always-connected Internet to spread on a global level. Therefore, early detection of viruses is necessary to minimize damage.

There are many antivirus defense mechanisms available today, but chief among these is signature detection, which involves looking for a fingerprint-like sequence of bits (extracted from a known sample of the virus) in the suspect file [9]. Metamorphic viruses are quite potent against this technique since they can use a variety of code morphing techniques to change the structure of the viral code without altering its function.

A heuristic anti-virus program examines a target program (executable file, boot record, or possibly document file with a macro) and analyzes its program code to determine if the code

appears virus-like. Since this technique does not depend on virus signatures it can detect new and unknown viruses that have not yet been analyzed by antivirus researchers. Modern age anti-virus products incorporate a combination of all these techniques to defeat virus writers.

This project constituted the implementation of a metamorphic engine, employing various code obfuscation techniques, and using this engine on a well known virus sample to bypass basic detection mechanisms. Also, a set of anti-heuristic strategies were included to negate heuristic detection via code emulation. Based on the above results, the reliability and effectiveness of modern day commercial anti-virus programs were briefly discussed.

Chapter 2. Related Work

Obfuscation is a common term referring to any method that capable of transforming the original program code into an unreadable or misleading version with the intention of concealing the true purpose of code from interpretation by a human being, or any detection program. Wong W. [9] analyzed several metamorphic virus generator kits by defining a similarity index and using it to precisely quantify the degree of metamorphism produced by each generator. He then presented a detector based on the principle of statistical hidden Markov models. U.Mishra [15] presented a brief introduction to various scanning methods employed today, their strengths and their corresponding limitations, suggesting modifications and improvements that could be made on existing detection techniques. Babak R., Maslim M. and Suhaimi I. [14] also surveyed the most common scanning and detection methods used in modern day anti-virus software, drawing a feature comparison and suggesting modifications. Our research comprises of analyzing the robustness of a handful of anti-virus engines against the ubiquitous code obfuscation techniques by using a well known virus as a sample.

Chapter 3. Evolution of Virus – The Stages

.A computer virus is a program designed or sequence of instructions written to infect and potentially damage files on a target system. The term "virus" is also commonly, but erroneously, used to refer to other types of malware, that do not have a reproductive ability. For replication and spreading, viruses need to have authorization to read/write to memory. Hence, a lot of viruses attach themselves to legitimate executable files. When such infected programs are run, the attached piece of virus code also gets executed, thereby damaging the system [6]. Modern viruses utilize the Internet to spread on a wider domain.

The virus evolution phenomenon is believed to have started with an academic project done by Fred Cohen (1983), following which Len Andleman coined the term “virus” [1]. Cohen, widely considered as the “Father of computer viruses”, proved that it was impossible to detect all variations of a virus program. The Creeper Virus written by Bob Thomas in 1971, was the first successful virus, capable propagation through ARPANET [13].

Generally a computer virus consists of the following modules:

- infect() defines the mechanism of virus replication
- trigger() is a conditional test that decides whether to execute the payload or not.
- payload() defines actual damage causing instructions existing in the virus.

```
def virus() :  
    infect ()  
    if trigger () is true then  
        payload ()
```

Figure 1 Modules of a computer virus [1]

3.1 Stealth viruses

Virus writers have been crafting techniques to avoid detection from the frontier days of computer viruses. One of the first elementary techniques was restore the last modified date of an infected file to make it appear untouched. The detectors responded by maintaining cyclic redundancy check (CRC) logs on files that would indicate any sort of code modification or infection. Others, such as 'Brain', tried to hide in memory and maintained different copies of infected files, occupying system functions for reading disk sectors and redirecting anti-virus programs to the unaffected copies to bypass malicious flagging.

3.2 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. Encrypted viruses typically carry along a decryption engine and thus they have to maintain a small portion of the virus body unencrypted.

Antivirus programs started to identify such viruses by looking for the signature bits in this unencrypted portion. [6]

Oligomorphic viruses took the stage, where the viruses employed multiple decryption algorithms (carry multiple decryption engines and pick randomly) making pattern based detection virtually impossible.

Subsequently, polymorphic viruses entered the scene, which were encrypted viruses with the ability to mutate their decryption engines in each generation. These operate with the assistance of an encryption engine which changes with each virus replication; this keeps the encrypted virus functional, while still hiding the polymorphic virus from the computer it infects. Polymorphic viruses can generate many unique decryptors and can use many other encryption methods for encryption. This feature helps bypass common signature detection techniques.[9]

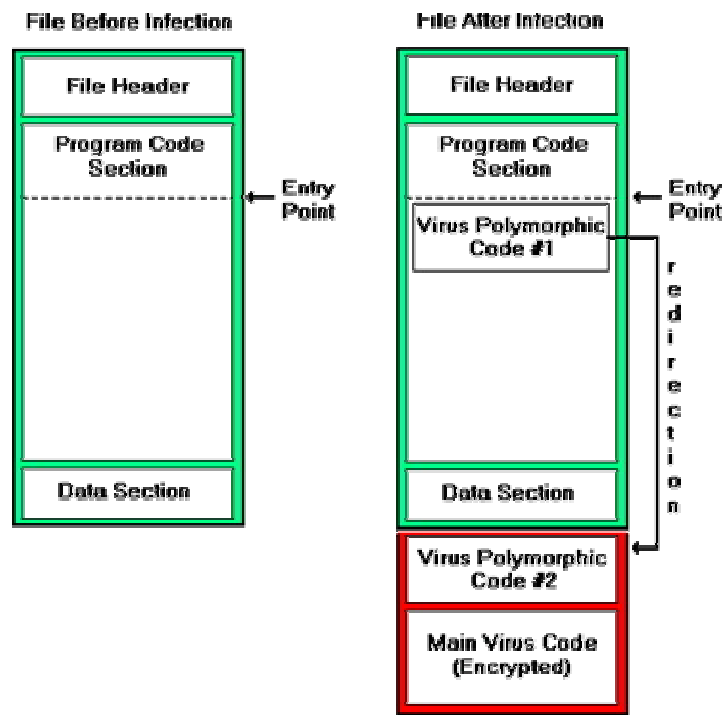


Figure 2 Polymorphic Virus PE Layout

Polymorphic viruses required modifications in anti-virus technology and the problem gave birth to static emulation. In this method, the virus decryption process is executed in a controlled environment to capture the location of the decrypted virus. Here, detector can scan for a signature string in the decrypted virus and use that to detect further infections of the same virus.

3.3 Metamorphic Viruses

Metamorphic viruses modify their code to produce an equivalent one during propagation. They are a step ahead than polymorphic viruses, since the latter keeps the virus body constant in each generation. Such viruses attempt to avoid creating alarm through static analysis by implementing code obfuscation techniques. Techniques like are swapping of interchangeable instructions, inserting junk instructions and introducing conditional/unconditional jumps to produce the child virus. The child virus possesses the same functionality but a different pattern signature. In this method, the signature of a virus is broken by changing the order of instructions without altering the control flow. Metamorphic code can also mean that a virus is capable of infecting executables from several operating systems (Windows or GNU/Linux) or even multiple computer architectures. A sophisticated virus type will generate code based on the host's operating system by translating the existing instructions to the corresponding machine code.

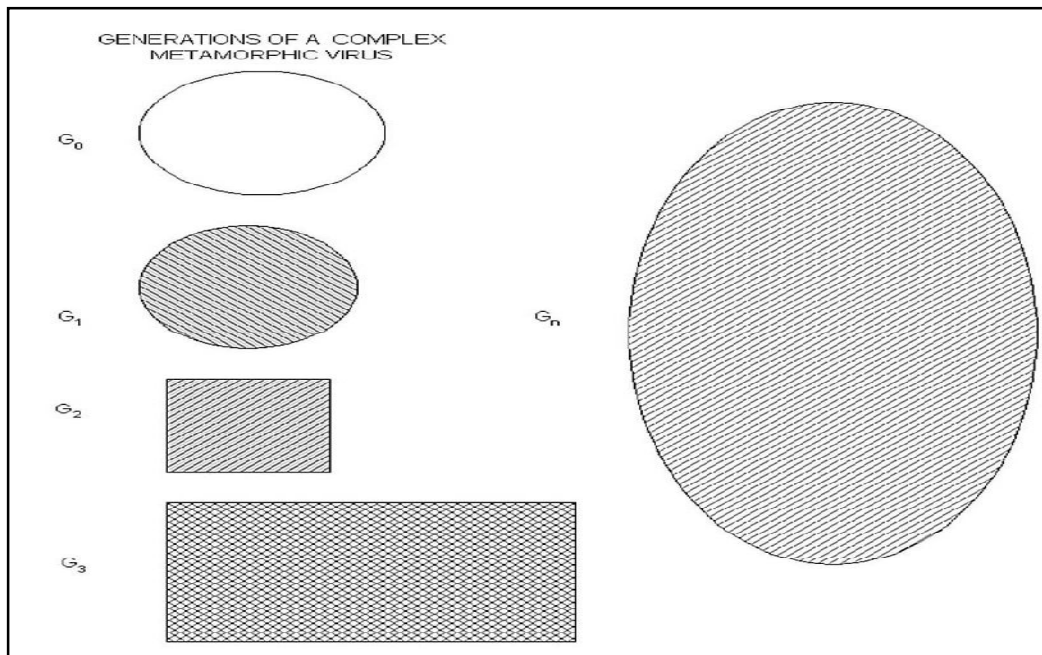


Figure 3 Generations of a metamorphic virus [6]

3.1.1 Anatomy of a Metamorphic Virus

A metamorphic virus has the metamorphic engine embedded within itself. Zperm was found to carry along its own metamorphic engine, known as the Real Permuting Engine (RPME)[3]. During infection, a metamorphic virus creates several morphed copies of itself using this embedded engine. A typical metamorphic engine is expected to contain [11]:

1. Internal disassembler
2. Opcode shrinker
3. Opcode expander
4. Opcode swapper
5. Relocator/recalculator
6. Garbager and Cleaner

Internal disassembler disassembles the binary / executable code, per instruction. Opcode shrinker optimizes the program instructions. Opcode shrinker replaces two or more instructions with a set of equivalent instructions. Opcode expander performs the reverse operation of opcode shrinker. It replaces one instruction with several instructions.

Opcode swapper changes the order of the instructions. Generally it swaps two unrelated instructions. Relocator relocates relative references like jump and call. Garbager inserts do-nothing instructions. Cleaner undoes Garbager, i.e. it removes do-nothing instructions inserted by Garbager.

Characteristics of an effective metamorphic engine are [11]:

1. A metamorphic engine should be familiar with any opcode of an assembly language. An engine should know all of the opcodes of the targeted system architecture.
2. Opcode shrinker and swapper should be able to process more than one instruction concurrently.
3. Garbager is used in moderate amount.
4. Garbage should not affect actual instructions.
5. Opcode swapper should analyze each instruction, swapping unrelated instructions and should not affect the execution of next instruction.

The metamorphic engine used in the project is implemented as a tool separate from the seed virus. This tool reads in an assembly program generated by virus toolkits or disassembled virus executable and performs the metamorphosis.

Chapter 4. Virus Detection Strategies

This section provides an overview of all major detection methods employed by modern day antivirus software. The objective of using different methods is 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.

4.1 Signature Based Detection

The most popular technique in anti-virus scanners today is pattern matching. It is not as effective as some other techniques but it is the fastest. This technique involves extraction of a unique sequence of bits from a known virus and using this sample as a fingerprint to subsequently match against while scanning for the existence of the virus. Statistical techniques are also used to extract these patterns. [6]

This method of detection is fast and fairly accurate since the chances of false alarms are very low in this system. The main drawback of the system is the heavy dependence on an updated database of all the signature files of malware. The accuracy is totally determined by the signature database of the system. Signature based detection systems fail to detect a new virus since the database do not contain any information about the new virus. Figure 4.1 shows an illustration of a search pattern for the ‘Stoned’ boot sector virus. Here, the sequence of bits selected was chosen by observing a unique behavioral peculiarity of the virus (it read 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                                     ; CODE XREF: sub_7C3A+27↓j
seg000:7C43                                     ; read one sector
seg000:7C43 next:      mov     ax, 201h
seg000:7C43                                ; read one sector
seg000:7C46 0E                push   cs
seg000:7C47 07                pop     es
seg000:7C48                                assume es:seg000
seg000:7C48 8B 00 02          mov     dx, 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 4 Stoned virus showing the search pattern 0400 B801 020E 07BB 0002 33C9 8BD1 419C [6]

4.2 Heuristic Analysis

Heuristic analysis is suitable for detecting unknown or ‘disguised’ viruses. Heuristic analysis may be static or dynamic. The first heuristic engines were introduced to detect DOS viruses in 1989. Static heuristics analyze the file format and the code structure to look for suspicious characteristics of a virus body, while dynamic heuristics utilize code emulators designed to detect viral code. Heuristic analysis is done in two stages [5] – Data Gathering in which the data is collected using many heuristics and Analysis in which the techniques like data mining, expert systems or neural networks can be for virus sample analysis They do this by employing either weight-based systems and/or rule-based systems.

Depending on the environment and the technological level, the following components can be found within heuristic engines [5]:

- Variable/memory emulator
- Parser
- Flow analyzer
- Analyzer

- Disassembler/emulator
- Weight-based system and/or Rule based system.

The following are some of the suspicious characteristics defined as rules for heuristic engines, indicating a possible 32-bit PE (Portable Executable) virus [6]

- Code execution starts in the last section
- Incorrect virtual size in PE header
- Unnecessary 'Gaps' between sections
- Suspicious altered code section name
- Suspicious API imports from Kernel32.dll, (importing by ordinal instead of importing by name)

One shortcoming of heuristic analysis is that it can create many false positives. Even though chances of false alarm are relatively higher, it has a better chance of detecting new viruses. The critical issue is that raising a false alarm is not as potentially harmful as tagging a new virus positive. However, such systems can be trained gradually by intruders to consider abnormal behavior as routine. Thus, system might fail to detect the abnormal activity in such cases

4.3 Code Emulation

Code emulation is a detection technique in which a virus is executed in a simulated environment without actually impacting the original host machine. A virtual machine is implemented to simulate the CPU and memory management systems to mimic the code execution. This is a dynamic analysis method as the code of the virus is run in real time to observe its behavior. A good dynamic code emulator comprises of five functionalities [1], which are CPU emulation, Memory emulation, Operating System and Hardware emulation, Emulation controller and Analyzer. It is imperative to define memory access functions to fetch 8-bit, 16-bit,

and 32-bit data (and so on). Further, the functionality of the operating system should be emulated to create a virtualized system that will support system APIs, file and memory management. This technique is highly potent against polymorphic encrypted viruses, since the decryption routine can be emulated to locate the unencrypted plaintext code on which pattern matching can be performed. Table 4.1 gives an overview of the various detection techniques.

Detection Technique	Strength	Weakness
Signature based	Fast, efficient, accurate	New malware
Heuristic Analysis	New malware	Implementation cost, False positives
Emulation Based	Encrypted viruses	Costly to implement

Table 1 Overview of Detection Methods

Chapter 5. Code Obfuscation Techniques

5.1 Code Morphing Techniques

Metamorphic engines use various code morphing techniques to generate morphed copies of the original program. Generally, the morphed code is more difficult to read and understand than the original, due to a higher complexity of instructions used. Code morphing can be used to generate a large number of distinct copies of a parent file. This section describes some morphing techniques that are applied to assembly code. Code morphing techniques for assembly programs can apply to the control flow, code, or data (Borello and Me, 2008). Control flow obfuscation involves instruction reordering, typically through insertion of jumps, or calls. Figure 5.1 provides an overview of some well-known metamorphic viruses and their code obfuscation techniques.

	EVOL (2000)	ZMIST (2001)	ZPERM (2000)	REGSWAP (2000)	METAPLOR (2001)
Instruction Substitution				✓	
Instruction Permutation	✓	✓			✓
Dead code Insertion	✓	✓			✓
Variable Substitution	✓	✓		✓	✓
Changing the Control Flow		✓	✓		✓

Figure 5 Metamorphic viruses and code obfuscation techniques

5.1.1 Dead Code Insertion

Inserting dead code or do-nothing instruction does not affect the code execution. Dead code can be a single instruction or an instruction block. Inserting dead code changes the appearance of a program by altering its binary pattern. Adding different block sizes of dead code

on each generation creates different looking programs with the same functionality. However, such insertions cause swelling of the size of the original program. Hence, dead codes should not be used excessively. The Evol virus implemented dead code insertion by adding a block of dead code between core instructions as shown.

		BF0F00055	mov edi, 5500000Fh
		893E	mov [esi], edi
		5F	pop edi ; garbage
C7060F000055	mov [esi], 5500000Fh	52	push edx ; garbage
C746048BEC5151	mov [esi+0004], 5151EC8Bh	B640	mov dh, 40 ; garbage
		BA8BEC5151	mov edx, 5151EC8Bh
		53	push ebx ; garbage
		8BDA	mov ebx, edx
		895E04	mov [esi+0004], ebx

Figure 6 Dead Code Insertion in Evol Virus [16]

5.1.2 Register Exchange (Register Renaming)

Register renaming substitutes register operands of an instruction without changing the instruction itself. The instructions remain constant across all morphed copies. Since only the operands change, it alters the binary signature. RegSwap was one of the earliest metamorphic viruses to employ register usage exchange. The underlying principle is to try change the operational code pattern and bypass the signature detection Figure 5.3 shows two pieces of code from two different copies of RegSwap.


```

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]
89E4BA18110000 mov   [edx+edi*4+00001118],esi

```

Figure 7 Two different generations of RegSwap [6]

5.1.3 Equivalent Code Substitution

Equivalent code substitution is the replacement of an instruction with an equivalent instruction or a similar block of instructions. In assembly language, generally a single task can be achieved in different ways. This method is highly successful in defeating signature detection systems because it totally detects the viruses based on the opcode pattern. The obfuscation introduced through this method, though effective is not permanent. These obfuscations are removed if the executable is made to go through a cycle of assembly/disassembly processes. The W32/MetaPhor virus is one of the metamorphic virus generators that includes the instruction substitution technique.

Single Instruction	Instruction block
XOR Reg,Reg	MOV Reg,0
MOV Reg,Imm	PUSH Imm POP Reg
OP Reg,Reg2	MOV Mem,Reg OP Mem,Reg2 MOV Reg,Mem

Figure 8 Instruction Substitution in MetaPhor Virus [7]

5.1.4 Transposition

Transposition or instruction permutation changes the instruction execution order in a program. This can be done only if no relation exists among instructions. Consider two instructions Instruction-1 (OP1 R1, R2) and Instruction-2 (OP2 R3, R4). These two instructions can be transposed if following conditions are met:

1. $R1 \neq R3$
2. $R1 \neq R4$
3. $R2 \neq R3$

However, this technique should be used very carefully since it results in program corruption.

5.1.5 Subroutine Permutation

This is a basic obfuscation technique in which the subroutines of a program are reordered. A program with n different subroutines can generate $(n-1)!$ different subroutine permutations. Subroutine permutation does not affect the functionality of a program nor the program execution flow as the order of subroutine is not important for its execution.

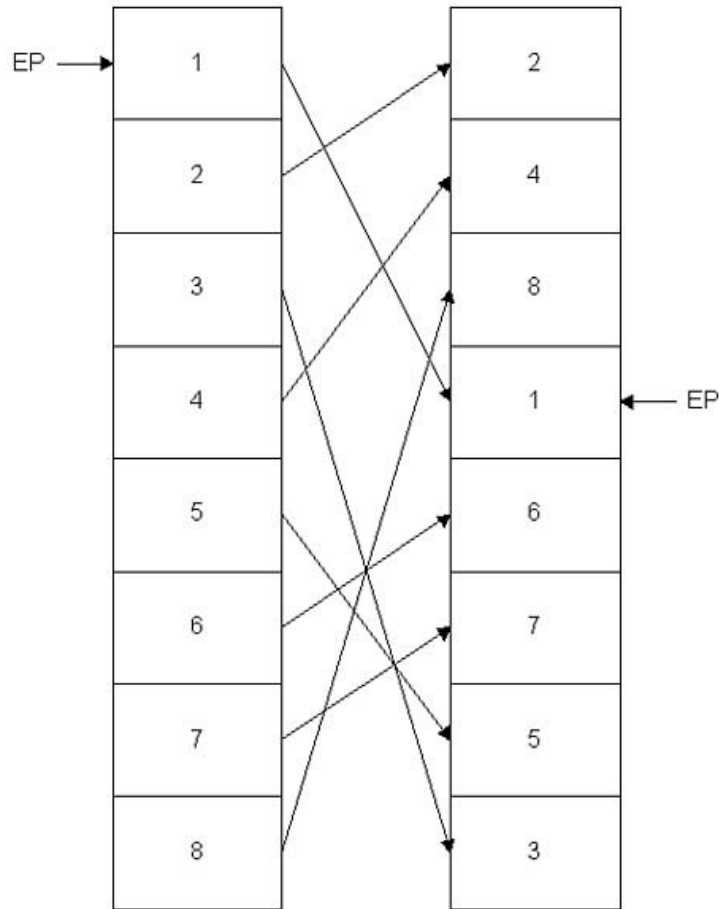


Figure 9 Subroutine Permutation

5.1.6 Instruction Reordering via Jump Statements

Code reordering inserts conditional or unconditional branching instruction after every single instruction or a block of instructions. These blocks defined by the branching instructions are permuted and shuffled to change the control flow. The modified code is termed as Spaghetti Code. The conditional branching instruction is generally preceded by a test instruction to force the execution of the branching instruction. However, this technique can be rendered useless by Control Flow Graph (CFG) Analysis, if the jump test instructions are not shrewdly implemented.

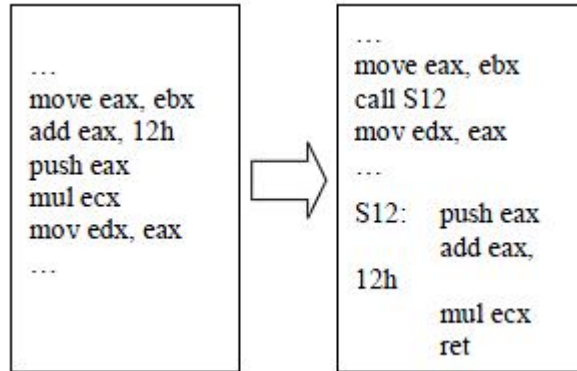


Figure 12 Subroutine Outlining

Out of the above listed, three techniques, namely Dead Code Insertion, Instruction Substitution and Transposition were implemented in our metamorphic engine.

5.2 Anti-Heuristic Techniques

Anti-heuristic techniques are efforts by virus writers to avoid their code being detected as a possible new virus by heuristic detection. Most of these techniques are developed by carefully studying the logistics of heuristic analysis and appending modifications to bypass those rules.

5.2.1 Call by API Hashing

A simple call to win32 API functions causes the imported function to be listed in the Import Table of the executable and the PE Export Table of the loaded modules. Figure 5.9 shows the PE Import table of a well known messaging application IP Messenger.

Function	Address	95-2 B	NT4 SP1	NT4 SP3	NT4 SP4	NT4 SP6	98 A	ME	2k SP1	2k SP2	2k SP3	2k SP4	XP SP1	XP SP2	2k3/XP64..
ADVAPI32.dll (11)															
RegDeleteKeyA	000221D8	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegEnumValueA	0002220A	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegEnumKeyExA	000221FA	+	+	+	+	+	+	+	+	+	+	+	+	+	+
GetUserNameA	00022160	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegOpenKeyExA	00022170	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegCreateKeyExA	00022180	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegCloseKey	00022192	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegQueryValueA	000221A0	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegQueryValueExA	000221B2	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegSetValueExA	000221C6	+	+	+	+	+	+	+	+	+	+	+	+	+	+
RegDeleteValueA	000221E8	+	+	+	+	+	+	+	+	+	+	+	+	+	+

Figure 13 Imported API Functions in ipmsg.exe

It is possible to obfuscate such calls to API functions by hashing API names. A typical algorithm is to add each ASCII character of an API function name to a 32-bit value, performing a bitwise rotation right 13 places for each character. This produces a hash with no collisions in any major system DLLs, making it an easy and safe method of obfuscation. CRC32 hashes are generally used for this purpose. For example, the hash value of string 'URLDownloadToFileA' [10] can be used as input parameter to a subroutine which retrieves the address of the function from the loaded Dynamic Link Library (DLL) files

call URLDownloadToFileA → push 702F1A36h ; hash value
call Get_API ; procedure to retrieve API

5.2.2 Delay Routine Insertion

The idea is that these heuristic scanners only emulate the first set of instructions and then stop to speed up scanning, since spending significant time on a single file is not feasible to their application. This property can be exploited by introducing endless loops or loops with very large counter variables. A classic form is shown below:

```
mov cx, 0FFFFh ; Counter variable
```

loop_head:

```
jmp short over_it
```

```
mov ax, 4C00h
```

```
int 21h .
```

over_it:

```
loop loop_head
```

It can also be achieved by calling the Win32 Sleep API function and setting the sleep parameter to an order of 10 seconds. The success of this technique depends on the configuration of the Anti-virus engine and the speed of the processor on which emulation is carried out.

5.2.3 Obfuscating suspicious elements

The Next Generation Virus Creation Kit (NGVCK) virus used plain strings to store the extension of executable files to infect. Heuristic analysis involves searching for such suspicious elements and raising the alarm if it is encountered. A solution to avoid setting the heuristic flags is to store encrypted form of such string elements, to be retrieved later by a decryption routine.

```
Filemask db '*.Exe', 0          XOR 77h  
                                ⇒ filemask db 5Dh, 59h, 32h, 0Fh, 12h, 0
```

Chapter 6. Project Implementation

6.1 Creation of Base Virus

1. The base virus, which was given as the input to the code obfuscation engine, was created using a virus construction kit, NGVCK (Next Generation Virus Creation Kit), obtained from VX Heavens website [12].
2. The virus constructor had specific instructions and options to create a seed virus. Seed viruses were created following the instructions given by the virus construction kits. The following features were included in the test sample
 - Upward directory traversal for file infection
 - Max file infection count = 20
 - API Search Type – CRC32 Hashing
 - Entry Point Obscurity (EPO) disabled
3. Resultant output assembly file was then compiled using TASM 5.0 using options suggested by the program file.
4. The compiled virus executable was uploaded to Jotti's malware scanner website [8] and scanned across multiple antivirus engines updated with the latest virus signature database.

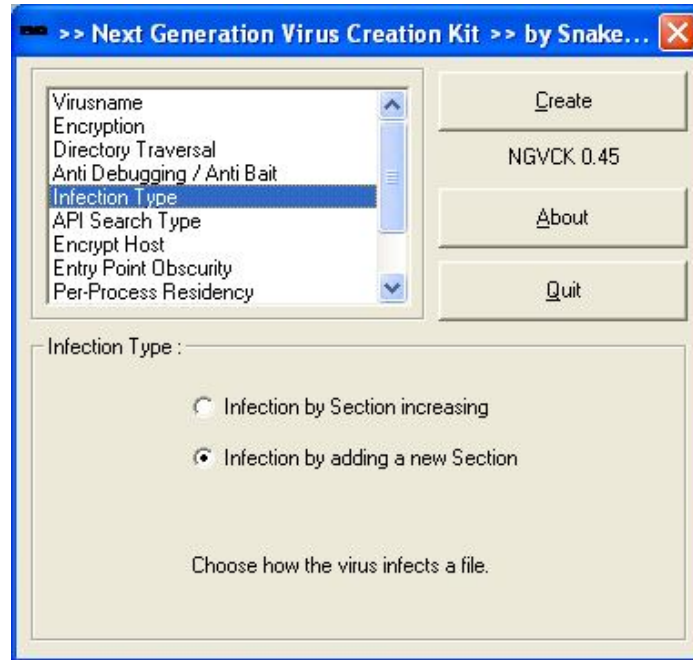


Figure 14 NGVCK User Interface

6.2 Applying Metamorphic Engine

A metamorphic engine was implemented to introduce code obfuscations in the original virus. Each of the obfuscation technique was designed as a separate module, and had own process to decide when and how to apply the techniques. The engine was implemented using Java SE, organized into separate class files for each of the instructions supported for obfuscation.

6.2.1 Engine Algorithm

The metamorphic engine follows a general algorithm to generate metamorphic copies of the base virus file. The high level metamorphic algorithm is summarized as below:

1. Determine the start of code section.

For every instruction matching supported instruction list

2. **RAND_NUM_SUB** = random number from 0 to 2

3. If **RAND_NUM_SUB** \leq 1 then select the instruction for Substitution // substitution is done for about 2 in 3 instructions.

4. Substitution:

a. **RAND_JUNK_EQUI** = random number from 0 to 2.

b. If (**RAND_JUNK_EQUI** $<$ 2) //equivalent code substitution is done 66%

i. Perform equivalent code substitution

c. Else

i. Perform junk code insertion

//randomly select among Single NOP instruction insertion

//jump NOP, and Evol transformations.

5. Repeat steps 2 to 4 till end of the file.

6. Perform **transpose** on the generated morphed code.

The assembly source file of the seed virus created earlier was given as input to the metamorphic engine. The engine, using multi threading, was configured to create 10 metamorphic copies of the source program.

6.3 Applying Anti-Heuristic Techniques

The anti-heuristic techniques, discussed earlier, were applied on the resultant metamorphic virus copies. Delay routines were inserted randomly at different locations of the ‘.code’ section.

NGVCK stored the file extension wildcard as a plain string '*.Exe', which was encrypted by XOR-ing the ASCII value of each character with the 8-bit value 77h, and storing the resultant 8-bit value. This string was retrieved as and when required, by writing a simple decryption routine that involved XOR-ing the encrypted byte value with the same key (77h). The basic decryption routine algorithm was as follows:

1. Save all register values (pushad instruction)
2. Load the offset (from the .data section) of required string in the register
3. Execute the instruction
$$[\text{Reg}] = [\text{Reg}] \text{ XOR } 77\text{h}$$
4. Increment register to move to next byte
5. Repeat Steps 3 and 4 till null character is reached
6. Restore all register values (popad instruction)

At the end of each step, the resultant virus was compiled (using Borland's Turbo Assembler TASM) and the PE file was uploaded to Jotti's malware scanner [8] for detection purposes.

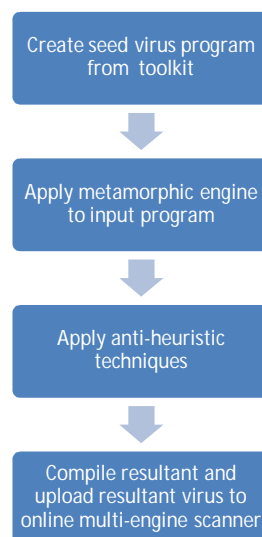


Figure 15 : Process Flow Diagram

A brief summary of the tools utilized during the process are given in Table 6.1

Experiment Platform OS	Windows XP SP3 VMWare Workstation 7.0
Meta Engine Programming Language	Java 2 SE
Assembler	Borland Turbo Assembler 5.0(TASM)
PE Analyzer	Safer Networking File Analyzer 2.0.5
Virus Generator	Next Generation Virus Creation Kit (NGVCK)
Virus Scanners	Multi Engine AV Scanner (Jotti's Malware Scan)

Table 2 Summary of Tools Used

Chapter 7. Results

Jotti's Online Scanner was used to obtain detection results simultaneously from 20 different popular Antivirus engines. The scan gave details regarding the malware type and fingerprint, if detected as malicious. The resultant executables from each stage of the process were uploaded and the results were evaluated.

7.1 Base Virus

The original virus created from a toolkit was, expectedly, detected by a number of antivirus engines, flagging it as a 'Generic Win32.FileInfector'.

Detections – 7/20 (35%)

Jotti's malware scan		Additional info	
Filename:	xyz1.EXE	File size:	8192 bytes
Status:	Scan finished. 7 out of 20 scanners reported malware.	Filetype:	PE32 executable for MS Windows (console) Intel 80386 32-bit
Scan taken on:	Wed 2 May 2012 07:01:02 (CET) Permalink	MD5:	ccc53897d38939e6fa57098314ac4042
	Next file	SHA1:	804f263aa224133eaac6b5a5396ac4e0c0377e37
		Packer (Kaspersky):	PE_Patch

Scanners	
ArcaVir	2012-05-02 Found nothing
avast!	2012-05-01 Found nothing
AVG	2012-05-01 Found nothing
AntiVir	2012-05-02 HEUR/Malware
bitdefender	2012-05-02 Gen:Win32.FileInfector.aqX@au16Zse
Clam AV	2012-05-01 Found nothing
CP engine	2012-05-02 Found nothing
Dr.WEB	2012-05-02 Found nothing
Emsisoft	2012-05-02 Found nothing
eSet	2012-05-01 unknown WIN32
F-PROT	2012-05-01 W32/Parasitic-Fileinfector-based!Maximus
F-Secure	2012-05-02 Gen:Win32.FileInfector.aqX@au16Zse
GDATA	2012-05-02 Gen:Win32.FileInfector.aqX@au16Zse
IKARUS	2012-05-02 Found nothing
KASPERSKY	2012-05-02 Found nothing
PANDA	2012-05-01 Found nothing
Quick Heal	2012-04-30 Found nothing
SOPHOS	2012-05-02 Found nothing
VBA32	2012-04-30 Unknown.Win32Virus
VirusBuster	2012-05-01 Found nothing

Figure 16 Scan Results for Base Virus

7.2 Base Virus with API Name Hashing

The 'CRC32' option was used for the searching of API Names in the seed virus using the toolkit. The F-PROT antivirus, earlier detecting as 'W32/Parasitic-Fileinfector-based!Maximus', now showed the file as clean.

Detections – 6/20 (30%)

Jotti's malware scan		Additional info	
Filename:	xyz.EXE	File size:	8192 bytes
Status:	Scan finished. 6 out of 20 scanners reported malware.	Filetype:	PE32 executable for MS Windows (console) Intel 80386 32-bit
Scan taken on:	Wed 2 May 2012 06:53:07 (CET) Permalink	MD5:	7b36b78f35dfeb9935c3ab67542c5128
Next file		SHA1:	5e4612419874e92d556978014d97a4bfc65ba8f7
		Packer (Kaspersky):	PE_Patch

Scanners	
ArcaVir	2012-05-02 Found nothing
AVAST!	2012-05-01 Found nothing
AVG	2012-05-01 Found nothing
AntiVir	2012-05-02 HEUR/Malware
bitdefender	2012-05-02 Gen:Win32.FileInfector.aqX@aa3ydsj
ClamAV	2012-05-01 Found nothing
CP engine	2012-05-02 Found nothing
Dr.WEB	2012-05-02 Found nothing
Emsisoft	2012-05-02 Found nothing
eset	2012-05-01 unknown NewHeur_PE
F-PROT	2012-05-01 Found nothing
F-Secure	2012-05-02 Gen:Win32.FileInfector.aqX@aa3ydsj
G DATA	2012-05-02 Gen:Win32.FileInfector.aqX@aa3ydsj
IKARUS	2012-05-02 Found nothing
KASPERSKY	2012-05-02 Found nothing
PANDA	2012-05-01 Found nothing
Quick Heal	2012-04-30 Found nothing
SOPHOS	2012-05-02 Found nothing
VBA32	2012-04-30 Unknown.Win32Virus
VirusBuster	2012-05-01 Found nothing

Figure 17 Scan Results for Base Virus with API Name Hashing

7.3 Virus with Metamorphic Code Obfuscations

The base virus, with integrated call to API functions via name hashing, was given as input to the metamorphic engine to generate highly obfuscated copy of the virus, which was then compiled to a portable executable (PE) and scanned. VBA32, BitDefender, F-Secure and G-DATA dropped the malicious flagging of the virus.

Detections - 3/20 (15%)

Jotti's malware scan		Additional info	
Filename:	idan0.EXE	File size:	8192 bytes
Status:	Scan finished. 3 out of 20 scanners reported malware.	Filetype:	PE32 executable for MS Windows (console) Intel 80386 32-bit
Scan taken on:	Wed 2 May 2012 06:57:10 (CET) Permalink	MD5:	8a72a73554740e460ece1c874c19376a
	<input type="button" value="Next file"/>	SHA1:	d79ae70d646b8acdcf2ef2bd4ddf2a52c819f346
		Packer (Kaspersky):	PE_Patch





















Scanners			
	2012-05-02	Found nothing	
	2012-05-01	Found nothing	
	2012-05-01	Found nothing	
	2012-05-02	HEUR/Malware	
	2012-05-02	Found nothing	
	2012-05-01	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-01	Win32/Kryptik.AVF	
	2012-05-01	W32/Troj_Obfusc.L.gen!Eldorado	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-01	Found nothing	
	2012-04-30	Found nothing	
	2012-05-02	Found nothing	
	2012-04-30	Found nothing	
	2012-05-01	Found nothing	

Figure 18 Scan Results for Virus with Metamorphism

7.4 Metamorphic Virus with Delay Routines

Introduction of delay routines and call to Sleep API function caused F-PROT to declare the virus as clean, indicating a small time-out configuration of its emulation. The heuristic analysis of Avira AntiVir and ESET NOD32 showed resilience against all code obfuscations

Detections – 2/20 (10%)

Jotti's malware scan		Additional info	
Filename:	xyzhm.EXE	File size:	8192 bytes
Status:	Scan finished. 2 out of 20 scanners reported malware.	Filetype:	PE32 executable for MS Windows (console) Intel 80386 32-bit
Scan taken on:	Wed 2 May 2012 09:33:45 (CET) Permalink	MD5:	d5398323efbdb6be4ffa9ee627320d17
	Next file	SHA1:	d43a2a0ce04372c0b420a8450532af9443c57cbf
		Packer (Kaspersky):	PE_Patch












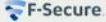








Scanners			
	2012-05-02	Found nothing	
	2012-05-01	Found nothing	
	2012-05-01	Found nothing	
	2012-05-02	HEUR/Malware	
	2012-05-02	Found nothing	
	2012-05-01	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-01	unknown NewHeur_PE	
	2012-05-01	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-02	Found nothing	
	2012-05-01	Found nothing	
	2012-04-30	Found nothing	
	2012-05-02	Found nothing	
	2012-04-30	Found nothing	
	2012-05-01	Found nothing	

Figure 19 Scan Results for Metamorphic Virus with Delay Routines

7.5 End Virus with Encrypted String Constants

Final virus was obtained by encrypting string constants which could raise suspicious flags on heuristic analysis. Avira heuristic engine alert depended on the presence of ‘*.Exe’ string constant in the NGVCK virus, which was flagged clean when the string was stored as its CRC32 hash. Only NOD32 antivirus detected it as ‘unknown NewHeur_PE’, proving its robustness for detection of new malware.

Detections – 1/20 (5%)

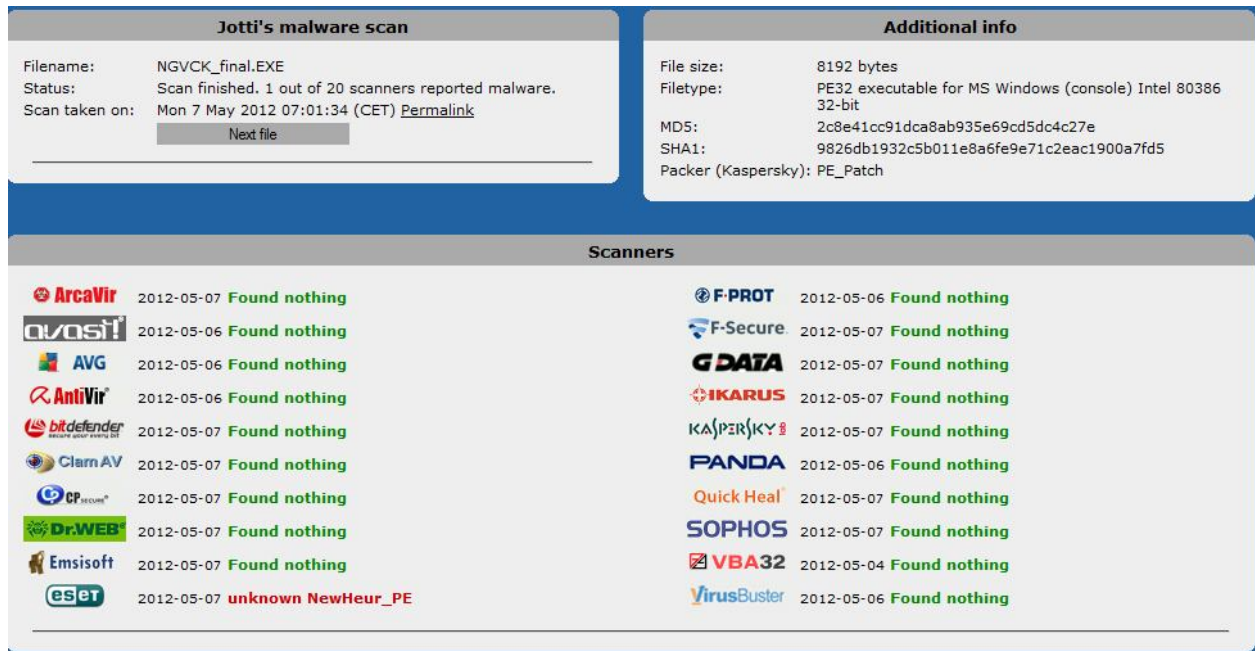


Figure 20 Scan Results for End Virus

The detection rates for various samples of the virus can be compared from the following bar graph

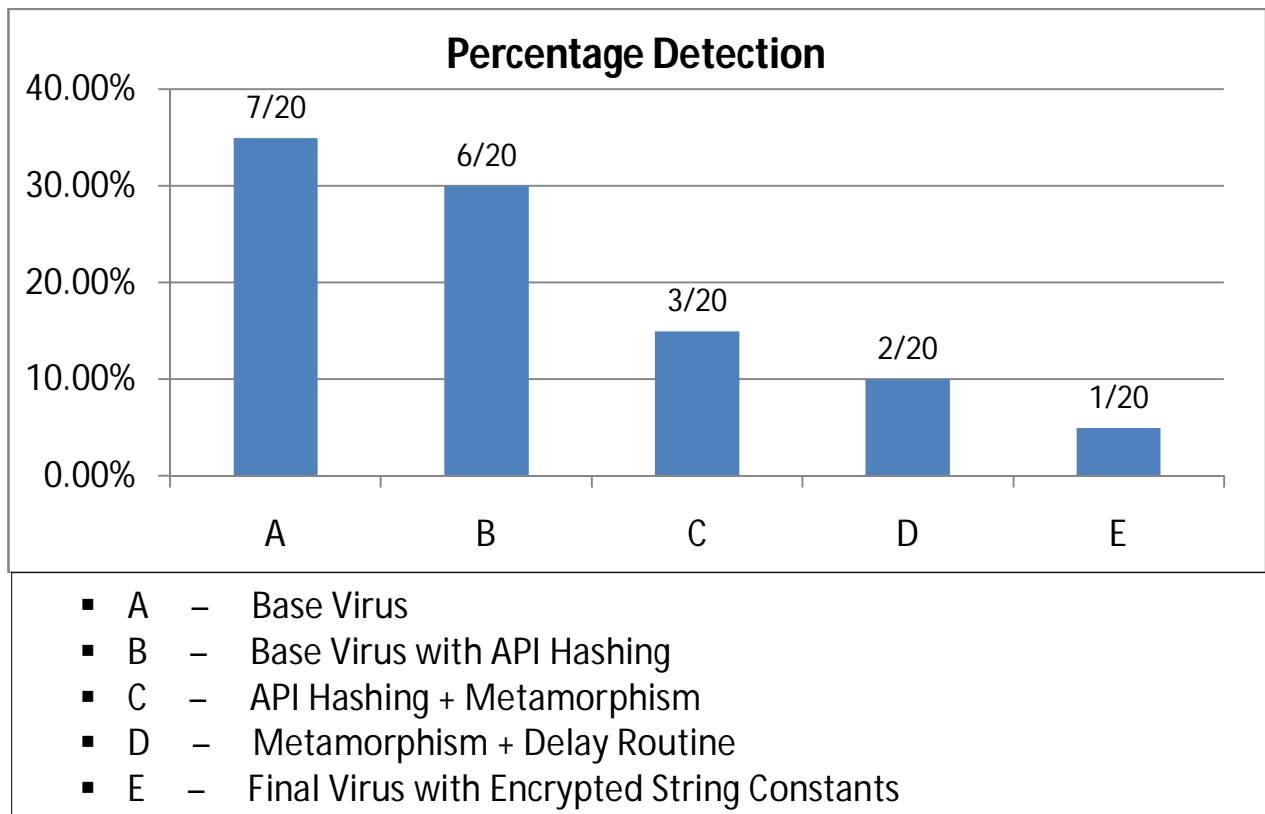


Figure 21 Detection rate of different virus samples

The scans obtained above can be gathered to draw a comparative analysis between the major anti-virus products available in the market today. The performance evaluation is done on the basis of parameters such as the strength of signature detection algorithms, immunity against anti-heuristic techniques like API name hashing and delaying and robustness of code emulation process. The remaining antivirus engines are similar in behavior of that of AVG and hence not mentioned explicitly.

Anti Virus Product	Strong signature detection	Immune to API Hashing	Immune to Delaying	Strong code emulation
Avira Antivir	✓	✓	✓	
F-Secure	✓	✓		
F-Prot	✓			
AVG				
ESET NOD32	✓	✓	✓	✓
BitDefender	✓	✓		
VBA32	✓	✓		

Figure 22 Feature comparison of Anti-virus engines

Chapter 8. Conclusion

A metamorphic virus, consisting of an engine employing code obfuscation techniques, is able to bypass weak signature based detection systems. However, most of the rated anti-virus engines today employ a mixture of both signature based and heuristic detection. Heuristic analysis and code emulation techniques were shown to be inefficient by simple modifications at right locations. Only two of the twenty AV engines tested proved to be reliable. Thus, the detection of NGVCK virus was brought down from 7/20 to a 1/20 ratio, highlighting the inability of modern day antivirus software to prevent malicious activity on systems, if carefully crafted.

References

- [1] Aycock J., “Computer Viruses and Malware”, Springer Publications 2006.
- [2] Desai P., “Towards an Undetectable Computer Virus”, Master’s thesis, San Jose State University, 2008
- [3] Szor, P., “Advanced code evolution techniques and computer virus generation toolkits”, March 2005, <http://www.informit.com/articles/article.aspx?p=366890>
- [4] Lin, D., Stamp, M., “Hunting for undetectable metamorphic viruses”, *Journal in Computer Virology* Vol. 7, No. 3 (2011), pp.201-214.
- [5] Schmall, M., “Heuristic Techniques in AV Solutions: An Overview”, February 2002
- [6] Szor, P., “The Art of Computer Virus Defense and Research,” Symantec Press 2005.
- [7] Borello, J. and Me, L. (2008) ‘Code obfuscation techniques for metamorphic viruses’, *Journal in Computer Virology, Vol. 4, No. 3, pp.211–220.*
- [8] Jotti’s Malware Scan, <http://virusscan.jotti.org/en>
- [9] Wong, W., Stamp, M.: Hunting for metamorphic engines. *J. Comput. Virol.* 2(3), 211–229 (2006)
- [10] Suenaga M., “A Museum of API Obfuscation on Win32”, Symantec Security Response, November 2009
- [11] “Benny/29A”, Theme: metamorphism, <http://www.vx.netlux.org/lib/static/vdat/epmetam2.htm>,
- [12] VX Heavens, <http://vx.netlux.org/>,
- [13] http://en.wikipedia.org/wiki/Computer_virus

[14] Masrom M., Rad B., Ibrahim S., “Evolution of Computer Virus Concealment and Anti-Virus Techniques: A Short Survey”, International Journal of Computer Science Issues, Vol. 8, Issue 1, January 2011

[15] Mishra U., Methods of virus detection and their limitations, <http://trizite.com>

Appendix A: Equivalent instruction substitution [2]

Notations:

R – Register (eax, ax, ah, al)

RR – Random register

mem, [mem] – Memory address ([esi])

imm – Immediate value (12h)

op1 – To-operand with length more than 1 including R and mem

op2 – From-operand with length more than 1 including R, mem, and imm

loc – any location or label

add R, imm	3. sub R, new_imm where new_imm = imm x (- 1) 4. lea R, [R + imm]
add R, 1	3. not R neg R
mov R, imm	1. mov R, random_imm add R, new_imm where new_imm = imm - random_imm 2. mov R, random_imm sub R, new_imm where new_imm = (random_imm - imm) mov R, random_imm xor R, new_imm
mov R1, R2 (no 8 bit R)	1. push R2 pop R1
mov R, mem (no 8 bit R)	1. push mem pop R
mov R, imm (no 8 bit R)	1. push imm pop R 2. lea R, [imm]
mov mem, R (no 8 bit R)	1. push R pop mem
mov mem, imm	1. push imm pop mem
cmp R, 0	1. or R, R 2. and R, R 3. test R, R
cmp R1, R2	1. sub R1, R2
cmp R, mem	1. sub R, mem
cmp R, imm	1. sub R, imm
cmp mem, R	1. sub mem, R
cmp mem, imm	1. sub mem, imm
and R1, R2	1. push RR mov R, R1

	or R, R2 xor R1, R2 xor R1, R pop RR 2. not R1 not R2 or R1, R2 not R1
dec R	1. neg R not R
dec mem	1. neg mem not mem
inc R	1. add R, 1 2. not R neg R
inc mem	1. add mem, 1 2. not mem neg mem
invoke op1, op2	1. stdcall [op1], op2
jmp loc	1. cmp RR, RR jz loc
jmp R	1. push R ret
lea R, [R1 + R2]	1. mov R, R1 add R, R2
lea R, [R + R1 + imm]	1. add R, imm add R, R1
lea R, [R1 + R2 + imm]	1. lea R, [R1 + imm] add R, R2
lodsb	1. mov al, [esi] add esi, 1
lodsd	1. mov eax, [esi] add esi, 4
movsb	1. push eax mov al, [esi] add esi, 1 mov [edi], al add edi, 1 pop eax
movsd	1. push eax mov [eax], esi add esi, 4 mov [edi], eax

	add edi, 4 pop eax
neg R	1. not R add R, 1
neg mem	1. not mem add mem, 1
not R	1. neg R sub R, 1 2. neg R dec R 3. neg R add R, -1 4. xor R, -1
not mem	1. neg mem sub mem, 1 2. neg mem dec mem 3. neg mem add mem, -1
or R1, R2	1. push RR mov RR, R1 xor RR, R2 and R1, R2 xor R1, RR pop RR
or R1, mem	1. push RR mov RR, R1 xor RR, mem and R1, mem xor R1, RR pop RR
or R1, imm	1. push RR mov RR, R1 xor RR, imm and R1, imm xor R1, RR pop RR
or mem, R	1. push RR mov RR, mem xor RR, R and mem, R xor mem, RR pop RR

Table 3 Equivalent Instruction Substitutions

Appendix B: Dead code instructions [2]

Transfer Dead Code	<ol style="list-style-type: none">1. mov R, R2. push R followed by pop R
Arithmetic Dead Code	<ol style="list-style-type: none">1. add R, 02. sub R, 03. adc bx, 04. sbb bx, 05. inc R followed by dec R
Logical Dead Code	<ol style="list-style-type: none">1. shl R, 02. shr R, 03. and R, 14. test R, 15. or R, 06. xor R, 0
Floating Point Dead Code	<ol style="list-style-type: none">1. fadd st2, st02. fmul st2, st03. fld st24. fsub st2, st05. fdiv st2, st0

	6. fst st3
Miscellaneous Dead Code	1. nop 2. neg R, not R, dec R

Table 4 Dead Code Instructions [2]