Metamorphic Virus: Analysis and Detection

Evgenios Konstantinou
Supervisor: Dr. Stephen Wolthusen

Technical Report
RHUL-MA-2008-02
15 January 2008

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
http://www.rhul.ac.uk/mathematics/techreports

Abstract

Metamorphic viruses transform their code as they propagate, thus evading detection by static signature-based virus scanners, while keeping their functionality. They use code obfuscation techniques to challenge deeper static analysis and can also beat dynamic analyzers, such as emulators, by altering their behavior. To achieve this, metamorphic viruses use several metamorphic transformations, including register renaming, code permutation, code expansion, code shrinking, and garbage code insertion. In this thesis, an in-depth analysis of metamorphic viruses is presented, along with the techniques they use to transform their code to new generations. In order to give a better understanding of metamorphic viruses, a general discussion on malicious code and detection techniques is given first. Then, the description of several techniques to detect metamorphic viruses is given. A fair number of papers on metamorphic viruses exists in the literature, but no one is a complete discussion of all metamorphic techniques and detection methods. This thesis aims at a complete discussion of all metamorphic techniques used by virus writers so far, and all detection techniques implemented in antivirus products or still experimental. It accomplishes this by an in-depth research on malware and metamorphic viruses, through the existing literature. Due to space and time limitations, an exhaustive discussion was not possible in this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

The recent years have been very interesting, but at the same time very frustrating for the information security professional. As information technology is expanding and improving, so are its threats. Its adversaries evolved from the 15 year old “script kiddy”, to the professional hacker employed by organized crime. A recent research in the UK, has shown that the exponential increase of broadband internet connection has been accompanied by high virus infection rates. The research showed that around 97% of businesses in the UK have internet connection and around 88% have broadband, thus the thread from malicious software has never been greater [1].

A recent research funded by the UK government, reports that virus infection was the biggest single cause of respondents' worst security incidents, accounting for roughly half of them. Two-fifths of these were described as having a serious business impact. The report also informs that virus infections tended to take more effort to resolve than other incidents, some of them needing more than 50 person-days.

The latest trend for cyber criminals, is to target confidential information in order to use it for fraud, such as identity theft, or sell the information to third-parties for financial gain. From the volume of the top 50 malicious code reported to Symantec between July 1 and December 31, 2006, 66% targeted confidential information [2]. This is very serious as it demonstrates that modern virus writers threaten not only our precious time – as they used to do – but also our reputation and money.

With the exception of rootkits, – which is a very recent technology – metamorphic viruses must be the most sophisticated malicious pieces of code. To write a decent metamorphic engine is a very challenging task and this is why we haven't seen many examples of this type of virus in the wild. Some of them are so well written that modern antivirus products can still miss them some times, as shown by Christodorescu and Jha in [3]. Because of their complexity their study is very interesting, and the fact that here were no real metamorphic viruses in the wild since Simile in 2002, should not make the virus researcher relaxed. The technology is there and waiting to be exploited and implemented into modern types of malware, such as network worms and spyware.

1.2 Outline

The first three chapters are introductory and their purpose is to prepare the reader for the most advanced metamorphic techniques and their detection. In Chapter 2, an introduction to different forms of malicious software is given. It begins by giving a formal definition of a computer virus and then by describing the different types of viruses that exists. Then, the computer worm and the Trojan horse are defined. In the last two sections two more recent malware are described; spyware and – the very dangerous – rootkits. Chapter 3 describes some of the most common and widely used – by antivirus products – detection techniques. It begins with the most simple techniques and finishes with some more recent and more advanced techniques for detection of computer viruses. Chapter 4 describes some more advanced code evolution techniques, implemented by virus writers in order to make their viruses avoid detection. It begins with the simple encrypted virus and finishes with the more advanced polymorphic virus.

The advanced metamorphic virus is described in Chapter 5. Formal definitions of a metamorphic virus are provided and a description of the way they work and why they are dangerous is given. Then, Chapter 5 describes the different techniques used by metamorphic viruses to avoid detection by mutating their code and form new generations. In the last section, a more detailed description of the two most advanced metamorphic viruses ever created is given: Win95/Zmist and {Win32,Linux}/Simile. Chapter 6 describes several different techniques used to detect metamorphic viruses. It begins by discussing the weak points of a metamorphic virus and why although very difficult to detect, they are not invisible. Then, the detection techniques implemented into antivirus software are described. The last section, describes several experimental techniques published in academic papers, but not yet implemented in commercial antivirus products. This thesis concludes with some future trends on general malware and metamorphic malware.

1.3 Final Remarks

The term *computer virus* was originally used by Dr. Fred Cohen in his PhD thesis, in 1986 [4]. The term has been widely used and is now a synonym to any form of malicious software. However, since 1986 many other forms of malicious software were created, such as computer worms and Trojans, which do not follow Dr. Cohens' definition of a computer virus. Thus, the term computer virus is not technically correct to describe worms or Trojans. However, the term computer virus is used excessively in literature to describe all forms of malicious software. In this report, the term *computer virus* will only be used to describe the file infecting, replicating malicious code defined in section 2.1. The term *malware* will be used to describe all forms of malicious software. However, the term *virus writer* will be used to describe the person who is responsible for creating all types of malicious software.

Chapter 2

Introduction to Malicious Software

Malicious software or malware for short, are “programs intentionally designed to perform some unauthorized – often harmful or undesirable – act” [5]. Malware is a generic term and is used to describe many types of malicious software, such as viruses and worms.

This chapter describes the most popular types of malicious software, which are computer viruses, worms, Trojan horses, spyware, and rootkits. Emphasis is given to the first three, as spyware and rootkits are less relevant with the subject of this paper – the analysis and detection of metamorphic viruses.

2.1 Viruses

In 1987, Fred Cohen, the pioneer researcher in computer viruses, defined a computer virus to be: *“A program that can infect other programs by modifying them to include a possibly evolved copy of itself”*. Figure 2.1 on the following page, is Dr. Cohens’ pseudo-code of a simple computer virus.

This is a typical structure of a computer virus which contains three subroutines. The first subroutine, *infect-executable*, is responsible for finding available executable files and infecting them by copying its code into them. The subroutine *do-damage*, also known as the payload of the virus, is the code responsible for delivering the malicious part of the virus. The last subroutine, *trigger-pulled* checks if the desired conditions are met in order to deliver its payload. Examples of conditions could be the day of the week, the number of infections, or the current date.

Nowadays, there are so many different kinds of computer viruses that it is difficult to provide a precise definition. In [7], it is suggested that Cohens’ definition can be a bit misleading if taken to its strictest sense. For example, *companion viruses* do not strictly follow Cohens’ definition because they do

```
program virus :=
{1234567;

subroutine infect-executable :=
   {loop: file = random-executable;
   if first-line-of-file = 1234567
      then goto loop;
   prepend virus to file;
   }

subroutine do-damage :=
   {whatever damage is desired}

subroutine trigger-pulled :=
   {return true on desired conditions}

main-program :=
   {infect-executable;
   if trigger-pulled then do-damage;
   goto next;
   }

next:}
```

Figure 2.1: Simple virus V (from [6])

not modify the code of other programs and do not need to include a copy of themselves within other programs.

Peter Szor in [7], attempts to give a more accurate definition: *"A computer virus is a program that recursively and explicitly copies a possibly evolved version of itself."* He also explains that viruses infect other files or system areas with intend to multiply themselves and form new generations. With a few exceptions, viruses do not exist as stand alone files or programs but require a host file. Viruses require user interaction to spread. This paper will follow the definition given by Roger A. Grimes in [8].

Definition 1. A virus is a malicious program that modifies other host files (few exceptions) or boot areas to replicate. In most cases the host object is modified to include a complete copy of the malicious code program. The subsequent running of the infected host file or boot area then infects other objects.

Boot Sector Viruses

Boot sector viruses infect the Master Boot Sector of hard drives or floppy drives and infect other machines only when the machine boots up from an infected floppy disk. Boot Sector viruses were the first successful viruses created and can infect a machine regardless of what Operating Systems runs on it [7]. Today they are rarely found because floppy disks are not so common any more.

One of the most successful boot sector viruses of all time is the Stoned virus, including his infamous variant Michelangelo. Stoned is a very simple boot sector virus, but at one time it was so prevalent that National Computer Security Association reported that roughly one out of every four virus infections involved some form of Stoned [1].

As described in [9], Stoned hides the original boot-sector somewhere on the disk and then it occupies the usual boot sector location. Then the BIOS loads the virus at startup and transfers control to it. After the virus is finished with its work, it loads the original boot sector which then loads the operating system. This technique is almost universal among boot-sector viruses and it has the advantage of being somewhat operating system independent.

File Infecting Viruses

Program viruses infect executable programs, such as EXE or COM, by attaching themselves to them. The virus executes and infects other executables when its host file is executed. To infect an EXE file, a virus has to modify the *EXE Header* and the *Relocation Pointer Table*[2], and add its own code to the *Load Module*. This can be done in many ways.

The Intruder-B virus attaches itself to the end of an EXE file and gains control when the file first executes. To do this it needs a routine which copies program code from memory to a file on disk, and then adjusts the file. The virus has its own code, data, and stack segments. At the end the virus passes control to the host program. The above description is given in [9].

Memory Resident Viruses

Memory resident viruses remain in memory after the initialization of virus code. They take control of the system and allocate a block of memory for their own code [7]. They remain in memory while other programs run and infect them. File infecting viruses could be memory resident as well.

Memory resident viruses are highly portable because they can jump across both directories and disk drives by taking advantage of the users'

[1]*NCSA News*, (Mechanicsburg, PA), Vol. 3, January 1992, p. 11

[2]A variable length table of pointers to segment references

actions as they change directories and drives in the normal use of their computer. Moreover, these viruses distribute the infection process over time better than direct acting viruses. A technique that a virus can use to go resident is to take advantage of a *memory hole*, which is an unused part of memory that is not likely to be overwritten [9].

Macro Viruses

These viruses are written in macro languages and infect files that make use of the particular language. A macro is a series of steps that could otherwise be typed, selected, or configured, but are stored in a single location so they can be automated. Some programs are nothing than hundreds of macros build around vendors' applications. Macro languages are used to allow more sophisticated macro development and environment control, like manipulating and creating files, changing menu settings, and much more. Macro languages are so easy to use that virus writers can learn to write their first virus in a couple of days.

What makes them particularly fast spreading is that users share documents and data in a much larger degree than application files. Macro viruses became a big problem because Microsoft allowed macro code to be saved within the body of a document or data files (Microsoft Office files). Macro viruses can be cross-platform and multicultural, infecting any computer capable of running Office. A single macro virus can infect different types of computers running under different languages because different versions of word share the same macro language.

Macro viruses can do almost anything that is possible to be done on a system, such as corrupting data, creating files, inserting pictures, sending files across the internet, modifying registries, looking for passwords, infecting other programs, and even formatting hard drives. They can use the VBA[3] SHELL command or utilize the operating systems' API to run any external command they want.

Macro viruses usually spread when a user opens or closes an infected document. The macros contained in the document infects the users' program and other documents. Documents are spread by using email, the web, or other portable media. The first widespread macro virus, Concept, was spread on two CD-ROMs which Microsoft made available as part of their marketing handouts. The virus then jumped from the CD-ROMs into memory, infecting other documents. All information about macro viruses comes from [8].

[3]Visual Basic for Applications: A macro language incorporated into many of Microsofts' applications.

2.2 Worms

The first researcher who tried to give a formal and mathematical definition of a computer worm was Fred Cohen in [10]. A more easy to understand definition is given in [8]:

Definition 2. A worm is a sophisticated piece of replicating code that uses its own program coding to spread, with minimal user intervention.

A worm usually exists as a standalone program that executes itself automatically on a remote machine, without any user interaction. Worms are network viruses, primarily replicating on networks [7]. They typically do not require any host files although some types of worms do. It is derived from the word *tapeworm*, a parasitic organism that lives inside a host and saps its resources to maintain itself.

A network worms' life cycle, as described in [11], is divided into seven phases. The first phase is the *Initialization Phase*, in which the worm may install software, determine the configuration of the local machine, instantiate global variables, and begin the main worm process. Following is the *Payload Activation Phase*, in which the worm activates its payload. This phase is logically distinct from the other phases and it does not usually affect the network behavior of the worm. Next is the *Network Propagation Phase*, which is the phase that encompasses the behavior that describes how a worm moves through a network. In this phase the worm selects a set of targets and tries to infect them. The *Network Acquisition Phase* describes the process a worm goes through to select targets for infection. In the next phase, *Network Reconnaissance Phase*, the worm attempts to learn about the environment, particularly with respect to the targeted hosts. This phase includes validating what a worm knows about the environment and enables the worm to make more informed decisions about its operations. Next is the *Attack Phase*, which is when the worm performs actions that enable it to acquire elevated privileges on a remote system, usually exploiting other software vulnerabilities. The *Infection Phase* is the last phase in the worms' life cycle and is when the worm leverages the acquired privileges on the target host to begin the Initialization Phase of a new instance of the worm.

What makes worms faster spreading and more dangerous than viruses is that they require no user interaction to replicate. Some worms use active network connections to send themselves and infect new targets. SQL Slammer, which was the fastest spreading worm ever created, infected its victims by randomly selecting IP addresses, eventually finding and infecting all susceptible hosts [12]. Other worms, like the infamous Melissa, attach themselves in emails and then send themselves to unsuspected users. Melissa was a Macro Virus but can also be seen as a worm because it used its own code to spread.

Worms usually exploit vulnerabilities in other software. In 1988, the famous Morris Worm exploited flaws in utility programs in systems based on BSD-derived versions of UNIX. The flaws allowed the worm to break into those machines and copy itself [13]. Slammer took advantage of a buffer-overflow vulnerability in computers on the Internet running Microsofts SQL Server or Microsoft SQL Server Desktop Engine (MSDE) 2000. More information on Slammer can be found in [12].

2.3 Trojan Horses

A Trojan horse program is any program that intentionally hides its malicious actions while pretending to be something else. The following definition was given in [8]:

Definition 3. A Trojan, or Trojan horse, is a non-replicating program masquerading as one type of program with its real intent hidden from the user.

The term *Trojan horse* comes from the Greek mythology. It was the name of a giant horse built to trick the Trojans and help the Greek soldiers sneak into Troy. Simple forms of Trojans pretend to be some kind of program, like a game or utility, but when a user runs them they do something malicious to the system, usually without the users' knowledge. Some malicious users create their own Trojan-infected programs from scratch, while others find legitimate programs, attach the Trojan to them, and place them on the internet for unsuspected users to download. In 2006, Trojans outnumbered by far windows worms or any other type of malware averaging around 80% of all malware detected by Sophos Labs throughout the year [14].

A Trojan is different from a virus as it does not copy its code to other host files or boot areas [8]; instead, it exists as a standalone program that masquerades as something else. Also, it does not copy itself in order to replicate. It depends on users to send the Trojan to other users or download the Trojan from the internet and run it on their machines. Because worms do not infect other files they are often treaded as Trojans, but worms do not masquerade as other programs as trojans do, but they use their own code to replicate. Trojans are often distributed by worms.

Some of the many different types of Trojan horses are described in the following paragraphs. All information about the types of Trojans comes from [8].

Remote administration Trojans (RAT) allow remote malicious users to have complete control of a machine. Complete control means that a malicious user can read what the user is reading, record key strokes, capture screen shots, copy and delete files, and many more. RATs consist of a client which is installed on the victims' computer, and a server which enables malicious users to remotely control the client. Some of the most popular RATs found in-the-wild was *NetBus*, *Subseven*, *DeepThroat*, and the infamous *Back Orifice*.

Backdoor Trojans open up a new entry point for hackers by installing a new TCP/IP service or mapping a new drive share. The Trojan can easily open up Windows drive shares, which can be accessed from the Internet. Also, a Trojan can allow the malicious user to modify files or install more services.

Network Redirection Trojans as their name suggests, allow network redirection. They allow malicious users to redirect specific attacks, through a compromised intermediate host, towards a new target. This can allow Trojans to subvert filtering firewalls, and also makes tracing back the attacker difficult.

Distributed Attacks Trojans spread to as many machines as they can and then wait for commands. Usually, the exploit is directed towards another central target. Up to thousands of these malicious programs can be spread over a period of months and can be then used to attack a common target, causing a Denial of Service (DoS) or gather information that can be used later. Infected machines are called *Zombies* and the set of infected machines is called a Zombie Network.

2.4 Spyware

Spyware is the name given to the class of software that is surreptitiously installed on a computer, monitors user activity, and reports back to a third party on that activity [15]. The Federal Trade Commission defines spyware as software that aids in gathering information about a person or organization without their knowledge, and that may send that information to another entity without the users' consent. Spyware includes:

Adware are applications that may monitor user web browsing activity and send targeted advertisements to the user desktop based on that browsing activity. They actually change the way a users web browser works by installing *browser helper objects*, or changing the default settings of Web browsers to display different home pages and bookmark lists and redirect searches to different search systems [15].

Key Loggers are software that monitor what a user types on the keyboard. They are used to capture usernames, ID, passwords, credit card numbers, or any other sensitive information. They are usually considered to be malicious despite the fact that some key-loggers can be used for legitimate purposes.

Trojan Horses See section 2.3.

Spyware can be used by anyone who wants to know something about another person and their computing habits [15]. Some legitimate uses include parental control programs that are used to monitor the computing habits of children and employers monitoring workers for appropriate internet use. Also, businesses are increasingly making the use of spyware to gather valuable customer data, and it is becoming increasingly popular in e-business circles to use spyware as a means to gain additional revenues [15]. But the main problem is hackers and other malicious users, who make use of spyware to steal personal information and use it for gaining money through fraud. Hackers may use the information themselves or sell the information to other interested third parties.

Spyware can be installed by viruses and worms or when a user opens the attachment of infected emails, or by just surfing to a certain web page. Malicious users exploit vulnerabilities of some web browsers to download and install spyware on a system usually without the users' knowledge – a technique known as *drive-by download*. Most spyware requires some user action for it to be installed on a computer, such as downloading a piece of software. Peer-to-peer software help spyware spread because spyware travel hidden in program utilities shared by users who use peer-to-peer software. Also, users may be tricked to download spyware by pop-up windows that ask them to download and install utility programs [16].

2.5 Rootkits

The term rootkit originally referred to a collection of tools used to gain administrative access, *root access*, on UNIX operating systems. The collection of tools often included well-known system monitoring tools that were modified to hide the actions of an unauthorized user. An unauthorized user would replace the existing tools on the system with the modified versions preventing authorized users from discovering the security breach [17]. Modern rootkits are defined as a set of malicious programmatic tools that enable an intruder to conceal the fact that a system has been compromised, by hiding files, processes, registry keys, and other objects, and to continue to make use of that compromise [18]. There are two types of rootkits: *user mode* and *kernel mode* rootkits.

User mode rootkits, like their name reveals, run in user or application mode. When an application makes a system call, a rootkit can hijack it on the way. These rootkits components run within user applications, by patching the APIs in each application process. A common technique user mode rootkits use, is replacing or modifying system DLLs in order to hide their presence. But because user mode applications run in their own memory space, the rootkit must modify the memory space of each application. It must also monitor the system for new applications and patch those applications' memory space as well before they are fully launched [18].

Kernel mode rootkits are much more dangerous than user mode rootkits because they run in system mode, which provides them with almost unlimited system privileges. Their damage potential is almost unrestricted; however, due to their complexity is much more difficult to install and maintain reliably [18]. A technique that kernel mode rootkits use is direct kernel object modification (DKOM). The rootkit can modify the data structures in the kernel memory and remove itself or other malicious processes from the kernels' list of running processes [17].

Rootkits is the new weapon in the arsenal of malicious users and it is likely that this is the path that malware creators will follow in the future. King and Chen introduced a project they call *SubVirt*, and demonstrate how attackers can use virtual machine technology (VM) to improve current malware and rootkits [19]. They showed how attackers can install a virtual machine monitor (VMM) underneath an existing operating system and use that VMM to host arbitrary malicious software. They call the resulting malware a virtual machine-based rootkit (VMBR), which has more control than current malware and supports general purpose functionality. This VMBR can hide all its state and activity from intrusion detection systems running in the infected OS and applications. They also demonstrated that a VMBR can be implemented on current hardware and can be used to implement several malicious services. What is more, they showed that once installed, a VMBR is difficult to detect and remove. This type of rootkit is very difficult to be detected because its state cannot be accessed by software running on the hosted operating system. There is still no malicious implementation of a VMBR but the technology is there and it will not be long before we see them in the wild.

More recently, a security researcher has announced the development of a technology code-named *Blue Pill*, which is about creating an 100% undetectable walware. As she explains, "the idea behind Blue Pill is simple: your operating system swallows the Blue Pill and it awakes inside the Matrix controlled by the ultra thin[4] Blue Pill hypervisor. This all happens on-the-fly (i.e. without restarting the system) and there is no performance penalty and all the devices, like graphics card, are fully accessible to the operating system, which is now executing inside virtual machine. This is all possible thanks to the latest virtualization technology from AMD called SVM/Pacifica" [21]. Figure 2.2 on the next page, represents the idea of

[4]Thin hypervisors control the target machine transparently and they are based on hardware virtualization, such as SVM, VT-x [20].

Blue Pill and was taken from Joanna Rutkowskas' presentation of Blue Pill at SyScan '06.

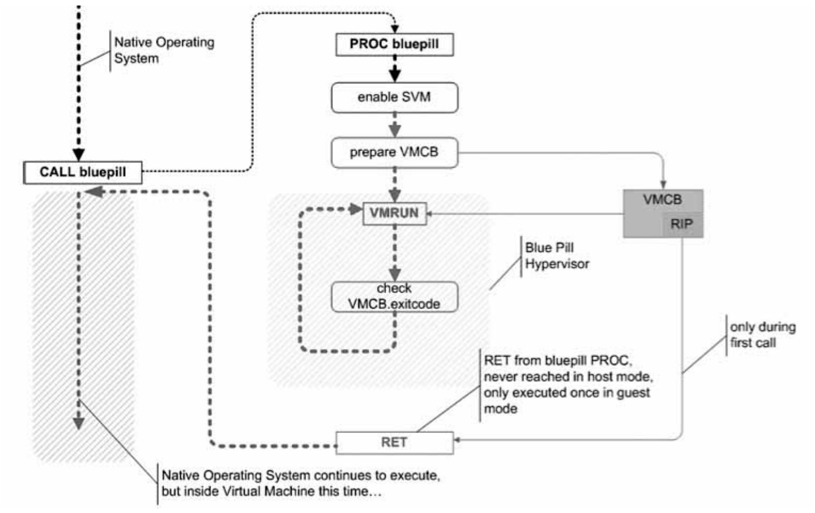

Native Operating System
PROC bluepill
enable SVM
prepare VMCB
CALL bluepill
VMCB
RIP
VMRUN
Blue Pill Hypervisor
check VMCB.exitcode
only during first call
RET from bluepill PROC, never reached in host mode, only executed once in guest mode
RET
Native Operating System continues to execute, but inside Virtual Machine this time...

Figure 2.2: Blue Pill idea (from [21])

In 2007, Joanna Rutkowska and Alexander Tereshkin decided to re-design and write from scratch the *New Blue Pill* rootkit, so that it would be possible to use it for further research. The New Blue Pill implements several new features and it's based on a different architecture than the original – the HVM-like approach. More information can be found in [20].

Chapter 3

Virus Detection Mechanisms

This chapter presents some of the most popular techniques used by antivirus software to detect computer viruses. All information about the detection techniques, except otherwise stated, were taken from Peter Szors' book "The Art of Computer Virus Research and Defense" [7].

3.1 First-Generation Scanners

First-generation scanners use simple methods to detect computer viruses. Techniques usually involve scanning for pre-defined sequences of bytes called strings.

3.1.1 String Scanning

String scanning is the simplest technique used by anti-virus software to detect computer viruses. It searches for sequence of bytes (strings) that are typical of a specific virus but not likely to be found in other programs. This sequence of bytes is often called the signature of the virus, which is extracted for each different virus and organized in a database. The virus scanner will then use this database to search files and system areas for presence of the virus. The following example string is a typical pattern of the W32/Beast virus in EXE files, which was published in [22].

```
83EB 0274 1683 EB0E 740A 81EB 0301 0000
```

The virus scanner checks the code of EXE files and when it encounters the previous string it announces detection of the Beast virus. Usually sixteen unique bytes is long enough to detect a 16-bit malicious code without any false positives. But for 32-bit viruses to be detected safely, longer strings might be necessary, especially when a virus is written in a high level language.

3.1.2 Wildcards

Scanners that support wildcards are allowed to skip bytes or byte ranges. In the following example the bytes represented by the '?' character are skipped. The wildcard %2 means that the scanner will try to match the next byte – 03 in our example – in the two positions that follow it.

```
83EB 0274 ??83 EB0? 740A 81EB %2 0301 0000
```

Some early generation encrypted and polymorphic viruses can be detected using wildcards. In [23] is demonstrated that even the metamorphic virus W32/Regswap could be detected using this method.

3.1.3 Mismatches

Mismatches allow any given number of bytes in a string to be of arbitrary value, regardless of their position. The string 11 22 33 44 55 66 77 88 with mismatch value 3, would match the following strings.

```
A3 11 22 33 C9 44 55 66 0A 77 88

11 34 22 33 C4 44 55 66 67 77 88

11 22 33 44 D4 DD E5 55 66 77 88
```

This technique was developed for the IBM Antivirus and is useful in creating better generic detections for families of viruses. However, it is a relatively slow scanning algorithm.

3.1.4 Generic Detection

When more than one variants of a virus are discovered, the variants are compared to find a common string in their code. This technique uses one common string to detect several or all known variants of a family of viruses. Usually, this technique makes use of wildcards and mismatches.

3.1.5 Bookmarks

Bookmarks are a simple technique to guarantee more accurate detections and disinfections. An example bookmark could be the *zero byte* of the body of a virus which is the distance between the start of the virus body and the detection string. In case of boot viruses, good bookmarks could point to references of the locations of the stored boot sectors. In case of file viruses, bookmarks should point to an offset to the stored original host program header bytes. Also, the size of the virus could be a very useful bookmark. These bookmarks are calculated and stored separately in the virus detection record.

3.1.6 Top-and-Tail Scanning

Scanning only the first and the last 2, 4, or 8KB of a file for each possible position is a good way to make virus detection much faster. This technique is called top-and-tail scanning and is used to optimize scanning speed by reducing the number of disk reads. Top-and-tail scanning became popular as the majority of early computer viruses prefixed, appended, or replaced host objects.

3.1.7 Entry-Point and Fixed-Point Scanning

Another two techniques designed to make antivirus scanners even faster. They take advantage of the entry-point of objects, which are made available in the headers of objects such as executable files. The entry-point is a common target for viruses, so entry-point scanners focuses on that position and typically have a single position to mask their scan string.

Fixed-point scanning is used when the entry point does not have enough good strings. The scanner sets a start position M and then match each string at positions $M + x$ bytes away from this fixed point. Typically, x is 0 so the number of computations is reduced and also the disk I/O is reduced.

3.2 Second-Generation Scanners

The next generation scanners appeared when simple string matching was no longer enough to detect the more advanced computer viruses that started to appear. Also, exact and nearly exact identification were introduced, which made the scanning process more reliable.

3.2.1 Smart Scanning

When virus mutator kits appeared, simple string matching detection techniques were not very useful because these mutation kits made the virus look very different from its original form. The mutation kits worked with assembly code and tried to insert junk instructions into the source code.

Smart scanning could skip junk instructions, such as NOPs, in the host file and also did not store them in the virus signature. To enhance the likelihood of detecting related variants of viruses, an area of the virus body was selected which had no references to data or other subroutines. Smart scanning is also useful in detecting script and macro viruses, which appear in textual forms. Smart scanning can drop characters like Space and TAB, which are used to change the form of the virus body and thus enhance the detection process.

3.2.2 Skeleton Detection

Skeleton detection does not use strings or checksums for detection and is especially useful for detecting macro virus families. The scanner parses the macro statements of the virus line-by-line and drops all nonessential statements and all white spaces. What is left is the skeleton of the body that has only essential macro code common in macro virus, which is then used by the scanner to detect the virus.

This technique was invented by the Russian virus researcher Eugene Kaspersky, designer of the Kaspersky Anti-Virus (KAV).

3.2.3 Nearly Exact Identification

Nearly exact identification uses different methods to detect viruses more accurately. One method is to use two strings, *double string detection*, instead of just one. If both strings are found, the virus is nearly exact identified and this makes disinfection more safe as the identified virus is less likely to be a variant of the original – which needs a different disinfection method. The use of bookmarks makes this technique much safer.

The use of checksum ranges selected from a virus body is another method for nearly exact identification. A checksum of the bytes of a targeted area in a virus body is calculated. This allows for longer areas of the virus body to be selected for better accuracy, without overloading the antivirus database.

Nearly exact identification can also be achieved without any use of search strings. Eugene Kaspersky in his KAV algorithm is not using any strings, but rather makes use of two cryptographic checksums which are calculated at two preset positions and length within an object.

3.2.4 Exact Identification

Exact identification uses as many ranges of constant bytes in the virus body as necessary to calculate a checksum of all constant bits of the virus body. The variable bytes of the virus body are eliminated and a map of all constant bytes is created. This method is the only method to guarantee precise identification of virus variants and is usually combined with first-generation techniques. It can also differentiate precisely between different variants of the same virus.

Exact identification has many benefits, but scanners which implement this technique are slow. Also, it is very difficult to map the content ranges of large computer viruses.

3.2.5 Heuristics Analysis

Heuristics analysis is useful when detecting new viruses. This technique is also particularly useful for detecting macro viruses. For binary viruses heuristic analysis can be very useful too, but it creates many false positives which is a big problem for scanners. If an antivirus scanner creates many false positives, users will loose their trust in the scanner and would not buy it.

However, in many cases a heuristic analyzer can be very useful and can also be used to detect variants of existing virus families. Heuristic analysis, as described in [24], can be static or dynamic. Static analysis base the analysis on file format and the code structure of the virus body. Dynamic heuristics use emulators to detect suspicious behavior while the virus code runs inside the emulator. More on emulation techniques in section 3.4.

The following are examples of heuristic flags, which describe particular structural problems not likely to be found in PE files compiled with a 32-bit compiler.

- Code Execution Starts in the Last Section
- Suspicious Section Characteristics
- Virtual Size Is Incorrect in PE Header
- Possible "Gap" Between Sections
- Suspicious Code Redirection
- Suspicious Code Section Name
- Suspicious Imports from KERNEL32.DLL by Ordinal
- Multiple PE Headers

The previous heuristic tests come from [7].

3.3 Algorithmic Scanning Methods

There are cases when the standard algorithm of the virus scanner cannot deal with a virus. In cases like this, a new detection code must be introduced to implement a virus-specific detection algorithm. This method is called algorithmic scanning. In [7], is suggested that the name *algorithmic scanning* could be misleading and *virus-specific detection* would be more accurate.

Early scanners implemented algorithmic scanning by hard-coding detection routines that were released with the core engine code, but this technique caused many problems like stability issues of the scanner. To solve this problem, researchers introduced virus scanning languages, which allowed seek and read operations in scanned objects.

Modern scanners implement algorithmic scanning as a Java-like p-code using virtual machines. This makes the detection routines highly portable and cross-platform, however p-code execution is relatively slow compared to real-time code execution. In the future it is expected that algorithmic scanners will implement a JIT (jut-in-time) system to compile the p-code-based detection routines to real architectural code [7].

3.3.1 Filtering

Filtering is a popular technique used widely in second-generation scanners to increase the performance of the scanner regarding speed. A typical virus only infects a subset of known objects. For example, macro viruses only infect files or documents that are able to run macros, executable viruses infect executable programs like EXE and COM, boot viruses only infect boot areas, etc. This gives an advantage to the scanner as it can skip objects that are not likely to be infected with a specific virus, thus reducing the number of string matches the scanner must perform.

Because algorithmic scanning is slow and more expensive in terms of performance, it relies heavily on filtering. Some examples of filters are: type of the executable, the identifier flags of a virus in the header of an object, and suspicious code section characteristics. However, some viruses give little opportunities for filtering.

3.3.2 Static Decryptor Detection

Some viruses encrypt their code to avoid detection using search strings (see section 4.1 on page 23). This technique causes problems to virus scanners because the ranges of bytes that the scanner can use to identify the virus are limited. Using decryptor detection specific to certain viruses is not a very good method because it can be very slow and can also cause many false positives and false negatives. What is more, because the virus body is encrypted, this technique cannot guarantee disinfection.

However, decryptor detection can be relatively fast if combined with efficient filtering. It can also be used to detect polymorphic viruses (see section 4.3 on page 28) as even strong mutation engines use at least one constant byte in their decryptor. Understanding advanced polymorphic engines like MtE (see section 4.3.2 on page 29) can be a very tedious task, but unfortunately sometimes this is the only way to detect such viruses.

3.3.3 X-RAY Scanning

This is another method to detect viruses that use encryption in order to avoid detection. Instead of searching for the decryptor, this method attacks the encryption of the virus code. X-raying uses a method relying on the known plaintext on the virus body. It performs all single-encryption methods on selected areas of files, such as top, tail, and near entry-point code.

This way X-ray scanning can detect encrypted and advanced polymorphic viruses using search strings.

X-raying has been used since the DOS days to to detect encrypted and polymorphic viruses without having to emulate their decryption code. This technique takes its name from the use of x-rays in medical science, it is used to provide a "picture" of the virus, seeing through the layers of encryption [25]. X-raying attacks certain weaknesses on the virus encryption algorithm.

A big advantage of this method is its ability to detect an infection without having to locate the decryption code in the infected object. This is good for detecting Entry-Point Obscuring (EPO)[1] viruses, because these viruses can bury the decryption loop and the decryption key so deeply in the infected object that parsing the object to find the key is more costly than X-raying the body of the virus [25].

The following simple X-raying example was given in [25]. One very popular encryption method used in viruses is byte exclusive or XOR. Each byte of the ciphertext is derived from one byte of the plaintext by XOR-ing it with a fixed byte value between 0 and 255. The fixed byte value is the encryption key. For example, the plaintext:

```
P = E8 00 00 5D
```

is encrypted with key 0x99, by XOR-ing each byte with the key. After encryption P becomes:

```
C = (E8 XOR 00) (00 XOR 99) (00 XOR 99) (5D XOR 99)
C = 71 99 99 C4
```

We can determine if C could be an encryption of P in two steps. First, we guess what the value of the key k must be according to the value of the first byte of C. In our example, on the assumption that 71 = E8 XOR k, we guess that k = 0x99. The next step is to verify that the rest of the ciphertext C decrypts correctly by applying the guessed key k to the following bytes.

The downside of this technique is that it is relatively slow. Also, additional slowdown is introduced when the start of the virus body cannot be found at a fixed position and therefore the encryption methods must be performed on a long area of the file. However, the complete decryption of the virus body makes disinfection possible.

3.4 Code Emulation

This extremely powerful technique implements a virtual machine to simulate the CPU and memory management system and executes malicious code

[1]With the EPO method, some place in the victim body is patched by virus instructions in the hope that this point will gain control somewhere [26].

inside the virtual machine. The malicious code cannot escape the virtual machine of the scanner, thus this technique is relatively safe.

The code emulator mimics the instruction set of the CPU using virtual registers and flags. The functionality of the operating system must also be emulated to create a virtualized system that supports system APIs, files, memory management, etc. The emulator mimics the execution of programs and analyzes each instruction opcode one-by-one.

To detect polymorphic viruses, the scanner examines the content of the virtual machines' memory after a predefined number of iterations or when any other stop conditions are met. If emulation is long enough, polymorphic viruses will decrypt themselves and present their real code in the virtual machines' memory. In order to decide when to stop the emulation, the scanner uses the following methods: Tracking of active instructions, tracking of decryptor using profiles, and stopping with break points. When the emulation stops, the virus code is identified by using search strings or any other detection method.

Code emulation scanning techniques depend on the iterations of the decryptor loop. The method is fast enough for short decryptors, but for longer decryption loops the decryption of the virus – even partially – can take more than several minutes. The code of self-encrypting viruses can be decoded easily using emulation; however, in situations that emulation is not the optimal solution, the virus code can be decoded using a subprogram that applies cryptanalysis to this code [27].

The are two types of virtual machine emulators, as described in [28]: *Hardware-bound emulators* and *pure software emulators.* Hardware-bound emulators rely on the real, underlying CPU to execute non-sensitive instructions at native speed. Hardware-bound emulators can be split into two subcategories, *hardware-assisted* and *reduced privilege guest.* The most widely used type is reduced privilege guest emulators which include the popular *VMWare*, *VirtualPC* and *Parallels.*

The second type of emulators is *pure-software emulators*, which work by performing equivalent operations in software for any given CPU instruction. Examples of this type of emulators include *Hydra*, *Bochs*, and *QEMU.* Antivirus software use a method of emulation which emulates both the CPU and a portion of the operating system. Two examples of this type are *Atlantis*, which supports DOS, Windows, and Linux, and *Sandbox*, which supports only Windows.

All types of emulators are not designed to be completely transparent from the system, so there are methods that can be used to detect them. Some malicious software, in order to avoid detection, change their behavior or refuse to run at all, if they detect the presence of an emulator. Methods that can be used to detect the presence of each emulator mentioned in this section, are described in [28].

3.4.1 Dynamic Decryptor Detection

Dynamic decryptor detection is a combination of emulation and decryptor detection and it is useful for viruses with longer loops. It identifies the possible entry-point of the virus in a virus-specific manner. Specific algorithmic detection can check which areas of the virtual machines' memory have been changed during emulation, and if suspicious changes are found, additional scanning can check which instructions were executed. These instructions can be profiled and the essential set of decryptor instructions can be identified, which is then used to detect the virus. However, in order to completely decrypt a virus, the emulator must run for a longer time, thus making this technique not useful for disinfection.

In order to detect difficult polymorphic viruses a new dynamic technique was introduced, which uses code optimization in an attempt to reduce the polymorphic decryptor to a certain core set of instructions – removing the junk. Junk instructions, such as NOPs and jumps that do not have any effect in changing state, are removed. This makes emulation faster and provides a profile of the decryptor for identification. However, this technique cannot be applied in all cases. Complex polymorphic mutation engines like MtE, produce code that cannot be optimized effectively. Furthermore, this technique is not useful against viruses that apply multiple encryptions on top of each other and dependent on one another.

Chapter 4

Advanced Code Evolution Techniques

Malware writers are continually trying to invent new methods to defeat antivirus software. Their worst enemies are the most commercially popular antivirus products. Virus writers had to come up with ideas that made first-generation virus scanners useless.

This chapter describes the most popular techniques designed to make virus capable of evading detection from antivirus scanners. In particular, virus encryption techniques, 32-bit oligomorphic, and 32-bit polymorphic viruses are described. Although metamorphic viruses belong to this category, they are not discussed in this chapter as chapter 5 is completely devoted to them.

4.1 Encrypted Viruses

One of the first and easiest methods virus writers used to hide the functionality of the virus code was encryption. Usually, an encrypted virus consists of two parts; the decryptor and the encrypted main body of the virus. The decryptor executes when an infected program runs, and decrypts the virus body. Virus writers use encryption for the four following reasons, as described in [29]:

1. **To prevent static code analysis.** Static analysis of code involves disassembling the code and examining it for suspicious instructions or blocks of code. An example of a suspicious instruction is INT 26H, which performs absolute disk write, bypassing the file system. The use of encryption can disguise suspicious instructions and prohibits the use of static analysis programs that search for the previous suspicious instructions.

2. **To prolong the process of dissection.** Although encryption makes the analysis of the virus code more difficult, it usually does not add more than a few minutes to the time required for analysis. However, the Whale virus dedicates most of its code to encryption, in an attempt to confound disassembly. Viruses designed to confound disassembly are sometimes called *armored viruses.*

3. **To prevent tampering.** The use of encryption makes it more difficult for anyone to modify a virus and create new variants. Anyone who wants to do this must first decrypt the virus, make any desired changes and re-encrypt it again before reassembling it.

4. **To evade detection.** In early encrypted viruses, the decryptor was identical in all infected files. This made detection based on the decryptor alone an easy task. However, more sophisticated viruses use self-modifying encryption which makes detection based on the decryptor impossible, as no two samples of the same virus have any usable search string in common.

A very simple encryption method was used by the Pretoria virus. The virus XOR-ed each byte with a fixed value, which is equivalent to a simple substitution algorithm. The XOR command is very practical for viruses because XOR-ing with the same key twice results in the initial value. This way virus writers can avoid implementing two different algorithms for encryption and decryption. The following code of the Pretoria virus comes from [29].

```
again:
lodsd                ; get a byte to decrypt
xor     al, 0a5h     ; decrypt using key
stosb                ; and store it back
dec     bx           ; finished ?
jnz     again        ; if not, continue
```

One of the first viruses that used encryption was the DOS virus Cascade. Its algorithm was a little more sophisticated that the simple substitution algorithm. It consists of XOR-ing each byte twice with variable values, one of which depends on the length of the program [29]. The following decryptor of the Cascade virus comes from [7].

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

Decrypt:
xor     [si],si ; decryption key/counter 1
xor     [si],sp ; decryption key/counter 2
```

```
inc     si      ; increment one counter
dec     sp      ; decrement the other
jnz     Decrypt ; loop until all bytes are decrypted

Start:          ; Encrypted/Decrypted Virus Body
```

Although such encryption is considered cryptographically weak, early antivirus programs could detect them only by detecting the decryptor using search strings. The problem with this method is that by detecting a virus based on its decryptor alone, the algorithm is unable to identify the variant or the virus, as several viruses with different functionality could implement the same decryptor. Antivirus programs that use this method would be unable to repair infected files, and would also produce false positives as some non-viruses, such as antidebug wrappers, might use a similar decryptor [7].

The previous code evolution method was also used in 32-bit Windows viruses, such as Win95/Mad and Win95/Zombie. Such viruses can be detected without trying to decrypt the virus body, but such detection is not exact. Fortunately, the repair code can decrypt the encrypted virus body and deal with minor variants easily [7].

The following strategies, which are used to make encryption and decryption more complicated, are described in [7]:

- The virus writer can change the direction of the loop and can support forward and backward loops.
- Some viruses, such as RDA.Fighter, do not store the encryption key in the virus body and use exhaustive key search to decrypt themselves. This is sometimes called the RDA (random decryption algorithm). Such viruses are much harder to detect.
- Other viruses make use of strong encryption algorithms to encrypt their body. The IDEA family of viruses uses the IDEA cipher but because the viruses carry the decryption key, the encryption cannot be considered strong. However, the disinfection of such viruses is difficult because the antivirus has to re-implement the encryption algorithm.
- Some computer worms used the Microsoft crypto API which encrypts DLLs on the system using a secret/public key pair generated on the fly. Examples of malware making use of the crypto API is the Win32/Crypto and the Win32/Qint@mm.
- In some viruses, such as Win95/Resur and Win95/Silcer, the decryptor in not part of the virus code. They force the Windows Loader to relocate the infected program images when they are loaded into memory. The virus injects special relocations for the purpose of decryption

and does this so the act of relocating the images decrypts the virus body.

- The Cheeba virus kept the encryption key external from the virus body. Its payload was encrypted using a filename and only when the virus accessed the filename would it decrypt its body.
- Virus writers can use different ways to generate the encryption key. Some ways are constant, random but fixed, sliding, and shifting.
- Some viruses such as Tequila, used the decryptors' code functions as a decryption key. If the code of the decryptor is modified with a debugger, the previous method can cause problems. The technique can also cause problems to emulators that use code optimization techniques to run decryptors more efficiently.
- A very important factor for encrypted viruses is the randomness of the encryption key. Some viruses generate new keys only once per day while others generate new keys every time they infect an object. To select the seed of randomness viruses can use timer ticks, CMOS time and date, CRC32, etc.
- The virus writer can choose to decrypt the code in different locations. The most common technique is to decrypt the code at the location of the encrypted virus body. The problem with this method is that the encrypted data must be writable in memory, which depends on the operating system. Another method is to build the decrypted virus body on the stack. This way the encrypted data does not need to be writable. A third method is for the virus to allocate memory for the decrypted code and data. This technique has some disadvantages as non-encrypted code needs to allocate memory before the decryptor.

4.2 Oligomorphic Viruses

As long as the code of the decryptor is long enough and unique enough the detection of an encrypted virus is a simple task for the antivirus software. In order to challenge the antivirus software, virus writers invented new techniques to create mutated decryptors.

Oligomorphic viruses, as described in [7], change their decryptors in new generations, unlike encrypted viruses. One very simple technique is to have several decryptors instead of one. The Whale virus was the first virus to use this technique. It carried a few dozens of different decryptors and picked one randomly.

Win95/Memorial had the ability to built 96 different decryptor patterns, thus making detection based on the decryptor alone not practical. Most antivirus software tried to detect the virus by dynamic decryption of the virus body [7]. Memorial was the first Windows 95 virus with oligomorphic properties and was the first step towards Windows 95 polymorphism [30]. The code below is an example decryptor of the Memorial virus published in [7]:

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

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

;     Data area

Start:
;     encrypted/decrypted virus body
```

The first virus to use self-modifying encryption was Datacrime II. The decryption/encryption routine of the virus modified itself in order to prevent tracing through the decryption process using a debugger. It could also be described as an armored feature to prevent disassembly.

```
again:
mov       al,cs:[bx]  ; get next byte to be decrypted
mov       cs:[di],22h ; change the next instruction from xor al,dl to and al,dl
xor       al,dl       ; perform the decryption
ror       dl,1        ; rotate the key
mov       cs:[bx],al  ; store the decrypted byte
inc       bx          ; increment counter
mov       cs:[di],32h ; change the instruction back to an xor instruction
loop      again       ; until all bytes have been decrypted
```

The encryption method used in this virus is simple. Each byte is XOR-ed with a key which is rotated by one bit each time. All information about Datacrime II comes from [29].

4.3 Polymorphic Viruses

Polymorphism is the next step virus writers took to challenge antivirus software. The term polymorphic comes from the Greek words "poly," which means many, and "morhi," which means form. A polymorphic virus is a kind of virus that can take many forms. Polymorphic viruses can mutate their decryptors to a high number of different instances that take millions of different forms [7]. They use their mutation engine to create a new decryption routine each time they infect a program. The new decryption routine would have exactly the same functionality, but the sequence of instructions could be completely different [31].

The mutation engine also generates an encryption routine to encrypt the static code of the virus before it infects a new file. Then the virus appends the new decryption routine together with the encrypted virus body onto the targeted file. Since the virus body is encrypted and the decryption routine is different for each infection, antivirus scanners cannot detect the virus by using search strings. Mutation engines are very complex programs – usually far more sophisticated than their accompanying viruses. Some of the more sophisticated mutation engines can generate several billions of different decryption routines [31].

4.3.1 The 1260 virus

The 1260 virus was the first known polymorphic virus. It used an encryption routine which represents a significant development in encryption techniques. The encryption routine inserts various one-byte and two-byte garbage instructions between the functional instructions. The garbage instructions have no effect on the decryption process, but make the extraction of a search string virtually impossible [32]. Some of the garbage instructions used by 1260 include:

```
nop
dec bx
xor bx, cx
inc si
clc
```

The virus randomly selects a variable number of the garbage instructions and inserts them between the actual decoding instructions. It can also change the order of the decoding instructions, without affecting the execution of the program. The result is that the longest sequence of bytes present in all infected programs is only three bytes long, which is too short for the extraction of search strings [32].

4.3.2 The Dark Avenger Mutation Engine (MtE)

The Mutation Engine or MtE was the first polymorphic engine and was released in 1991. It was written by the Bulgarian virus writer nicknamed Dark Avenger and was released as a help to novice virus writers to use it with their own viruses. The following description of MtE comes from [7].

MtE makes a function call to the mutation engine function and pass control parameters in predefined registers. The engine builds a polymorphic shell around the simple virus inside it. The parameters of the engine include:

- A work segment
- A pointer to the code to encrypt
- Length of the virus body
- Base of the decryptor
- Entry-point address of the host
- Target location of encrypted code
- Size of decryptor
- Bit field of registers to use

The output of MtE is a polymorphic decryption routine with an encrypted virus body in the supplied buffer. The following is an example decryptor generated by MtE [7].

```
mov     bp,A16C       ; This Block initializes BP

                      ; to "Start"-delta
mov     cl,03         ; (delta is 0x0D2B in this example)
ror     bp,cl
mov     cx,bp
mov     bp,856E
or      bp,740F
mov     si,bp
mov     bp,3B92
add     bp,si
xor     bp,cx
sub     bp,B10C       ; Huh ... finally BP is set, but remains an
                      ; obfuscated pointer to encrypted body
```

```
Decrypt:
mov     bx,[bp+0D2B] ; pick next word
                     ; (first time at "Start")
add     bx,9D64      ; decrypt it
xchg    [bp+0D2B],bx ; put decrypted value to place
mov     bx,8F31      ; this block increments BP by 2
sub     bx,bp
mov     bp,8F33
sub     bp,bx        ; and controls the length of decryption
jnz     Decrypt      ; are all bytes decrypted?

Start:
      ; encrypted/decrypted virus body
```

There in only one constant byte in an MtE decryptor, followed by a negative offset, but it is placed at a variable location each time. Fortunately, MtE had a couple of minor limitations that enabled detection of the virus reliably using an instruction size disassembler and a state machine. However, MtE kept most antivirus companies to the workbench to introduce a virtual machine for the use of the scanning engine.

MtE was followed by other similar engines and today hundreds of polymorphic engines are known – most of them only used to create a couple of viruses. Although mutation engines create different looking viruses each time, there is no true randomness and a reliable signature can be calculated even in random-looking code [8].

4.3.3 Polymorphic Viruses for Windows

The first polymorphic viruses written for the 32-bit Windows environment, were Win95/HPS and Win95/Marburg – both written by the Spanish virus writer GriYo in 1998.

HPSs' polymorphic engine is very powerful and advanced. It supports subroutines using CALL/RET instructions and conditional jumps with non-zero displacement. The polymorphic engine has random byte-based blocks inserted between the generated code chains of the decryptor. The full decryptor is built only during the first initialization phase, thus an infected PC must be rebooted in order for the virus to create a new decryptor. This cause problems to antivirus vendors because they cannot test their scanners' detection rate efficiently. The decryptor consists of Intel 386 instructions and the virus body is encrypted and decrypted using different methods, including XOR/NOT and INC/DEC/SUB/ADD instructions with 8, 16, or 32 -bit keys, respectively. All information about HPS were taken from [7].

Marburg is probably the first polymorphic Windows 9x virus and it is written in assembler. Its polymorphic engine is similar to that of HPS, but it does not hook system functions; instead, Marburg is a direct action infector. Marburg utilizes a slow polymorphic replication mechanism and the infection method differs slightly in some files. When there are no relocations for the first 255 bytes from the entry-point, the virus places a jump instruction in the code at the entry-point of the host, then builds a random garbage code block first and puts the jump to the polymorphic decryptor at the end of it. The polymorphic decryptor then decrypts the virus body that proceeds. The above description of Marburg was published in [33].

The code below is an illustration of a Win95/Marburg decryptor instance, which was published in [7] along with its description:

```
Start:
                          ; Encrypted/Decrypted Virus body is placed here

Routine-6:
dec     esi               ; decrement loop counter
ret

Routine-3:
mov     esi,439FE661h     ; set loop counter in ESI
ret

Routine-4:
xor     byte ptr [edi],6F ; decrypt with a constant byte
ret

Routine-5:
add     edi,0001h         ; point to next byte to decrypt
ret

Decryptor_Start:
call    Routine-1         ; set EDI to "Start"
call    Routine-3         ; set loop counter

Decrypt:
call    Routine-4         ; decrypt
call    Routine-5         ; get next
call    Routine-6         ; decrement loop register
cmp     esi,439FD271h     ; is everything decrypted?
jnz     Decrypt           ; not yet, continue to decrypt
jmp     Start             ; jump to decrypted start

Routine-1:
call    Routine-2         ; Call to POP trick!
```

```
Routine-2:
pop     edi
sub     edi,143Ah         ; EDI points to "Start"
ret
```

The polymorphic decryptor of the virus is placed after the encrypted virus body and it is split between small parts of code routines, which can appear in mixed order. The result of the above decryptor is millions of possible code patterns filled with random garbage instruction between the parts of code.

Some polymorphic viruses, such as Win95/Coke, use multiple layers of encryption. Other more sophisticated polymorphic engines use an RDA-based decryptor that implements a brute-force attack against its constant – but variably encrypted – virus body. Manual analysis of such viruses can be a daunting task. However, all polymorphic viruses have a constant body which must be decrypted before the virus can function. Advanced methods exists that can decrypt the virus body and identify the virus [7].

Chapter 5

Metamorphic Viruses

This chapter explains in detail what a metamorphic virus is and describes in detail some of the techniques these viruses use to avoid detection. Next, Win95/Zmist and {Win32,Linux}/Simile, the two most advanced metamorphic viruses ever created are described in some more detail. This chapter concludes by summarizing the computer virus evolution process, from the simple virus to the metamorphic virus.

5.1 Introduction

Creating a polymorphic virus is a very complex and challenging task for virus writers. They often waste months on creating a new polymorphic virus, a virus that can take an antivirus vendor just a few hours to detect. The problem with polymorphic viruses is that they eventually have to decrypt themselves and present their constant body in memory in order to function. Advanced detection techniques can wait for the virus to decrypt its self and then detect it reliably.

The first known 32-bit virus that did not use a polymorphic decryptor was the Win32/Apparition virus. As described in [7], the virus carries its source code around and when it finds a compiler installed on a machine, it inserts and removes garbage code into its source code and re-compiles itself. This enables new generations of the virus to look completely different from the previous. This kind of virus is called metamorphic.

Figure 5.1 on the next page, shows the comparison of three kinds of viruses in a very simplified way. In basic viruses, the entry point code is modified to give execution control to the virus code. Detection is trivial using search strings. Polymorphic viruses apply encryption to their body to prevent detection using search strings. Some advanced metamorphic viruses re-program themselves with little pieces of viral code scattered and with garbage code in between [34]. Not all metamorphic viruses are capable of doing this, though.

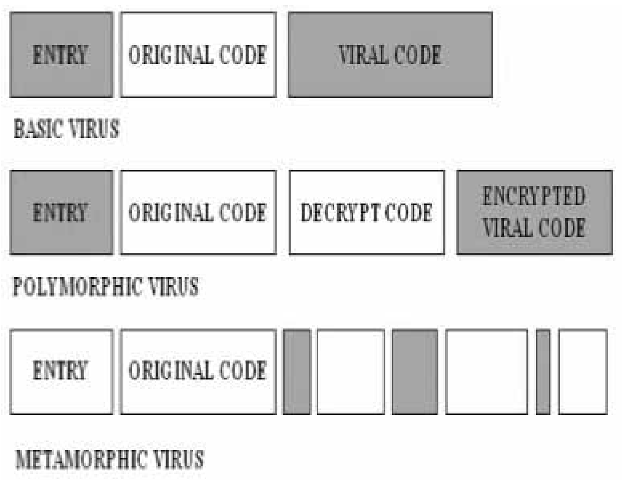


Figure 5.1: Three kinds of viruses (from [34])

5.2 The Metamorphic Virus

Metamorphic viruses transform their code as they propagate, thus evading detection by static signature-based virus scanners and have the potential to lead to a breed of malicious programs that are virtually undetectable statistically [35]. These viruses also use code obfuscation techniques to challenge deeper static analysis and can also beat dynamic analyzers, such as emulators, by altering their behavior when they detect that they are executing under a controlled environment [36].

The main goal of metamorphism is to change the appearance of the virus while keeping its functionality. To achieve this, metamorphic viruses use several metamorphic transformations, such as register usage exchange, code permutation, code expansion, code shrinking, and garbage code insertion [35].

Metamorphic viruses, as described in [7], do not have a decryptor, neither a constant virus body like polymorphic viruses do. However, they are able to create new generation that look different. They do not use a data area filled with string constants but have one single-code body that carries data as code. Metamorphic viruses usually avoid creating new generations that look very similar to their parents. Figure 5.2 on the following page, which was given in [23], illustrates the changes in different generations of a metamorphic virus. As mentioned before, the shape changes but the functionality remains the same.

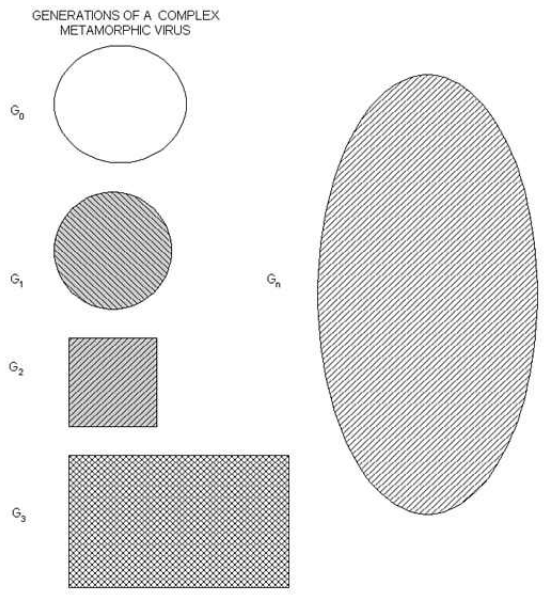


Figure 5.2: Metamorphic virus generations (from [23])

5.2.1 A formal definition

The following formal definition of a metamorphic virus was given in [37]. Let $\Phi_P(d,p)$ denote a function computed by a computer program P in the running environment *(d,p)* where d represents data and p represents programs stored on computers. *D(d,p)* and *S(p)* are two recursive functions. *T(d,p)* is called injury condition (trigger) and *I(d,p)* is called infection condition.

Definition 4. The pair (υ,υ') of two different total recursive functions υ and υ' is called a metamorphic virus if for all χ, $(\upsilon$,$\upsilon')$ satisfies

$$\Phi_{\upsilon(\chi)}(d,p) = \begin{cases} D(d,p) & , \ if\ T(d,p) \\ \Phi_\chi(d,p[\upsilon'(S(p))]) & , \ if\ I(d,p) \\ Phi_\chi(d,p) & , \ otherwise \end{cases}$$

and

$$\Phi_{\upsilon'(\chi)}(d,p) = \begin{cases} D'(d,p) & , \ if\ T'(d,p) \\ \Phi_\chi(d,p[\upsilon'(S'(p))]) & , \ if\ I'(d,p) \\ Phi_\chi(d,p) & , \ otherwise \end{cases}$$

where T(d,p) (resp.,I(d,p), D(d,p), S(d,p)) is different from $T'(d,p)$ (resp.,$I'(d,p)$, $D'(d,p)$, $S'(d,p)$).

A metamorphic virus (υ,υ') seems to combine two different viruses, υ and υ'. However, when υ infects a program, the program is infected by υ', and vise versa. The main difference between a metamorphic and a polymorphic virus is that each form of a polymorphic virus has the same kernel, but each each component of a metamorphic virus has its own kernel [23].

5.2.2 Anatomy of a Metamorphic Virus

This model of the anatomy of a prototypical closed-world[1], binary-transforming[2] metamorphic engine presented in this section, along with its explanation, was published by Walenstein, Mathur, Chouchane, and Lakhotia in [38]. The anatomy of a metamorphic engine identifies functional units, which are conceptual units of tasks that must be performed by the engine, but are not required to be implemented as modules or functions. Figure 5.3, illustrates the units of this model.

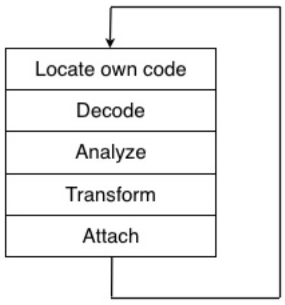


Figure 5.3: Anatomy of a metamorphic engine (from [38])

Locate own code. Each time a metamorphic engine is called to transform some code, it must be able to locate. Metamorphic viruses which transform both their own code and the code of their host, must be able to locate their own code in the new variants.

Decode. Next, the engine needs to decode the information required to perform the transformations. In order to transform itself, the engine must have some representation of itself so that it knows how to make the transformations. The metamorphic engine may also need to decode other types of information required for analysis or transformation. This information can be encoded in the virus body, in the data segments, or in the code itself. Unpacking or decrypting the whole executable is not a responsibility of this unit.

[1]Malware that are self contained.
[2]Malware that transform the binary image that is executed.

Analyze. In order for the metamorphic transformations to work correctly, certain information must be available. For some transformations to be performed correctly, the engine must have register liveness information available. If such information is not available, the metamorphic engine itself must construct it. Register liveness can be constructed, in part, by a *def-use* analysis. A piece of information that is frequently required for analysis and transformation, is the control flow graph (CFG) of the program. For example, it can be used to rewrite the control flow logic of a program if a transformation expands the code.

Transform. This unit is responsible for transforming the code into equivalent code. This is done usually by replacing instruction blocks in the code with other equivalent. More details on transformations are given later in this chapter.

Attach. The last step that the metamorphic engine has to take, is to attach the new generation of the virus to a new host file.

The units are ordered according to the direction of information flow, which is not necessarily the execution order. The feedback loop shows that the output of a metamorphic engine may become its input in the next generation. Lakhotia and Kapoor characterize this loop as the Achilles' heel of a metamorphic virus [36]. More on this in Chapter 6.

5.2.3 The Metamorphic Virus According to a Virus Writer

The following description is given by the virus writer Benny, member of 29A[3]. The description appeared in an article in 29A magazine, issue 4. It is a description of what a metamorphic virus should be, according to a real virus writer. It also demonstrates how challenging is the task of creating a metamorphic virus.

Benny says that coding a metamorphic virus is really hard. He believes that every good metamorphic engine should contain:

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

[3]A group of notorious virus writers. Former members of 29A include Zombie, Vecna, and "The Mental Driller."

7. Cleaner

The *internal disassembler* will disassemble the code, instruction by instruction. The *opcode shrinker* will shrink or optimize two, or more instructions to one. The *opcode expander* will expand one instruction to more than one. The *opcode swapper* will swap two, or more instructions. The *relocator/recalculator* will relocate or recalculate all relative references, such as jumps, calls, and pointers. The *garbager* will insert one, or more do-nothing instructions between real code. The *cleaner* will clean garbage code inserted by the garbager.

According to Benny, the metamorphic engine should know every opcode and should be able to process instructions concurrently to provide shrinking and swapping, something which is much harder than expanding. Also, the re-calculator for all relative references, would be very hard to code. Jumps and calls to known code will be easy, but jumps and calls to unknown (not yet mutated) code will be harder. The virus writer would probably need to mark the opcode and redundantly check if jumps and calls can be recalculated. The relocation of pointers will be solved by user defined tables, and as Benny says, it is the ugliest effect as the virus writer will have to mark down all pointers. Also, the garbager should only use a limited set of instructions, because the garbage shouldn't affect the other instructions. Finally, the opcode swapper should be able to analyze every instruction and test if its action will not effect the second instruction.

Benny continues by saying that there are still more problems and limitations. The virus writer will not be able to mix variables with code, and they will need to keep somewhere some original virus code or recalculate all values in the pointer table at run-time. Benny claims that he does not know how to do this. His last words should give the point:

> "Yeah, I said it will be hard. And have I ever said to you, that the tiniest metas (metamorphic engines) are about 20KB?"

5.3 Metamorphic Techniques

To avoid detection, metamorphic viruses use several different techniques to evolve their code into new generations that look completely different, but have exactly the same functionality. This sections describes in detail many of these techniques.

5.3.1 Garbage Code Insertion

Garbage code (or junk code) insertion is a simple technique used by many metamorphic and polymorphic viruses to evolve their code. The idea behind this technique is to make their code look different so that no usable hexadecimal search string can be extracted. The instructions inserted into the code are called garbage because they have no impact on the functionality of the code.

The Win32/Evol virus, which appeared in July 2000, implemented a metamorphic engine that was able to run on any major Win32 platform. One of the functionalities of Evols' metamorphic engine was to insert garbage between core instructions. The following piece of code is an early generation of Evol:

```
C7060F000055    mov [esi], 5500000Fh
C746048BEC5151  mov [esi+0004], 5151EC8Bh
```

The next piece of code is a later generation of Evol:

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

The previous two pieces of code, which were published in [39], look different but have the same exact functionality. The instructions commented as garbage have no impact on the functionality of the code. It is clear that no usable hexadecimal string can be extracted from the previous two instances of the Evol virus.

A more advanced form of garbage code insertion is performed by the Win95/Bistro virus, which appeared in October 2000. A routine which is activated on random, creates a "do-nothing" code block at the entry-point of the virus body. When is activated the code block can generate millions of iterations to challenge the speed of the emulator [39].

5.3.2 Register usage exchange

Another simple technique used by metamorphic viruses is register usage exchange. This method was used by the Win95/Regswap virus, which was created by the virus writer Vecna and released in 1998. Different generations of the virus will use the same code but with different registers. The following two pieces of code belong to two different generations of the Regswap virus:

```
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

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

It is obvious that the complexity of the virus is not high and the different generations have enough common area to enable detection using wildcard strings. However, many antivirus scanners do not support wildcard strings, thus a virus like this might need algorithmic detection. The information about Regswap and register usage exchange comes from [23].

5.3.3 Permutation Techniques

The Win32/Ghost and the Win95/Zperm viruses introduced a new level of metamorphism. Although the virus code is constant, metamorphosis is achieved by dividing the code into frames, and then position the frames randomly and connect them by branch instructions to maintain the process flow. The branch instructions could be simple jump instructions or a complex transfer of control, such as “push val32; ret.” The flow of control always remains the same [40].

The Win32/Ghost virus, which was discovered in May 2000, had the ability to re-order its subroutines from generation to generation. If the number of subroutines is n, then the number of different virus generations is $n!$. The Win32/Ghost had 10 subroutines, thus there were 3628800 different possible virus generations. Figure 5.4 on the following page, illustrates how a virus re-orders its modules. BadBoy was a DOS virus that also used the same permutation technique. It had 8 subroutines so the number of possible generations was 40320.

5.3.4 Insertion of Jump Instructions

Another method used by some metamorphic viruses to create new generations is inserting jump instructions within its code. The Win95/Zperm virus is a very good example of this technique. It appeared in June and

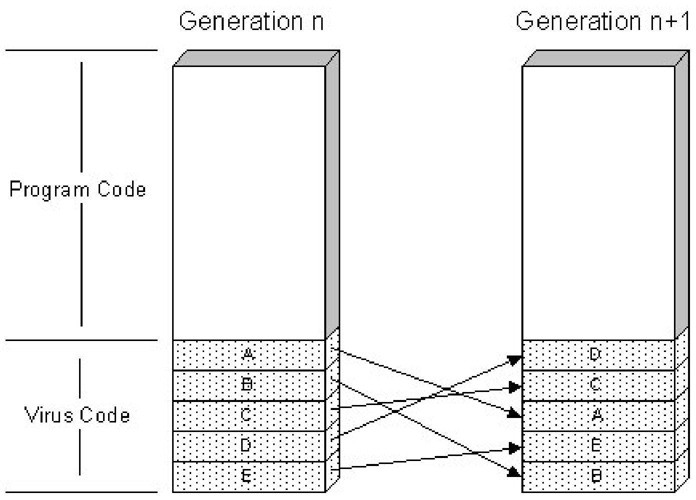


Figure 5.4: Illustration of module re-ordering (from [41])

September of 2000 and was written by the same person who created the Win95/Zmorph [23]. The virus inserts and removes jump instructions within its code and each jump instruction will point to a new instruction of the virus.

Zperm is permutating its own code each time it infects a Portable Executable (PE) file. In a first generation the virus may look like this:

```
instruction 1     ; entry point
instruction 2

.
.
.

instruction n
```

In later generations the virus changes itself by inserting a random number of jump instructions. A later generation might look like this:

```
instruction 2
jmp instruction 3
instruction 1     ; entry point
jmp instruction 2
instruction 3
jmp instruction n
```

The previous description was published in [39]. A good illustration of this is given by figure 5.5 on the next page. Zperm never generates a constant body anywhere, not even in memory, so detection of the virus using search strings is virtually impossible.

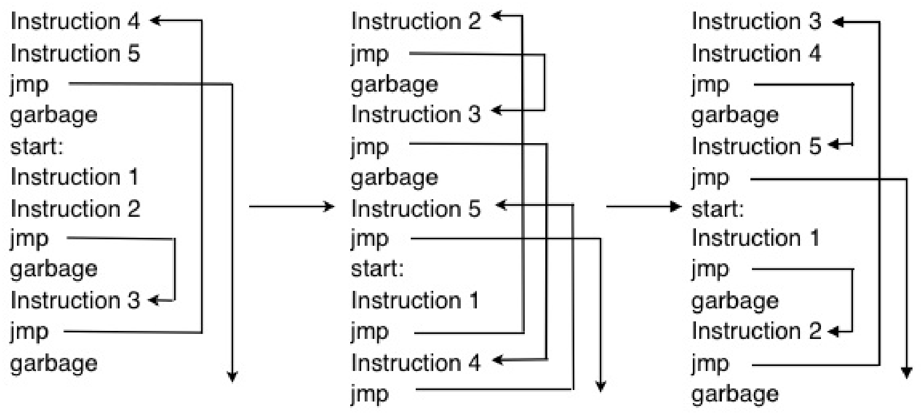


Figure 5.5: Example of Zperm inserting jumps into its code (from [23])

5.3.5 Instruction Replacement

Some metamorphic viruses are able to replace some of their instructions with other equivalent instructions. In addition to jump insertions, Win95/Zperm had the ability to perform instruction replacement. For example, the virus could replace the instruction "xor eax, eax" with the instruction "sub eax, eax." Both instructions perform the same function – zeroing the content of the eax register – but have a different opcode [23].

Another example of instruction replacement is the Win95/Zmist virus. The types of instruction replacement that can be performed by Zmist, as described in [42], include:

- reversing of branch conditions
- register moves replaced by push/pop sequences
- alternative opcode encoding
- xor/sub and or/test interchanging

Win95/Bistro performs similar replacements also. Here is the original code of a target before processing by Bistro:

```
55          push    ebp
8BEC        mov     ebp, esp
8B7608      mov     esi, dword ptr [ebp + 08]
85F6        test    esi, esi
743B        je      401045
8B7E0C      mov     edi, dword ptr [ebp + 0c]
09FF        or      edi, edi
7434        je      401045
31D2        xor     edx, edx
```

And after:

```
55          push    ebp
54          push    esp             ; register move replaced by push/pop
5D          pop     ebp             ; register move replaced by push/pop
8B7608      mov     esi, dword ptr [ebp + 08]
09F6        or      esi, esi        ; test/or interchange
743B        je      401045
8B7E0C      mov     edi, dword ptr [ebp + 0c]
85FF        test    edi, edi        ; test/or interchange
7434        je      401045
28D2        sub     edx, edx        ; xor/sub interchange
```

The previous example code comes from [7].

5.3.6 Host Code Mutation

The Win95/Bistro virus not only mutates itself in new generations, but it also mutates the code of its host. This way the virus can generate new viruses and worms. To do this, the virus uses a randomly executed code-morphing routine. Also, because the entry-point code of the application could be different, disinfection cannot be done perfectly.

The code-morphing routine of Bistro uses techniques previously described in this section. Code permutations of worms and viruses, as done by Bistro, would be very difficult to deal with. If similar morphing techniques would be implemented by a 32-bit worm, a major problem would occur as new mutations of old viruses and worms would be created endlessly. Information about Win95/Bistro and host code mutation were taken from [39].

5.3.7 Code Integration

The Win95/Zmist virus implemented an even more sophisticated technique. This technique, named code integration, has never been seen in any previous virus.

Zmists' engine can decompile Portable Executable (PE) files to their smallest elements, requiring 32MB of memory. Then the virus moves code blocks out of the way, inserts itself into the code, re-generates code and data references, and rebuilds the executable [23]. This way the virus can integrate itself seamlessly to the code of its target, making it very hard to detect and even harder to repair. More details about Win95/Zmist can be found in section 5.4.1 on the following page.

5.4 Advanced Metamorphic Viruses

This section describes in more detail two of the most advanced metamorphic viruses, Win95/Zmist and {Win32, Linux}/Simile. Zmist was created by the virus writer Z0mbie and released in 2000. Simile – named MetaPHOR by its creator – was created by “The Mental Driller” and was released in 2002.

5.4.1 Win95/Zmist

The Russian virus writer Z0mbie released Win95/Zmist in 2000, along with his “Total Zombification” magazine. Z0mbie is the author of many other polymorphic and metamorphic viruses, including Win95/Zmorph and Win95/Zperm. All information about Zmist in this section, were taken from [42].

At the time of its release, Zmist was one of the most complex viruses. Peter Ferrie and Peter Szor went as far as to call Zmist “one of the most complex binary viruses ever written.” Zmist is a Entry-Point Obscuring (EPO) virus that is metamorphic. In addition, it randomly uses a polymorphic decryptor.

As described in section 5.3.7 on the previous page, Zmist supports the unique technique called *code integration*. Also, it occasionally inserts jump instructions after every single instruction of the code section, each pointing to the next instruction. The fact that these extremely modified applications work – from generation to generation – was not expected by anyone, not even by Z0mbie. In [42] it is mentioned that “due to its extreme camouflage, Zmist is clearly the perfect anti-heuristics virus.”

Initialization

Zmist merges itself with the existing code of the host and becomes part of the instruction flow. It does not change the entry-point of its target. Because the virus code is inserted in random locations, sometimes the virus never receives control. If it does execute, it immediately launches the host as a separate process and hides the original process until the infection routine completes. In order to hide the original process, the RegisterServiceProcess() API must be supported on the current platform. Concurrently, the virus begins to search for new targets.

Direct Action Infection

After Zmist launches the host process, it checks if there are at least 16MB of physical memory installed and that the virus is not running in console mode. If the checks pass, it allocates several memory blocks, including a 32MB memory block for its engine, *Mistfall*. Then it permutates its body and begins a recursive search for PE .EXE files. The search takes place in the Windows directory and all subdirectories, then the directories referred to by the PATH environment variable, and finally all fixed or remote drives from A to Z.

Permutation

The permutation is done only once per machine infection. It includes instruction replacement, register moves replaced by push/pop sequences, alternative opcode encoding, xor/sub and or/test interchanging, and garbage instruction generation. Zmists' engine, Real Permutation Engine (RPME), was used in Win95/Zperm virus too.

Infection of Portable Executable files

For a file to be infectable it must:

- be smaller than 448KB,
- begin with "MZ",
- be a PE file, and
- be not previously infected

As a marker for infected files the virus uses "Z" at offset 0x1C in the MZ header. The virus reads the file into memory and chooses one out of three infection types. There is a one in ten chance that the virus will only insert jump instructions between existing instructions, and will not infect the file. With the same probability the virus will infect the file with an unencrypted copy of itself. Otherwise, it will infect the file by a polymorphically encrypted copy.

Structured Exception Handling protects the infection process from crashing due to errors. After rebuilding the executable, the virus erases the original file and replaces it with the infected one. In case of errors during the file creation, the original file is lost and nothing replaces it. The polymorphic decryptor consists of islands of code that are integrated into random locations throughout the host code section, which are linked together with jumps. The decryptor integration uses exactly the same method as the virus body integration.

For decrypting the virus code, Zmist uses an anti-heuristics trick. Instead of making the section writable in order to alter its code directly, the host is required to have – as one of the first three sections – a section containing writable and initialized data. The virtual size of this section is increased by 32KB, which allows the virus to decrypt code directly into the data sections and transfer control to there. If the virus cannot find such a section, it infects the file without using encryption. There are four ways for the decryptor to receive control:

- via absolute indirect call (0xFF 0x15)
- a relative call (0xE8)
- a relative jmp (0xE9)
- as part of the instruction flow

For the first three methods, control will be transferred soon after the entry point. For the last method, the virus inserts an island of the decryptor code into the middle of a subroutine, somewhere in the code. All used registers are saved before decryption and restored afterwards, so that the original code will not change its behavior.

If the virus use encryption, the code is encrypted with ADD/ SUB/ XOR using a random key. Then the key is modified on each iteration by ADD/ SUB/ XOR using a second random key. Zmist uses the Executable Trash Generator (ETG) to produce and add various garbage instructions in between the decryption instructions, using a random number of registers and a random choice of loop instructions.

Code Integration

In order to distinguish between offsets and constants, the integration algorithm requires that the host has fix-ups. The algorithm also requires that the name of each section in the host is one of the following: CODE, DATA, AUTO, BSS, TLS, .bss, .tls, .CTR, .INIT, .text, .data, .rsrc, .reloc, .idata, .rdata, .edata, .debug, and DGROUP. The most common compilers and assemblers use these section names. The strings are encrypted, thus these names are not visible in the virus code.

The virus allocates a block of memory which is equivalent to the size of the hosts' memory image, and each section is loaded into this array at the sections' relative virtual address. In order to rebuilt the executable, the virus saves the locations of every interesting virtual address and then begins the instruction parsing. When an instruction is inserted into the code all following code and data references must be updated. If some of these references are branch destinations, sometimes the size of them will increase as a result of the modification. When this occurs, more code and data references must be updated, some of which might be branch destinations, and the cycle repeats. However, this regression is not infinite.

During the instruction parsing, the type and length of each instruction is identified and the types are described using flags. In cases where an instruction cannot be resolved in an unambiguous manner to either code or data, the virus does not infect the file. After the parsing is completed, the virus calls the mutation engine. The engine inserts jump instructions after every instruction, or generates a decryptor and inserts the islands into the file. Then the virus rebuilt the file, updates the relocation information, re-calculates the offsets and restores the file checksum. Any overlay data appended to the original file are copied to the new file.

5.4.2 {Win32, Linux}/Simile

In March 2002, a virus writer who calls himself “The Mental Driller,” released the Win32/Simile virus. The particular virus writer had challenged the antivirus community in the past with some of his previous creations, such as Win95/Drill. Unless explicitly stated, all information in this section comes from [43].

Simile, which is even more complex than Zmist, is approximately 14,000 lines of assembly code. Its extremely powerful and complex metamorphic engine takes up about 90% of the virus code. His creator named the virus “MetaPHOR,” which stands for Metaphoric Permutating High-Obfuscating Re-assembler. There are four known variants of the virus, three of them (variants A, B, and D) written by the original author, and one (variant C) written by an unknown author [7].

Simile is very obfuscated and very difficult to understand. It attacks the disassembling, debugging, and emulation techniques. It also challenges the standard evaluation-based techniques for virus analysis. Just like Zmist, Simile makes use of EPO techniques.

Replication Routine

Similes’ replication routine is basic, using a direct action replication mechanism that attacks PE files on the local machine and the network. The virus gives much more emphasis on its, – unusually complex – metamorphic engine.

EPO Mechanism

The virus does not alter the entry point of the file. Rather, it searches for and replaces all of the possible patterns of certain call instructions, which reference ExitProcess() API calls, to point to the beginning of the virus. In order to further confuse the location of the virus body, in some cases the body is placed together with the polymorphic decryptor at the same location, while in other cases the polymorphic decryptor is placed at the end of the code and the virus body is placed at another location.

Polymorphic Decryptor

When an infected program is executed and the instruction flow reaches one of the hooks that the virus has placed in the code section, control is transferred to a polymorphic decryptor, whose location is variable. If the virus body is encrypted, the decryptor decodes it; otherwise, it directly copies it. The decryptor allocates a block of memory of about 3.5MB, then decrypts the encrypted virus body into it. In order to do this, it processes the encrypted data in a seemingly random order, thus it avoids triggering some of the decryption-loop recognition heuristics. This method is very unusual. The virus writer calls this method "Pseudo-Random Index Decryption," and it relies on the use of a family of functions that have interesting arithmetic properties modulo 2^n.

The size and shape of the decryptor varies greatly from generation to generation. To achieve this level of variability, the creator of the virus uses his metamorphic engine to transform code templates into working decryptors.

In some cases, the decryptor may start with a header whose purpose is to generate anti-emulation code on the fly. It contains a small oligomorphic code snippet containing the instruction "ReaD Time Stamp Counter" (RDTSC). The instruction retrieves the current value of an internal processor ticks counter, then the decryptor uses the value of this random bit to either decode and execute the virus body or bypass the decryption logic and exit. This method is used to confuse emulators that do not support the RDTSC instruction, and also attack all algorithms that rely on emulation to either decrypt the virus body or use heuristics to determine viral behavior.

When the virus is initially executed, it retrieves the addresses of 20 APIs required for replication and payload execution. Then, it checks the current date to determine whether either of its payloads should be activated.

Metamorphism

After the payload check, the virus generates a new body in the following steps:

1. Disassembles the viral code into an intermediate form, which is independent of the CPU on which the native code executes. This allows for future extensions, such as producing code for different operating systems or even different CPUs.

2. Shrinks the intermediate form by removing redundant and unused instructions. These instructions were added by earlier replications to interfere with disassembly by virus researchers.

3. Permutates the intermediate form by reordering subroutines or separating blocks of code and linking them with jump instructions.

4. Expands the code by adding redundant and unused instructions.
5. Re-assembles the intermediate form into a final, native form that will be added to infected files.

Most first generation metamorphic viruses could only expand. Simile can both expand and shrink to different forms. Simile.D is capable of translating itself into different metamorphic forms (V1, V2 . . . Vn) and does so on more that one operating systems (O1,O2 . . . On) [7]. The power of Similes' engine is demonstrated in the following code, which was published in [41]:

```
mov     dword_1, 0h
mov     edx, dword_1
mov     dword_2, edx
mov     ebx, dword_2
mov     edi, 32336C65h
lea     eax, [edi]
mov     esi, 0A624548h
or      esi, 4670214Bh
lea     edi, [eax]
mov     dword_4, edi
mov     edx, ebp
mov     dword_5, edx
mov     dword_3, esi
mov     edx, offset dword_3
push    edx
mov    dword_6, offset GetModuleHandleA
push   dword_6
pop     dword_7
mov     edx, dword_7
call    dword ptr ds:0[edx]
```

Similes' metamorphic engine could replace the previous code by the following five lines:

```
mov     dword_3, 6E72654Bh
mov     dword_4, 32336C65h
mov     dword_5, 0h
push    offset dword_3
call    ds:[GetModuleHandleA]
```

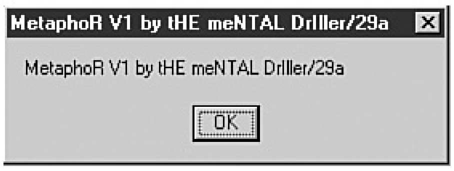
Replication

The next phase is replication. The virus begins searching for *.exe in the current directory, and then in all fixed and mapped network drives. The infection routine only scans to a depth of three subdirectories and completely avoids any directories that begin with the letter “W”. There is a 50% chance that each discovered file will be skipped explicitly. Files that begin with “F-”, “PA”, “SC”, “DR”, “NO”, or contain the letter “V” anywhere in the name will also be skipped. Some other character combinations are skipped unintentionally.

If a file cannot be infected safely, it is filtered out by the infection routine. For infectable files, random factors and the file structure will determine where the virus places the decryptor and the virus body. If the file contains no relocations – or by small chance – the virus body will be appended to the last section of the file. The decryptor will be placed either immediately before the virus body or at the end of the code section. If the name of the last section is “.reloc”, the virus will insert itself at the beginning of the data section.

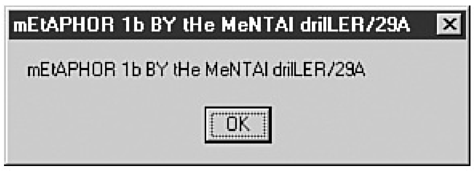
Payload

The first payload activates only during March, June, and September. Variants A and B display their message on the 17th day of these months and variant C on the 18th. Simile.A displays the following message:


MetaphoR V1 by tHE meNTAL DrIller/29a

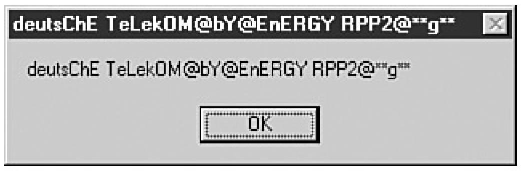
MetaphoR V1 by tHE meNTAL DrIller/29a

Simile.B displays the following message:


mEtAPHOR 1b BY tHe MeNTAl driLER/29A

mEtAPHOR 1b BY tHe MeNTAl driLER/29A

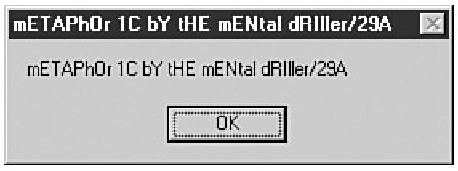
Simile.C attempts to display “Deutsche Telekom vy Energy 2002**” but due to the authors little understanding of the code, the message rarely appears correctly:


deutsChE TeLekOM@bY@EnERGY RPP2@**g**

deutsChE TeLekOM@bY@EnERGY RPP2@**g**

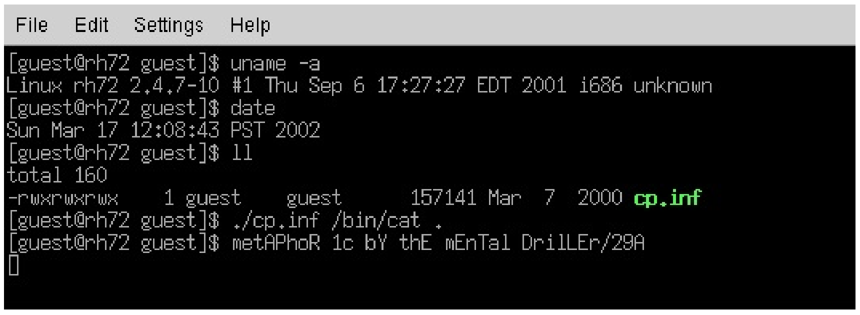
In variants A and B, the second payload activates on the 14th of May and displays the message “Free Palestine!” on computers that use the Hebrew locale. In variant C, the second payload activates on the 14th of July and attempts to display the message “Heavy Good Code!”, but due to a bug the message is only displayed on systems that the locale cannot be determined.

Simile.D used the first Win32/Linux cross-infector {Win32,Linx}/Peelf, which uses two separate routines to carry out the infection on PE and ELF[4] files. The virus was confirmed to infect files under Red Hat Linux versions 6.2, 7.0, and 7.2, and it is likely that it can infect under most other common linux distributions. Information about Simile variant D were taken from [44]. For Simile.D, if the virus host is a PE file, the following message will be displayed:


mETAPhOr 1C bY tHE mENtal dRIller/29A

mETAPhOr 1C bY tHE mENtal dRIller/29A

If the virus host is an ELF file, the virus attempts to output the following text message to the console:



```
[guest@rh72 guest]$ uname -a
Linux rh72 2.4.7-10 #1 Thu Sep 6 17:27:27 EDT 2001 i686 unknown
[guest@rh72 guest]$ date
Sun Mar 17 12:08:43 PST 2002
[guest@rh72 guest]$ ll
total 160
-rwxrwxrwx    1 guest    guest      157141 Mar  7  2000 cp.inf
[guest@rh72 guest]$ ./cp.inf /bin/cat .
[guest@rh72 guest]$ metAPhoR 1c bY thE mEnTal DrilLEr/29A
```

5.5 The Virus Evolution: A Simple Comparison

According to Fred Cohen, a computer virus is a program that can infect other programs by modifying them to include a possibly evolved copy of itself. A simple computer virus have three components: the *replication*, the *payload*, and the *trigger*. The payload is responsible for performing any kind of malicious actions in the infected system. The trigger is responsible for checking if the desired conditions are met in order to deliver the payload. The previous two components depend on the virus and have no significant effect on the evolution of the computer virus.

[4]Executable and Linking Format (ELF) is a standard file format in Unix for executables, object code, shared libraries, and core dumps.

The most significant component of a computer virus and the one that has evolved through the years, is the replication component. The replication component is responsible for replicating the virus by infecting other programs. The first generation of computer viruses is a very simple one. This virus infects other files by modifying them and attaching itself to them – usually at the entry-point. Because the simple virus attaches the exact same copy of itself to all infected files, detection is an easy case for the antivirus scanners. The only thing the antivirus vendors have to do is get a copy of the virus, extract a hexadecimal search string from the virus code, and scan for that specific string. Figure 5.6, is a simple illustration of the simple virus (V) replication, from generation to generation.

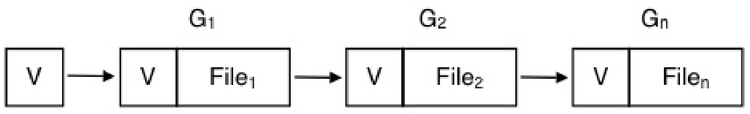


Figure 5.6: Simple virus replication

The encrypted virus carries a small encryption routine (decryptor), and encrypts and decrypts its constant body each time it infects a new file. This way the antivirus scanners cannot detect the virus body because it is encrypted – usually using a different encryption key each time. However, the encrypted virus still has a constant body under the encryption, and also has to carry the constant code of the decryptor. Although not a very good method to be used exclusively, antivirus scanners can detect the encrypted virus just by detecting its constant decryptor. Figure 5.7, illustrates how the encrypted virus replicates. The decryptor (D) is constant and behind the encryption the body of the virus remains constant too.

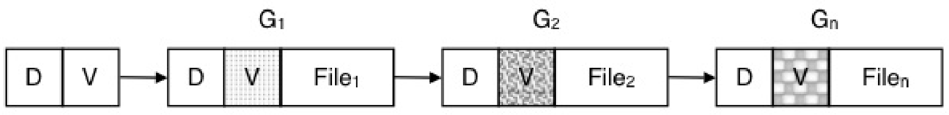


Figure 5.7: Encrypted virus replication

In order to solve this problem, the polymorphic virus use different methods to change the code of its decryptor, from generation to generation. This type of virus still has a constant – but encrypted – body and a decryptor, but each time the decryptor would change shape so that no search string can be extracted from its code, thus antivirus scanners cannot detect it using search strings. However, in order to function the virus has to decrypt itself and present its constant body somewhere in memory. Advanced antivirus techniques can decrypt the virus body and identify the virus. Figure 5.8

on the following page, illustrates how the polymorphic virus replicates. The decryptor (D) changes shape from generation to generation, but behind the encryption there is still a constant virus body.

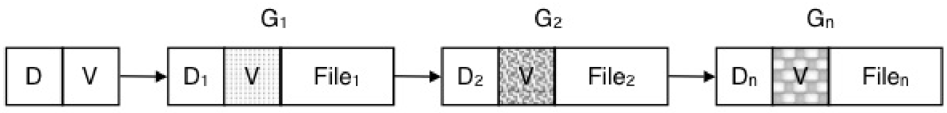


Figure 5.8: Polymorphic virus replication

The metamorphic virus uses no encryption – with some exceptions – to hide its code. In fact, an advanced metamorphic virus has no constant data anywhere between generations; new generations look completely different. Simply speaking, this virus changes its shape every time it infects a new file or a new system, while preserving its functionality. No hexadecimal search strings can be extracted from it, thus detection using strings is virtually impossible. Figure 5.9, illustrates the replication of a metamorphic virus. It is obvious that no constant data exists between different generations.

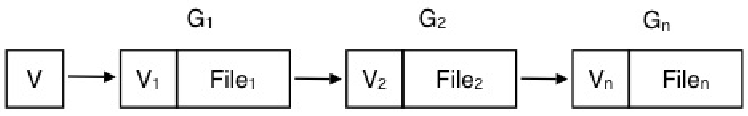


Figure 5.9: Metamorphic virus replication

The previous illustrations are simplified. Not all types of viruses can be visualised in the previous ways. For example, EPO viruses – can be encrypted, metamorphic, or metamorphic – do not attach themselves on the entry-point of the host file as shown above, but somewhere between the code of the host file.

Chapter 6

Metamorphic Virus Detection

Metamorphic techniques make virus detection using search strings virtually impossible. To detect a metamorphic virus, techniques such as examination of the file structure, or analysis of the behavior of the code must be used. For perfect detection of a metamorphic virus, detection routines must be written that can generate the essential instruction set of the virus body from the actual instance of the infection [7].

This Chapter discusses the weak points of metamorphic viruses and describes several techniques for their detection. Some of the techniques are used in commercial antivirus products and others are experimental methods proposed by academic papers.

6.1 The Weakness of Metamorphic Viruses

Metamorphic viruses transform their code as they propagate, in order to avoid detection by static signature-based virus scanners and use code obfuscation techniques to challenge deeper static analysis. They can also beat dynamic analysers, such as emulators, by altering their behavior when they detect that they are executing under a controlled environment [36].

Metamorphic viruses are difficult to detect because their creators have the advantage of knowing the weaknesses of antivirus scanners [36]. The limits of antivirus scanners come from the limits of static and dynamic analysis techniques. The following techniques are used by virus writers to attack the various stages in the analysis of binaries [45]:

- Attacks on disassembly
- Attacks on procedure abstraction
- Attacks on control flow graph generation

- Attacks on data flow analysis
- Attacks on property verification

Lakhotia, Kapoor, and Kumar believe that antivirus technologies could counter-attack using the same techniques that metamorphic virus writers use; identify similar weak spots in metamorphic viruses [36]. The authors of [45], explain that the virus writers face the same theoretical limits as antivirus technologies and this could be used to the advantage of antivirus researchers.

As explained in [36], in order to mutate their code generation after generation, metamorphic viruses have to analyse their own code, thus they too face the limits of static and dynamic analysis. Furthermore, a metamorphic virus have to be able to reanalyse the mutated code that it generates. This means that the complexity of transformations in the previous generations have a direct impact on how a virus analyses and transforms code in the current generation. Thus, metamorphic viruses need to use some coding conversions, or develop special algorithms that will help them to detect their own obfuscations.

The Achilles' heel of a metamorphic virus, as described by Lakhotia, Kapoor, and Kumar in [36], is its need to analyse its own code. An antivirus scanner should be able to analyse a metamorphic virus by using whatever method the virus uses to analyse itself. Thus, a "reverse morpher" could be created which would apply all the transformation rules of the virus in reverse and reveal the real virus code. The problem with this is that virus researchers must first have a sample of the virus in order to extract its transformation rules, assumptions, and algorithms.

6.2 Geometric Detection

Geometric detection is based on modifications that a virus has made to the file structure. Peter Szor calls this method *shape heuristics* because is far from exact and prone to false positives [7].

Geometric detection can be used to detect Win95/Zmist. Because the data section of a file is increased by at least 32KB when it is infected by an encrypted version of the virus, the file might be reported as being infected if the virtual size of its data section is at least 32KB larger than its physical size. However, this method could introduce false positives, as the previous file structure alternation could also be an indicator of a runtime-compressed file [7].

This method could be used in combination with virus infection markers in order to decrease the risk of false positives. Sometimes viruses will make use of an infection marker in order to detect already infected files and avoid multiple infections. For example, the Win95/Bistro.B virus places a high-byte value 0x51 at the minor linker version field [40]. However, the risk of false positives is never eliminated.

6.3 Wildcard String and Half-Byte Scanning

Simple metamorphic viruses, such as viruses that use register swapping and instruction replacement, can be detected by wildcard and half-byte scanning (see section 3.1.2 on page 15). The following example code fragment belongs to Win95/Regswap virus and was published in [40]:

First generation:

```
BE04000000      mov   esi,000000004
8BDD            mov   ebx,ebp
B90C000000      mov   ecx,00000000C
81C088000000    add   eax,000000088
8B38            mov   edi,[eax]
89BC8B18110000  mov   [ebx][ecx]*4[00001118],edi
2BC6            sub   eax,esi
49              dec   ecx
```

Second generation:

```
BB04000000      mov   ebx,000000004
8BCD            mov   ecx,ebp
BF0C000000      mov   edi,00000000C
81C088000000    add   eax,000000088
8B30            mov   esi,[eax]
89B4B920110000  mov   [ecx][edi]*4[00001120],esi
2BC3            sub   eax,ebx
4F              dec   edi
```

It is obvious that there exist many common opcodes that are constant to all generations of the Regswap virus. This makes the extraction of usable search stings using wildcards possible. If the scanner supports it, half-byte detection would also be appropriate for this type of infection [7].

6.4 Code Disassembling

Disassembling the virus code means separating the stream into individual instructions. This technique is good for detecting viruses that insert garbage code between their code, because instructions can be too long and simple search strings cannot be used. Also, as explained in [7], there is a possibility that a string can appear inside an instruction instead of being the instruction itself. For example, if one wants to search for the instruction "CMP AX, 'ZM' ", which tests if a file is executable – something common among viruses – one would search for the string:

```
66 3D 4D 5A
```

The above string can be found in the following stream:

```
90 90 BF 66 3D 4D 5A
```

If the above stream is disassembled and displayed, what is found is not the instruction mentioned above, but the following:

```
nop
nop
mov     edi, 5A4D3D66
```

Such mistakes can be avoided with the use of a disassembler.

This technique becomes a powerful tool when combined with a state machine[1], which could record the order in which "interesting" instructions are found. It becomes even more powerful if it is combined with an emulator, and it becomes capable of detecting difficult viruses like Win95/Zmist or Win95/Puron – based on the Lexotan engine [7]. The following sample detection of Win95/Puron was published in [23]:

```
movzx   eax, ax
mov     ecx, dword ptr [edx+3C] ; interesting
xor     esi, esi
mov     esi, 12345678
cmp     word ptr [edx], 'ZM'    ; interesting
mov     ax, 2468
```

Its easy to detect Lexotan and Puron using only a disassembler and a state machine, because both viruses execute the same instructions in the same order, with only garbage and jump instructions inserted between the core instructions [23].

6.5 Using Emulators

As described in section 3.4 on page 20, code emulation implements a virtual machine to simulate the CPU and memory management system and executes malicious code inside the virtual machine. The malicious code cannot escape the virtual machine of the scanner [7]. Antivirus scanners can

[1]A state machine is a model of behaviour composed of a finite number of states, transitions between those states, and actions [46].

run code inside an emulator and examine it periodically or when interesting instructions are executed. For example, “INT 21h” is a very common instruction to search for in DOS viruses. The following sample detection of the ACG virus was published in [23]:

```
mov  ax,  65a1
xchg dx,  ax
mov  ax,  dx
mov  bp,  ax
add  ebp, 69bdaa5f
mov  bx,  bp
xchg bl,  dh
mov  bl,  byte ptr ds:[43a5]
xchg bl,  dh
cmp  byte ptr gs:[b975], dh
sub  dh,  byte ptr ds:[6003]
mov  ah,  dh
int  21
```

In one class of the ACG virus, when the “INT 21h” is reached, the registers contain ah=4a and bx=1000. The basis of ACG detection is to trap enough similar instructions.

Detection of Win95/Evol

Evol is a virus that deals with the problem of hiding constant data as variable code inside itself, from generation to generation. Code tracing can be very useful in detecting such changes. Evol builds the constant data on the stack from variable data and then passes them to the actual function or API that needs them. With appropriate break points, emulators can be very useful on dealing with such viruses [7].

The only thing needed is a p-code scanning language that can be used to write algorithmic detections. For viruses such as Evol, which often build constant data on the stack, the emulator can be instructed to run the emulation until a predefined limit of iterations and to check the content of the stack after the emulation, for constant data built by the virus. The content of the stack can be very helpful in dealing with complex metamorphic viruses that decrypt data on the stack [7].

6.5.1 Using Negative and Positive Features

This is a technique used to make the scanning process faster. Positive detection checks for a set of patterns that exist in a virus body, while negative detection checks for the opposite. Negative detection can be used to stop the detection process by identifying a set of instructions that do not appear in any of the instances of the actual metamorphic virus.

6.5.2 Using Emulator-Based Heuristics

Heuristic detection does not identify viruses specifically but extracts features of viruses and detects classes of computer viruses generically. Information in this section were taken from [7], unless otherwise stated.

The heuristics engine can track the interrupts or implement a deeper level of heuristics using a virtual machine that simulates the operating system. Such systems can even replicate the virus inside the virtual machine on a virtual file system. Some antivirus products implement such systems and find them to be very effective, providing less false positives. This technique requires emulation of file systems. For example, whenever a new file is opened by the emulated virus, a virtual file is given to it. Then the emulated virus might decide to infect the virtual file in its own virtual system.

To detect and prove the virus replication process, the heuristics engine can take the modified virtual file from one virtual machine and place it in another one. If the modified virtual file modifies other virtual files in the new virtual machine – similarly to previous experienced virus-like changes – then the virus replication in detected.

One big problem with this is that is very difficult to emulate multi-threaded systems – especially Windows on Windows build into a scanner. Because of Windows complexity, such systems cannot be perfect, not even when using a system such as VMWARE. Third-party DLLs are not part of the actual virtual machine and if a virus attempts to make use of such an API set, the emulation of the virus is likely to be broken.

Another big problem is performance. It does not matter how good a scanner is and if it can detect every single virus; if it is not fast enough it is bound to fail as a product. Customers believe that faster is better, so antivirus scanners would always have to compromise regarding speed, even if all the resources to develop a perfect VM to emulate Windows on Windows inside a scanner were available. However, as Peter Szor comments,

> "...extending the level of emulation of Windows inside the scanner system is a good idea and leads to better heuristics reliability. Certainly, the future of heuristics relies on this idea."

Unfortunately, virus writers are aware of emulation-based antivirus techniques. They incorporate several anti-emulation techniques to challenge antivirus systems. EPO viruses, such as Win95/Zmist, can trick emulators. Anti-emulation tricks were known even in DOS days by viruses such as ACG, which replicates only on certain days or under similar conditions. If the scanner use pure heuristics and pay no attention to virus-specific details, detection would never be perfect. For example, if one scans on Tuesday for a virus that only replicates on Mondays, the virus could be easily missed. Another example is the Win32/Magistr virus which never infects without an active internet connection. If the virus looks for a specific web site and the emulator is not able to provide a proper real-world answer, the virus would not replicate and the scanner would not detect it.

Moreover, there will be viruses that cannot be detected not even in a perfect emulated environment. Only virus-specific detection can work for such viruses. No doubt that some of them will be metamorphic too.

6.5.3 Dummy Loops Detection

Another anti-emulation technique was introduced by an improved version of the Bistro virus, which was released some time after the original. This technique, which is called *random code insertion* (macho engine), inserts garbage instructions and dummy loops randomly before the decryptor code. This results to some emulators to fail to rebuild the real virus and emulate millions of garbage instructions.

Emulators must have a means of identifying garbage and do-nothing instructions and dummy loops and must be able to skip them. For example, the macho engine of the Bistro virus can be detected by monitoring the movement of IP and checking the “WRITE” operations. This method can be used for generic detection of all Win32 viruses that use macho engines; however, this method can produce false positives. Information about this technique comes from [40].

6.5.4 Stack Decryption Detection

Variants of the Zmorph virus, a metamorphic virus created by the virus writer Z0mbie, place a piece of polymorphic code at the entry point of an infected file. Then, they decrypt the virus instruction-by-instruction and rebuilt it by pushing the result into the stack memory. After decryption of the last instruction, control is transferred to the start of the virus body in the stack.

If the emulator is not capable of detecting stack decryption, such viruses would be missed. The memory accessed by the virus must be monitored by the emulator and when control is transferred to the stack memory, the emulator should detect it and dump the whole decrypted virus code for identification. The drawback of this technique is that is has a significant impact on the performance of the scanner, thus filter checking by geometric techniques (see section 6.2 on page 55) should be applied first. Information about this technique comes from [40].

6.6 Code Transformation Detection

Section 5.3.5 on page 42 discusses about instruction replacement techniques, were a metamorphic virus replaces instructions with other equivalent instructions. Code transformation is a method for undoing the previous transformations done by the virus. Code transformation is used to convert mutated instructions into their simplest form, where the combinations of instructions are transformed to an equivalent but simple form. After the transformation common code exhibited by the virus can be identified. All information and sample code about this technique were taken from [40].

The first metamorphic virus that this technique was applicable to, was Win32/Simile. This section provides a brief discussion on how Simile implements code transformation. For more details on Simile, see 5.4.2 on page 47. Some of the metamorphic techniques that Simile implements are: entry-point obfuscation (EPO), permutation, and heavy code mutation by shrinking and expanding techniques. The steps that the virus goes through to achieve code mutation are the following:

1. Disassembler
2. Shrinker
3. Permutator
4. Expander
5. Assembler

It uses a pseudo-assembler technique to decode instructions to a form it can manipulate. Then, it extracts the instructions, instruction lengths, registers, and other relevant information. Then, the shrinker compresses the disassembled code produced from previous generations, and removes garbage instructions. The following code is sample instructions that Simile has transformed:

```
xor     reg,-1          -> not reg
sub     mem,imm         -> add mem,-imm
xor     reg,0           -> mov reg,0
add     reg,0           -> nop
and     mem,0           -> mov mem,0
xor     reg,reg         -> mov reg,0
sub     reg,reg         -> mov reg,0
and     reg,reg         -> cmp reg,0
test    reg,reg         -> cmp reg,0
lea     reg,[imm]       -> mov reg,imm
mov     mem,mem         -> nop
```

The virus code is first processed by the permutator, in order to increase the level of metamorphism. The expander undoes what the shrinker did. It is also responsible for register translation and variable re-selection. The expander randomly selects instructions and in the final step, the assembler converts the pseudo-assembly code into real Intel IA-32 assembly instructions.

The following two pieces of code are two different generations of Simile. They look completely different but more detailed analysis will show that both assemble the string “kernel32.dll” in the stack and then call the GetModuleHandle API.

First generation of Simile:

```
mov     eax,06E72656B ;nrek
mov     [edx],eax
mov     eax,032336C65 ;23le
mov     [edx][04],eax
mov     eax,06C6C642E ;lld.
mov     [edx][08],eax
xor     eax,eax
mov     edx][0C],eax
call    .00040299D
```

Second generation of Simile:

```
push 6c6c442e                     ; mov ebp, lld.
pop ebp
mov edx,73b36c67                  ; mov edx, 23le > encrypted
and edx,3e7fdedd
push 4e72454b                     ; mov esi, nrek
pop esi
push ebp                          ; mov ecx, ebp
pop ecx
mov dword ptr ds:[42268c],ecx     ; mov mem+8, ecx
lea ebx,dword ptr ds:[esi]        ; mov ebx, esi
mov dword ptr ds:[422684],ebx     ; mov mem, ebx
mov dword ptr ds:[422688],edx     ; mov mem+4, edx
push 0                            ; xor reg2, reg2 or mov reg2, 0
pop edx
mov dword ptr ds:[422690],edx     ; mov [mem+c], edx
mov ecx,infect1.00422684          ; mov ecx, mem
push ecx                          ; push ecx
push <&kernel32.getmodulehandlea> ; mov edi, offset getmodulehandle
pop edi
call dword ptr ds:[edi]           ; call getmodulehandle via edi
```

The perfect solution for this virus is code detection but is very difficult to implement. It involves transforming the virus code back to its initial form, before the expansion stage – similar to the first generation. The virus must be transformed back to its simplest form, where common instructions for virus detection can be extracted. Three instructions are transformed to two or one instructions and two instructions are transformed into one.

However, to be able to guarantee perfect detection without compromising scanning speed, the code transformation module must be highly optimised and flexible. The virus location can be transformed from were the scan pattern is taken – this will reduce the impact on the performance of the scanner. If possible, checking filters by geometric techniques (see section 6.2 on page 55) will also improve the performance of the scanner.

6.7 Subroutine Depermutation

As the name suggests, subroutine depermutation technique is used for detection of viruses that use permutation of their code to form new generations. As described in section 5.3.3 on page 40, metamorphosis is achieved by dividing the code into frames, and then position the frames randomly and connect them by branch instructions to maintain the process flow.

The Zperm virus, a virus created by Z0mbie, uses the sophisticated Real Permutation Engine (RPME) in order to mutate its code. To detect such a virus, the scanner must perform partial emulation to reconstruct the virus code into its initial form before the permutation. Partial emulation means emulating branch instruction, such as jump instructions. Figure 6.1, which was published in [40], shows the process of rebuilding the permutated virus body.

Deciding when to stop decoding is the problem of this technique. Also, ensuring that the virus code is finished is another challenge. These problems can be solved with the help of the decode table and IP address table. In addition to rebuilding the virus code, this technique can be effective for removing garbage instructions too. Information about this technique comes from [40].

6.8 Using Regular Expressions and DFA

The use of disassembly code to match patterns (regular expressions) using Deterministic Finite Automata (DFA), is a very efficient way to deal with code obfuscation. All information about this technique comes from [40]. Before going any further, the following terminologies are appropriate:

Regular Expression: a formula for matching strings that follow a pattern. It provides a mechanism for selecting specific strings from a set of character strings.

```
Permutated code   Decoding procedure

    aaa1          1.   decode aaa1
    aaa2          2.   decode aaa2
    aaa3          3.   decode aaa3
    jmp @A        4.   change IP to @A
    bbb1          5.   decode aaa7
    bbb2          6.   decode aaa8

@B: aaa4          7.   decode aaa9
    aaa5          8.   change IP to @B
    aaa6          9.   decode aaa4
    jmp @C        10.  decode aaa5
    bbb3          11.  decode aaa6
    bbb4          12.  change IP to @C

@A: aaa7          13.  decode aaa10
    aaa8          14.  decode aaa11
    aaa9          15.  decode aaa12
    jmp @B        16.  decode ret

@D: aaa13
     aaa14
     ret
     bbb5

@C:  aaa10
     aaa11
     aaa12
     ret
```

Figure 6.1: Process of rebuilding a permutated virus body (from [40])

DFA: a transition table containing states and their corresponding next states.

Automaton: a predetermined sequence of operations. In this context, it corresponds to the sequence of disassembly codes.

Grammar: the rules of a language. In this context, the grammar pattern pertains to the collection or set of disassembly codes that the virus uses and provides the rule or the positive filter for detection.

Simply speaking, this method treats the virus as a series of disassembly codes that can be matched against a database of existing virus disassembly codes. This method terminates the scanning of a file automatically when the current disassembly code does not match any of the disassembly codes in the database, or when the disassembly code does not belong to the acceptable list of instructions for a certain virus. This makes this technique relatively fast.

The two main components involved in this method is the *builder* and the *simulator*. The builder creates the automaton of the virus using the grammar pattern. The simulator performs the automaton matching and conditional test, using regular expression operators during file scanning. The grammar pattern contains information on normalisation (a set of garbage or negative filters) and information on how to detect the malicious file. It uses regular expression where each item represents an assembly instruction.

An opcode can be any Intel IA32 assembly instruction and an operand can be any of the following:

- Exact: specifies the exact operand to match. For example:

```
push    eax
```

- Wildcard: specifies the general type of the operand. For example:

```
push    reg32
mov     reg,imm
```

- Variable: information on an operand may be stored and retrieved later for matching. For example:

```
dec     reg32_varset1
push    reg_var1
```

In wildcard instruction the opcode and the operand vary. Possible values for the register operand are the following “reg”, “reg8,”, “reg16”, “reg132”. Possible values for the immediate operand are the following "imm", "imm16", "imm32". Possible values for the memory operands are the following "mem", "mem16", "mem32". Assembly instructions are associated through operators such as start (*), plus (+), questionmark (?), and explicit dot (.).

Figure 6.2, shows the DFA building process. The pattern source format is processed by the DFA to produce automatons. Each assembly instruction is assigned a unique ID for easy matching and added to the corresponding garbage, accept, and grammar lists.

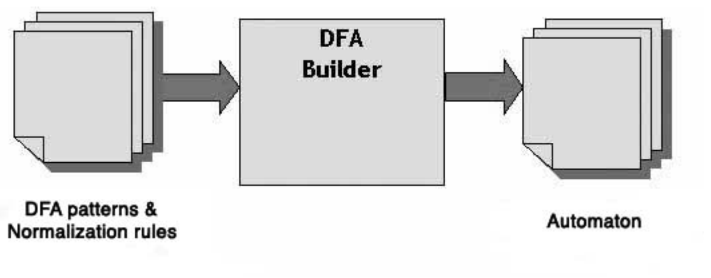


Figure 6.2: DFA building process (from [40])

The simulator, which scans files for viral code, has four sub-components:

- a disassembler,
- a depermutator,
- a normalizer, and
- a DFA simulator.

Before any input data is passed to the DFA simulator, it is pre-processed by the the first three components. Figure 6.3 on the following page, shows the components of the simulator.

The disassembler performs the conversion from binary code to assembly code. The depermutator component connects the subroutines of the permutated virus. The normalizer explicitly drops garbage instructions. The final step of the process is DFA simulation. Using the input symbol derived from the file being scanned and the automaton created in the building process, the DFA simulator scans the file for malicious code.

For every input symbol the simulator checks for the matching states and updates them accordingly. Wildcards and conflicts in pattern are resolved by having a set of transition states. When an input symbol is rejected, the DFA simulator checks the entries in the Accept section. If there is a match, the state is toggled back as if it was not rejected. It then reads the next input and continues the simulation. Once the final and accepting state is reached, the file is tagged as a virus.

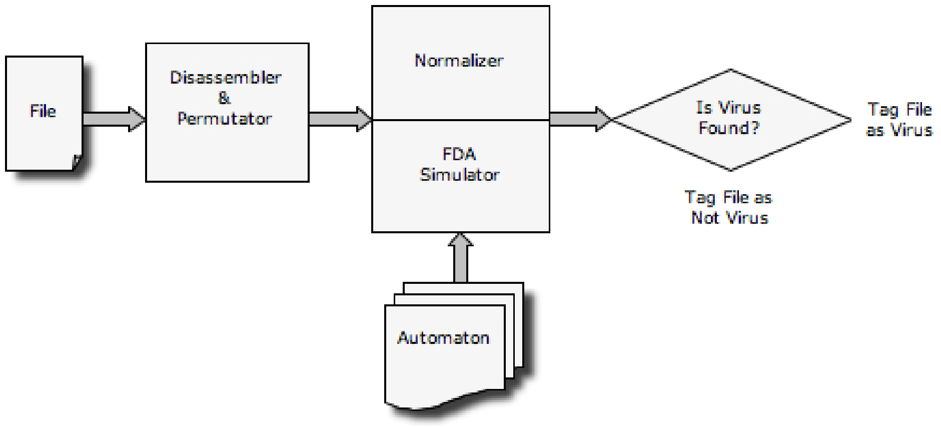


Figure 6.3: DFA simulation process (from [40])

This detection technique covers almost all of the code evolution techniques discussed in the previous chapters. Encrypted viruses can be detected by creating a virus signature based on the decryptors' disassembly code. Oligomorphic and polymorphic viruses can be addressed by creating an automaton based on the virus alphabets, or the possible set of instructions that it can produce during infection. This method also covers the detection of permutating viruses through the depermutator component, which connects the subroutines of the permutated virus.

Emulators are known to be slow and cannot handle viruses that generate do-nothing loops, unless they are able to detect dummy loops (see section 6.5.3 on page 60). This technique simply treats the virus as a series of disassembly codes that can be matched with a database of existing virus disassembly codes. For more sophisticated viruses, such as Zmist and Simile, this detection method works best if combined with an emulator.

6.9 Experimental Detection Techniques

This section describes some experimental techniques for detecting metamorphic viruses. These techniques were published in articles as proposals or experiments and are not implemented in commercial antivirus products yet.

6.9.1 Detection Using Engine Signature

This technique was published by Mohamed Chouchane and Arun Lakhotia in [35].

Introduction

This technique makes use of an engine-specific scoring procedure that scans a piece of code to determine the likelihood of that being part of a program that has been generated by a known metamorphic engine that implements instruction replacement techniques (see 5.3.5 on page 42). For this method, all that needs to be stored for detection purposes is information about the engine, rather than information about each possible virus variant it can generate.

The scoring technique is designed for metamorphic engines that transform their input variant of a virus using a finite set of transformation rules – mapping instructions to code segments that implement their operational semantics. When such an engine is given as input a code segment to be transformed, it scans the segment for instructions that can be transformed. The engine sometimes needs to perform simple context analyses, or make assumptions about the code, to decide whether or not the instruction can be transformed.

For example, the third rule in the instruction replacement system of Figure 6.4, is semantics preserving only if register "eax" is dead at that point. When the engine determines that it is safe to transform the instruction, it probabilistically decides whether or not to replace it with an equivalent code segment. The engine of Win32/Evol is an example of such an engine where each rule has its own application probability.

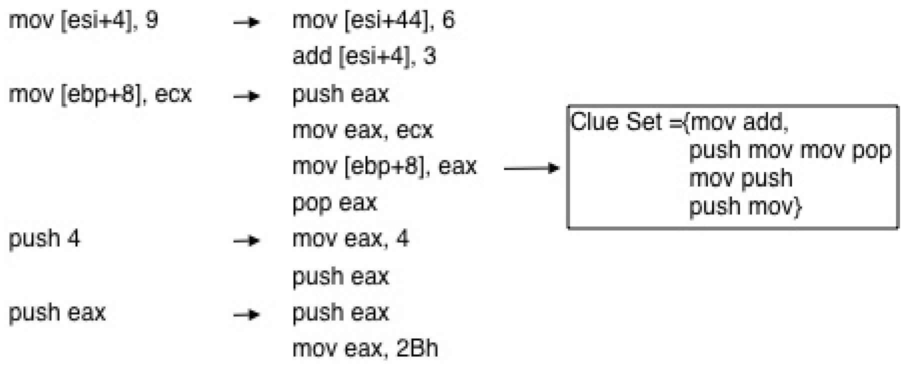


Figure 6.4: Instruction replacement system of Win32/Evol (from [35])

The scoring technique was inspired by Chouchane and Lakhotias' observation that over 50% of the original variant of instruction replacing metamorphic malware is transformable. They use the phrase *engine friendliness* to refer to this level of transformability of the variant. Such engines also tend to be written such that their transformation rules preserve this level of transformability across generations; that is, high engine friendliness usually applies to all variants of the malware. This is typically done – in the case of instruction-replacing engines – by ensuring that even the code segments that are used to replace other code segments contain at least one transformable region. This technique would be useful in detecting new viruses that make use of known metamorphic engines created by more advanced virus writers.

Evaluation

The scoring function used in [35] is given by the expression:

$$S_E(V) = \frac{\Sigma_c \Sigma_S w_c e_{cs}}{|V|}$$

The explanation of the above formula is out of the scope of this project. To evaluate the scoring function, Chouchane and Lakhotia implemented a prototype simulator of an instruction replacing metamorphic engine and used it to run two experiments. The rule set they chose to use was the one used by the metamorphic engine of the Win32/Evol virus.

The goal of *Evaluation 1: Tracking typical engine output to the engine*, was to determine how well, and for what parameter choices the scoring function can assist in discriminating the engines output from arbitrary code segments.

The goal of *Evaluation 2: Tracking variants of fixed metamorphic malware to the engine*, was to determine how well, and for what parameter choices the scoring function can assist in discriminating variants of known malware from arbitrary code segments.

Evaluation 1 showed that the scoring method successfully managed, in several cases, to score the engines' output considerably higher than engine-independent segments. It can be seen from the simulation results that the function performed particularly well on all variants, when the original variant engine-friendliness was over 50%. For lower engine-friendliness values, say 5%, the function was only successful in telling later generation variants from engine-independent segments.

Evaluation 2 showed that segments from Win32/Evol could be tracked to Evols' engine by measuring their closeness to the means of the distributions corresponding to each generation. An inspection of the simulation results reveals that segment scores seem to somehow converge towards some small range of values, as the segment mutates.

Conclusion

Chouchane and Lakhotia introduced a novel approach for dealing with metamorphic malware; a method that takes advantage of the fact that metamorphic engines – in order to generate an output program that is as different as possible from the input – typically require their input to be highly transformable. They used a scoring function to illustrate this approach on a metamorphic engine capable of performing instruction replacement, that of Win32/Evol. They concluded that more analysis would perhaps be needed should the scanner wish to gather more evidence that the code being scanned is in fact from a variant of Win32/Evol.

6.9.2 Detection Using Redundancy Control Strategy

This method was proposed by Ando, Quynh, and Takefuji in [34].

Introduction

Ando, Quynh, and Takefuji proposed a resolution-based detection method for detecting metamorphic viruses. As they explain, their method is the application of formal verification using theorem proving, which extracts parts of virus code from a large number of obfuscated operations and re-assembles them in order to reveal the signature of the virus. In their paper, they have shown that the complexity of metamorphic viruses can be solved if the obfuscated virus code is canonicalized and simplified using resolution-based state pruning and generation.

They tried to make their detection method more feasible and effective by applying redundancy-control strategies for the resolution process. In their paper they applied demodulation and subsumption in order to eliminate the redundant path for resolution. Their experiment showed that without the strategies mentioned above, resolving metamorphic code into several simplified operations in reasonable computing time, is not feasible.

They also presented statistics of reasoning process in detecting obfuscated API calls. They divided obfuscated API calls into four modules, according to the types of metamorphic techniques, and compared the conventional resolution with their method, applying redundancy-control strategy.

State resolution and demodulation

As Ando, Quynh, and Takefuji explain in their paper, if a program is infected, there has been a state transition to achieve some malicious operations, such as making API calls. To resolve these instructions from obfuscated code, the resolution system needs to store all states generated by the execution of every instruction. Their proposed method for the resolution-based detection of obfuscated metamorphic code is illustrated in Figure 6.5 on the following page.

Their method consists of two types of reasoning; resolution and demodulation. First, they apply hyper resolution to several instructions to deduce them to one simplified instruction. Second, garbage code such as "nop", is eliminated by a technique called demodulation, which is discussed later. As they explain, in the process of detection, they add the formulation of disassembly code of viral code besides the code under inspection and a reasoning

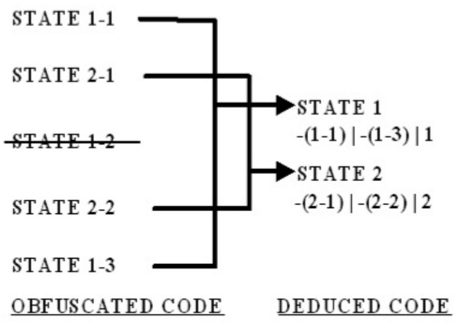


Figure 6.5: State pruning and resolution (from [34].)

program attempts to deduce the same as *signature clauses* from obfuscated metamorphic code. The reasoning program terminates (detection is succeeded) when the equivalence is found between two clauses (codes).

Redundancy-control strategy

As described in [34], redundancy control strategy is designed to reduce the obstacle for the reasoning program within the retained information. Ando, Quynh, and Takefuji applied two strategies for this; subsumption and demodulation.

Subsumption is the process of discarding a specific statement. The clause that is duplicated or is less general, is discarded in the already-existing information. As a result, subsumption prevents a reasoning program from keeping clauses that are obviously redundant [34].

Demodulation is the automated reasoning; one of the procedures of simplifying or canonicalizing information. Demodulation in an effective way to eliminate garbage instructions [34].

Conclusion

Ando, Quynh, and Takefuji introduced a resolution-based technique for detecting metamorphic viruses. In their method, scattered and obfuscated code is resolved and simplified to several parts of malicious code. Their experiment showed that compared with emulation, this technique is effective for metamorphic viruses which apply anti-heuristic techniques, such as register substitution or permutation methods.

6.9.3 Detection Using Control-Flow Graph Matching

This technique for the detection of metamorphic malicious code was proposed by Bruschi, Martignoni, and Monga in [47].

Introduction

As its authors explain, this method is used to detect metamorphic malicious code inside a program P based on the comparison of the control flow graphs of P against the set of control flow graphs of known viruses. More precisely, given an executable program P, Bruschi, Martignoni, and Monga disassemble it to obtain a program P'. On P' they perform a set of normalisation operations in order to reduce the effects of mutation techniques, and to uncover the flow connections between the benign and the malicious code. What is left is a new version of P, namely P_N.

Then, they take P_N and build its corresponding labelled *inter-procedural control flow graph* CFG_{P_N}. Then, they compare CFG_{P_N} against the control flow graph of a normalised malware CFG_M in order to verify whether CFG_{P_N} contains a subgraph which is isomorphic to CFG_M, thus reducing the problem of detecting a malware inside an executable. Using this strategy they were able to defeat most of the mutation techniques.

Detecting malicious code

The problem that exists when detecting metamorphic malicious code inside a file, is that the metamorphic malicious code scatters itself and becomes part of the host. The scanner must deal with both mutation and scattering. To deal with mutation, Bruschi, Martignoni, and Monga, try to normalise different instances of the same malicious code into a canonical and minimal form. To do this, they devised a detection process which is composed by two components, which are explained in the following paragraphs, as described by their authors: the *code normalizer* and the *code comparator*.

Code Normalizer — is the component responsible for normalising a program, which means transform it into a canonical form – simpler in terms of structure or syntax – while preserving the original semantics. Normalisation can be seen as a code optimisation method because it is responsible for removing all garbage instructions introduced during the mutation process. To do this, it performs the following steps:

- Decoding
- Control-flow and data-flow analysis
- Code transformation

Code Comparator — is responsible of taking a program P and a malicious code M, and find out whether program P is hosting the code M. Bruschi, Martignoni, and Monga decided to represent the malicious code and the host program by their *inter-procedural control flow graphs.* A control flow graph (CFG) is an abstract representation of a procedure; each node in the graph represents a basic block, and jump targets start a block and end a block. Directed edges are used to represent jumps in the control flow. An inter-procedural CFG links together the CFGs of every function of a program [47].

Conclusion

Bruschi, Martignoni, and Monga proposed a detection method for metamorphic and polymorphic malware using control-flow graph matching. Mutations are eliminated through code normalisation and the problem of detecting viral code inside an executable is reduced to a simpler problem. I quote their last words as a conclusion:

> "We believe that experimental results are encouraging and we are working on refining our prototype in order (i) to validate our approach in more real scenarios as we are aware that our current results are not complete and (ii) to be able to cope with malicious code that adopts countermeasures to prevent static analysis."

6.9.4 Detection Using Algebraic Specification

This experimental technique was proposed by Webster and Malcolm and was published in the "Journal in Computer Virology." More detailed information on this method can be found in [48].

Introduction

Webster and Malcolm introduced a metamorphic virus detection technique using an algebraic specification of the IA-32 assembly language. In order to identify instruction sequences with similar behaviour, they use OBJ[2] to formally specify the syntax and semantics of a subset of the IA-32 assembly language instruction set.

As Webster and Malcolm explain, the OBJ specification is useful for providing the equivalence or semi-equivalence of IA-32 instruction sequences by applying reductions – sequences of equational rewrites in OBJ. They continue by explaining that the OBJ specification, when combined with the OBJ term rewriting engine, can be used as an interpreter for programs

[2]An algebraic specification formalism and theorem prover based on order-sorted equational logic [48]

in IA-32 assembly language, and this can be used for dynamic analysis of computer viruses. In their experiment, they apply their proof methods to fragments of the metamorphic viruses Win95/Bistro and Win95/Zmorph.

Proving equivalence of viral code

Webster and Malcolm use the term *allomorphs* to describe any two generations of the same metamorphic virus that differ syntactically. They showed that, using the formalisation specification, it is possible to prove the equivalence or semi-equivalence of various allomorphs of metamorphic viruses using reductions in OBJ, by using the OBJ specification as an interpreter. Description of the formal specification of IA-32 or the OBJ specification can be found in [48] and is out of the scope of this project.

Application to antivirus scanning

If one specifies a programming language, such as IA-32, using a formal notation and theorem prover, such as OBJ, one obtains an interpreter and a program analysis tool for that language [49]. The behavior of a suspected piece of code, could be checked by interpreting it using the OBJ specification of IA-32. Webster and Malcolm used this method with two variants of the Win95/Zmorph.A virus and showed their equivalency with respect to the stack. This means that the state of the stack was affected in the same way by both generations of the virus.

They propose that the IA-32 specification in OBJ could be applied as a means of code emulation-based dynamic analysis. They also propose that checking whether a suspect code fragment behave equivalently or semi-equivalently to a signature of a metamorphic computer virus, could be an application to aid signature scanning.

Conclusion

Webster and Malcolm introduced a new method for the detection of metamorphic malware, by formally specifying the semantics of a IA-32 assembly language using OBJ. They proved that these techniques are readily available to real metamorphic viruses, by proving equivalence and semi-equivalence for allomorphs of Bistro and Zmorph viruses.

6.9.5 Hidden Markov Models

A method for detecting metamorphic viruses using Hidden Markov Models was implemented by Wong and Stamp and published in [50]. All information is this section and more details can be found in their original paper cited above.

Introduction

Hidden Markov Models (HMMs) is a useful technique for statistical pattern analysis. They are used in many applications including speech recognition and biological sequence analysis.

An HMM is a state machine where the transitions between states have fixed probabilities. Each state in an HMM is associated with a probability distribution for a set of observation symbols. An HMM can be trained to represent a set of data, where the data is in the form of observation sequence. The states in the trained HMM represent the features of the input data. The transition and observation probabilities represent the statistical properties of these features. Given any observation sequence, we can score it using the trained HMM – the higher the score the more similar the sequence is to the training data.

HMMs and Metamorphism

HMMs can be used to detect families of viruses, and since metamorphic viruses form families of viruses, HMMs can be used to detect them. The problem is that metamorphic viruses change their form from generation to generation. Despite the previous fact, some similarities always exist between them. If these similarities can be found, then HMMs can be used to detect these viruses. A trained HMM should be able to assign a higher probability to viruses that belong to the same family as viruses in the training set.

The experiment

Wong and Stamp trained their Markov models using the assembly code sequences of metamorphic virus files. They first disassembled the executable files and extracted sequences of opcodes from each. Then, they concatenated the opcode sequences to create one long observation sequence and used it for training the models. When trained with multiple sequences, the resulting HMM represents the average behavior of all sequences in the form of statistical profile. This way, Wong and Stamp were able to represent an entire virus family with a single HMM.

After training a model, they used the resulting HMM to compute the log likelihood for each virus variant in the test set and also for each program in the comparison set. Their test set consisted of viruses in the same family as those used for training, and the comparison set included normal programs and viruses in other families. Comparing the scores of the files in the test set with that of the files in the comparison set, they expected to see a clear separation between the two sets. Their data set consisted of 200 viruses generated by the Next Generation Virus Creation Kit (NGVCK).

The results showed that it is possible to set a threshold whereby the family viruses are always distinguished from the normal files. There were some false positives, but these were entirely due to non-family viruses.

Conclusion

Wong and Stamp experimented with Hidden Markov models to try to detect metamorphic malware. They were able to distinguish NGVCK viruses from normal programs, despite the fact that NGVCK viruses show a high degree of metamorphism. The fact that metamorphic malware have assembly code structure that is very different from normal programs makes them detectable by methods like HMMs. Wong and Stamp concluded that in order to avoid detection, metamorphic viruses also need a degree of similarity with normal programs and this is something very challenging for the virus writer.

6.9.6 Zeroing Transformation

As their authors, Lakhotia and Mohammed describe, this is a method to impose an order on the statements and components of expressions of a program. All information in this method comes from [51].

Introduction

This method, named *zeroing transformation*, is used to reduce the number of possible variants of a program created by reordering statements, reshaping expressions, and renaming variables. Lakhotia and Mohammed used this method with a collection of C programs, and were able to reduce the space of program variants due to statement reordering from 10^{183} to 10^{20}. If used for metamorphic viruses, this is a reduction in the number of signatures needed. This means that antivirus technologies may be improved by extracting signatures from the *zero* form of a virus, and not from its first generation [51].

This method is another step towards transforming a program – ultimately a metamorphic malware – to a canonical form, such that different variants of the program have the same form. In their paper, Lakhotia and Mohammed present a set of heuristics to impose order on the statements of C-like programs. They expect that by imposing such an order, antivirus technologies can undo the effect of statement/instruction reordering transformations performed by metamorphic malware. As they explain, while the most challenging viruses are binary, it is expected that for any significant analysis a virus will have to be first decompiled to a higher language, thus making this method applicable for binary viruses.

For their experiment they used GrammaTechs' CodeSurfer[3] to imple-

[3]A code browser that understands pointers, indirect function calls, and whole-program effects. CodeSurfer is a program-understanding tool that makes manual review of code easier and faster.

ment the technique in a prototype tool named $C\oplus$.

Zeroing Transformation

Lakhotia and Mohammed call their method the *zeroing transformation*, for its attempt to eliminate the effect of statement reordering, variable renaming, and expression reshaping. The zero form of a program is the result of applying the zeroing transformation on that program. The following are the steps of the zeroing transformation, as presented in [51]:

1. Create a Program Tree[4] (PT) representation of the program.
2. Partition the PT nodes into re-orderable sets[5], each set containing statements that may be mutually reordered without affecting the semantics of the program.
3. Partition each re-orderable set into a sequence of isomorphic sets, where every statement in an isomorphic set has the same string representation. The representation does not depend on names of the variables in the program, order of variables in commutative operators, and order of the statements in the program.
4. Assign a number to each statement. The numbering is done using a depth-first traversal of the PT. Statements in a re-orderable set are visited based on the order in the sequence of isomorphic set. Statements in an isomorphic set are visited in random order.
5. Create a new program by ordering the statements as per the numbers assigned in the last step. In each expression, replace each variable name by a new variable name created using the number of the statement where it is first defined.

Further explanation of the previous steps can be found in [51] and its out of the scope of this project.

The experiment

The authors of [51] developed a tool that implements their proposed method for imposing order on the statements of C programs. Their tool used the Program Dependence Graph (PDG) – generated by CodeSurfer$^{\text{TM}}$– to gather the control and data dependencies needed to identify re-orderable statements. They used the tool to analyse a set of real world C programs to study how well their proposed method imposed order on the programs'

[4]A Program Tree (PT) is a hierarchical ordering of statements in a program. [51]

[5]A set of nodes in PT is re-orderable if its nodes can be mutually reordered (swapped) without changing the semantics of the program represented represented by a PT [51]

statements. The programs used are the following: Bison, Cook, SNNS, Fractal, and Computer Vision.

Figure 6.6 shows the reorderable percentages of the test programs. On an average, 55% of the statements of the original program are reorderable. After processing only 6% of the statements remained reorderable – could not be ordered using this method. The most important filter is the SR1 filter, which significantly reduces the number of possible permutations for each program.

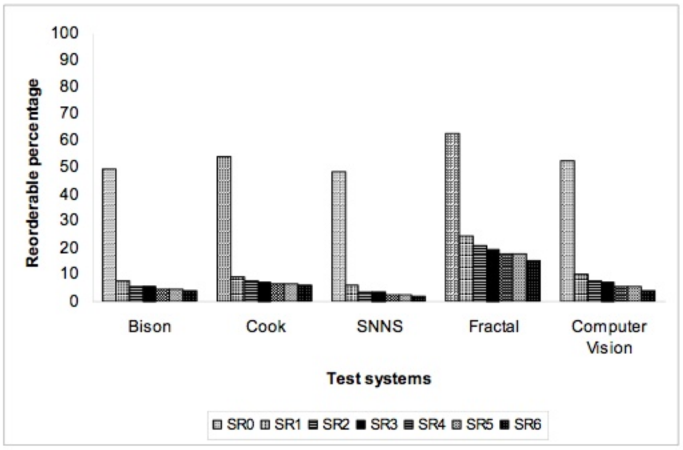


Figure 6.6: Reorderable percentages for test systems (from [51])

Conclusion

Lakhotia and Mohammed developed transformations to undo the effect of the following transformations performed by metamorphic viruses: statement reordering, expression reshaping, and variable renaming transformation. They called the resulting form a zero form. Their experiment showed that zeroing transformations results in a significant decrease in the number of possible variants of a – ultimately malicious – program.

Chapter 7

Conclusions and the Future

7.1 Current Trends in Malware

In the last few years, there has been a significant change in the field of information security, more specifically in the field that deals with malicious code. A few years ago, malware were designed to cause major indiscriminate damage. For example, in 2004 the Blaster worm was responsible for more that half of the worst incidents in large businesses [1]. The "successful" virus writer was the one whose malware caused the most infections and gathered the most media attention.

This is not the case any more. In the last two years, no major damage was done by a single malware, but by a high number of different malware and variants. A 2006 research reports that 60% of respondents could not recall which malware caused their worst incident [1]. The motivation of virus writers has changed, and as a result the nature of malware has changed too. Profit is what makes virus writers happy nowadays. Also, the law started to catch up with cyber-criminals and made them realise that facing prison just for fun is not a very smart thing to do. As a result, malware became more insidious and target specific user groups. From "hobbyists," virus writers became professional criminals motivated by profit. Lee Phisher, a McAfee Security Strategists points out:

> "We have entered a new phase of malicious activity. Cyber-crime is now driven by those out to make money, which has led to growing involvement by organised criminals. They will capitalise on every opportunity to exploit new technologies and the general lack of awareness surrounding securityand their methods are becoming more subtle and sophisticated every day."

Cyber criminals, among other techniques, make use of malicious code to infect computers and use them for a variety of reasons, including theft of private and confidential information. A recent report from Symantec showed that 66% of the volume of the top 50 malicious code reported to Symantec between July and December, 2006, targeted confidential information. The cyber-criminal can use the stolen confidential information for any of the following reasons, as described in [52]:

- Extortion
- Reputational Damage
- Fraud
- Money Laundering

The spyware menace

Spyware is a fast growing thread. As described in 2.4 on page 10, spyware are malicious software installed on a computer, monitor user activity, and report back to a third party. They are often downloaded and installed when a user visits a malicious website or a legitimate website which has been injected with malicious code. Although this problem is fairly new, there exist many COTS[1] products, solutions to the problem. However, a quarter of UK businesses are not protected against spyware and as a result, roughly one in seven of the worst malicious software incidents involve spyware [1].

Attack of the zombies

One increasingly popular activity among cyber-criminals is the use of malware to create *zombie computers.* A zombie computer or *bot*, is a computer on which a worm or virus has installed programs that run automatically and allow cyber-criminals access and control [52]. Cyber-criminals use these zombie computers to search for other vulnerable computers, install their programs or store data. A zombie network or bot network, is a set of infected machines often compromised weeks or months earlier by attackers using worms or viruses to plant backdoor components. These zombie networks can contain from a few hundreds to several thousands of compromises machines and can be used to launch simultaneous Denial-of-Service (DoS) attacks, send spam, host illegal websites or illegal material, and other criminal activities. Between July 1 and December 31, 2006, Symantec observed an average of 63,912 active bot-infected computers per day. This is an 11% increase from the previous period when Symantec observed an average of 57,717 active bots per day. Furthermore, Symantec observed 6,049,594 distinct bot-infected computers during the current reporting period [2]. Owners of zombie networks can use them for their own activities or rent them to other criminals, such as spammers.

[1]Commercial Of-The-Shelf is a term for software or hardware that are ready-made and available for sale, lease, or license to the general public.

Infection Strategies

One of the greatest changes of the last few years, is the way computers get infected. A couple of years ago, malware spread through the use of infected email attachments sent directly to users. According to Sophos Labs, in 2005, 1 in 44 emails was infected with some sort of malware. In late 2006 and early 2007 only 1 in 337 and 1 in 322 respectively was infected.

Today, malware are placed on a website and are downloaded and installed automatically when users browse to the compromised website. Users are lured to the compromised websites by spam email invitations. According to the latest Sophos Threat Report [53], the web-based thread that counts for nearly half the worlds' infected webpages is what Sophos Labs calls *Mal/Iframe*. It works by injecting malicious code into vulnerable webpages. The Sophos research shows that 80% of all web-based malware are being hosted on legitimate but compromised websites. Many threads are designed to attack web files, such as HTML, ASP, JS, and VBS, and an infection on a web server can infect up to thousands of web files, which may part of hundreds of different websites. This factor alone makes this attack very significant and one that demands immediate care.

Fortunately, anti-virus vendors are catching up with this thread and working on techniques to defend against. Peter Szor, the Security Architect of Symantec Security Response, informed me that on Norton AntiVirus 2008 they will include a new feature to protect against web- based malware [54].

7.2 Trends in Metamorphic Malware

The whole area of malware is changing and the metamorphic virus is not an exception. The latest worth mentioning metamorphic virus that was analysed by experts was Simile in 2002. Peter Szor could not even recall the names of any metamorphic viruses he examined after 2002 [54]. The fact that there were no significant metamorphic virus in the wild for the last five years might be because it is so difficult to write, that virus writers turn to other methods.

Metamorphism is moving to other malware, such as Trojans and spyware, which are typically distributed from infected or malicious websites. Some of these malware are changed each time a user visits the infected website upon clicking on a link of a phishing e-mail. This strategy can be described as metamorphism, since the code of the malware itself is changed, often without any encryption [54].

Attackers are more interested in money, so instead of spending much time and effort in writing a metamorphic malware, they simply use packers[2]

[2]Malware packers build an outer shell for malicious files in order to make them less recognisable by antivirus or antispyware programs

and wrappers[3] on the top of Trojans. This is the main reason why there are so many different Trojans in the wild [54]. From the volume of the top 50 malicious code reported to Symantec between July 1 and December 31, 2006, 45% were Trojans. Trojans accounted for 60% when measured by potential infections [2].

Return to Polymorphism

In the first half of 2006, Symantec Security Response noticed a renewed interest in self-mutating and polymorphic viruses. Over the past several months, malicious code authors have been developing increasingly sophisticated malicious code that employs these techniques. During March and April of 2006, a worldwide outbreak of two polymorphic viruses, Win32/Polip and Win32/Detnat, showed that security and antivirus vendors may have difficulties in detecting these threats. Moreover, Symantec has increasingly observed the use of polymorphic techniques in packers, which could lead to increasingly sophisticated and potentially more damaging malicious code being created [55].

The *Symantec Internet Security Thread Report X*, reports that more malicious code authors may begin to use polymorphic techniques at all levels of malicious code development. Obtaining samples for creating detection signatures of such malware will likely to be very difficult, thus if more malicious code use these techniques, organisations may be increasingly at risk.

Return to Metamorphism?

The return to polymorphic techniques indicates that virus writers are once again trying to make their malicious code more sophisticated and capable of escaping detection from modern antivirus scanners. Polymorphic malware may not be as difficult to write as metamorphic malware, but it is still quite challenging. Cyber criminals are now turning to polymorphism, tomorrow they might be turning to metamorphism. Probably not in the form of the file infecting metamorphic virus presented in this report, but to other types of malware, such as worm, Trojans, and spyware. Another good candidate is the use of metamorphic techniques to malware wrappers, thus creating millions of variants of current malware. Peter Szor, one of the most respectful virus researchers, explains:

> "In theory, viruses could evolve to a level where a virus would be able to export a polymorphic or metamorphic engine of itself for use in another virus or worm. Similarly, viruses would be able to exchange trigger routines and appear in newer combinations.

[3]A wrapper is similar to a packer but can allow a script file, such as JavaScript, to be presented in executable file format

This sounds superficial but the technology is out there to support these kinds of models."

The question is, is the antivirus community ready to deal with a highly metamorphic malware, such as a computer worm or a Trojan? Recent researches, such as [3] and [55], showed that is not. Should malicious code authors turn to this very sophisticated code evolution technique, small businesses, large organisations, or even the home users will be in increasing risk.

7.3 Future Work

Unfortunately, a number of topics could not be discussed in detail because of space and time limitations. If there was more time, the topic of this thesis could change to "Metamorphic Malware" and discuss more about malware obfuscation techniques in current threads. This way, it could give the background of metamorphic virus detection, and at the same time expand the discussion towards other types of malware and obfuscation techniques. The most interesting topic would be polymorphic or metamorphic wrappers, methods of rewrapping exploits in HTMLs or other web technologies. This is interesting because malware authors and cyber criminals are increasingly using the web to distribute their malicious code.

Further development of this research, should also focus on the method of canonicalizing the code of a metamorphic virus. This will decrease significantly the number of different variants an antivirus software has to scan for. If there was an efficient way to get a random generation of a metamorphic malware, canonicalize its code, and turn it into a simplified form that can be recognised by a scanner as member of a specific virus family, its detection would be an easy case. This simplified form is what Lakhotia and Mohammed name as the "zero" form. There are many papers written about this method, with every researcher trying a different way to canonicalize the malware code, but the outcome is the same: A simplified code form, which would be used to extract a detection signature. More information on this method can be found in [56], [51], [57], and [58].

The research for this thesis showed that there is not enough research on other metamorphic malware, except computer viruses. Some research should be done in the field of metamorphism to malware, such as worms, Trojans, and spyware. I have found no references to individual metamorphic worms, although I came up with the term many times. Moreover, most of the metamorphic techniques which are implemented on antivirus scanners are specifically designed to detect file infecting metamorphic viruses. The same situation exists for the experimental detection techniques; they are all designed for and tested on existing metamorphic virus examples. More research is needed on detection techniques that are able to capture metamorphic network worms or spyware.

Bibliography

[1] Alun Michael, Chris Poter, and Andrew Beard. Information security breaches survey 2006. Technical report, PriceWaterhouseCoopers, 2006.

[2] Symantec Security Response Team. Symantec internet security threat report. Technical Report XI, Symantec Corporation, March 2007.

[3] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[4] Fred Cohen. *Computer Viruses.* PhD thesis, University of Southern California, 1986.

[5] Darrell M. Kienzle and Matthew C. Elder. Recent worms: a survey and trends. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 1–10, New York, NY, USA, 2003. ACM Press.

[6] F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, 1987.

[7] Peter Szor. *The Art of Computer Virus Research and Defense.* Addison Wesley Professional, 1 edition, February 2005.

[8] Roger A. Grimes. *Malicious Mobile Code: Virus Protection for Windows.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[9] Mark Ludwig. *The Giant Black Book of Computer Viruses.* American Eagle Publications, Inc, 1995.

[10] Fred Cohen. A formal definition of computer worms and some related results. *Comput. Secur.*, 11(7):641–652, 1992.

[11] Dan Ellis. Worm anatomy and model. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 42–50, New York, NY, USA, 2003. ACM Press.

[12] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 01(4):33–39, 2003.

[13] Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, 1989.

[14] Sophos White Paper. Security threat report 2007. Technical report, Sophos, 2007.

[15] Thomas F. Stafford and Andrew Urbaczewski. Spyware: The ghost in the machine. *Communications of the Association for Information Systems*, 14:291–306, 2004.

[16] Sophos White Paper. Spyware: Securing gateway and endpoint against data theft. Technical report, Sophos, 2007.

[17] Symantec Security Response. Windows rootkit overview. Online.

[18] David Harley and Andrew Lee. The root of all evil? - rootkits revealed. Online.

[19] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Joanna Rutkowska and Alexander Tereshkin. Blue pill project. www.bluepillproject.org, 2007.

[21] Joanna Rutkowska. www.invisiblethings.org, 2006.

[22] Peter Szor. Virus analysis 1: Beast regards. *Virus Bulletin*, June 1999.

[23] Peter Ször and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, September 2001.

[24] Prabhat K. Singh and Arun Lakhotia. Analysis and detection of computer viruses and worms: an annotated bibliography. *SIGPLAN Not.*, 37(2):29–35, 2002.

[25] Frederic Perriot and Peter Ferrie. Principles and practise of x-raying. In *Virus Bulletin Conference*, pages 51–56, September 2004.

[26] Malivanchuk Taras. Epo - what is next? *Virus Bulletin*, pages 8–9, March 2002.

[27] Kaspersky Labs. Kaspersky anti-virus engine technology. On-line, 2005.

[28] Peter Ferrie. Attacks on virtual machines. In *AVAR Conference*, pages 128–143, December 2006.

[29] Fridrik Skulason. Virus encryption techniques. *Virus Bulletin*, pages 13–16, November 1990.

[30] Peter Szor. Junkie memorial. *Virus Bulletin*, pages 6–8, September 1997.

[31] Carey Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, 1997.

[32] Fridrik Skulason. 1260 - the variable virus. *Virus Bulletin*, page 12, March 1990.

[33] Peter Szor. The marburg situation. *Virus Bulletin*, pages 8–10, November 1998.

[34] Ruo Ando, Nguyen Anh Quynh, and Yoshiyasu Takefuji. Resolution based metamorphic computer virus detection using redundancy control strategy. In *WSEAS Conference*, Tenerife, Canary Islands, Spain, December 2005.

[35] Mohamed R. Chouchane and Arun Lakhotia. Using engine signature to detect metamorphic malware. In *WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, pages 73–78, New York, NY, USA, 2006. ACM Press.

[36] Arun Lakhotia, Aditya Kapoor, and Eric Uday Kumar. Are metamorphic computer viruses really invisible? part 1. *Virus Bulletin*, pages 5–7, December 2004.

[37] Zhihong Zuo, Qing-xin Zhu, and Ming-tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on information theory*, 51(8):2962–2966, August 2005.

[38] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane R. Chouchane, and Arun Lakhotia. The design space of metamorphic malware. In *Proceedings of the 2nd International Conference on Information Warfare*, March 2007.

[39] Peter Szor. The new 32-bit medusa. *Virus Bulletin*, pages 8–10, December 2000.

[40] Rodelio G. Finones and Richard t. Fernandez. Solving the metamorphic puzzle. *Virus Bulletin*, pages 14–19, March 2006.

[41] Myles Jordan. Dealing with metamorphism. *Virus Bulletin*, pages 4–6, Octomber 2002.

[42] Peter Ferrie and Peter Szor. Zmist oportunities. *Virus Bulletin*, pages 6–7, March 2001.

[43] Frederic Perriot, Peter Szor, and Peter Ferrie. Striking similarites: Win32/simile and metamorphic virus code. Technical report, Symantec, 2003.

[44] Frederic Perriot. Linux.simile. www.symantec.com, February 2007.

[45] Arun Lakhotia and Prabhat K. Singh. Challenges in getting 'formal' with viruses. *Virus Bulletin*, pages 15–19, September 2003.

[46] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach.* Number 0-8493-8086-3. Taylor & Francis Group, LLC, 1 edition, 2006.

[47] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *DIMVA*, pages 129–143, 2006.

[48] Matt Webster and Grant Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3):149–161, December 2006. DOI: 10.1007/s11416-006-0023-z.

[49] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.

[50] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[51] Arun Lakhotia and Moinuddin Mohammed. Imposing order on program statements to assist anti-virus scanners. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 161–170, Washington, DC, USA, 2004. IEEE Computer Society.

[52] McAfee. Mcafee virtual criminology report. Technical report, McAfee, Inc, July 2005.

[53] Sophos. Security threat report. update 07/2007. Technical report, Sophos, July 2007.

[54] Peter Szor. Personal Communcations, August 2007.

[55] Symantec Security Response Team. Symantec internet security threat report. Technical Report X, Symantec Corporation, September 2006.

[56] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84, Washington, DC, USA, 2006. IEEE Computer Society.

[57] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 235–248, November 2005.

[58] Moinuddin Mohammed and Arun Lakhotia. A method to detect metamorphic computer viruses. *The IEEE Computer Society's Student Magazine*, 10(1):24–36, 2003.