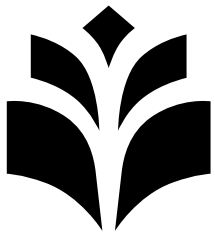
# Computer Viruses

Mika Reinikainen

Master's Thesis



UNIVERSITY OF
EASTERN FINLAND

School of Computing
Computer Science
October 2019

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Kuopio
School of Computing
Computer Science

Reinikainen, Mika: Computer Viruses
Master's Thesis, 83 p.
Supervisor: Keijo Haataja, Ph.D.
October 2019

**Abstract:** A computer virus is a program that infects other programs and is able to self-replicate. A virus generally propagates to other systems, either by physical means such as an USB drive, or through the network. Although some viruses exist that are not inherently malicious, generally they cause harm. At best, viruses are considered as uninvited and undesirable guests on our computers. Computer viruses have been witnessed in many types of systems, such as personal computers, mobile phones and industrial control systems. This Thesis explores computer viruses from four perspectives, based on published literature. First, the history and background of computer viruses is explored by going back 70 years to the very foundations of computer viruses. Second, the inner functionality and offensive mechanisms of viruses are examined. Third, various defense mechanisms against computer viruses are introduced. Finally, an applied perspective is offered through the investigation of two modern computing environments, mobile phones and cars, and their susceptibility to a viral infection.

**Keywords:** Computer virus; Computer virus defense; Anti-virus; Malware;

**ACM CCS (2012)**
**•Security and privacy →Intrusion/anomaly detection and malware mitigation;** *Systems security; Vulnerability scanners; Mobile and wireless security; Social aspects of security and privacy;* Embedded systems security;

# Foreword

Once upon a time, in the lecture halls of the computer science department, I heard a lecturer, whose name now eludes me, say that there is but one property about a Master's Thesis that stands above all others: getting it done. As I am writing these words today, over a decade later, I finally understood what he meant.

I would like to thank the UEF project "Digiteknologian TKI-ympäristöhanke" (Digikeskus-hanke; Hankekoodi: A74338) for providing support for this Thesis.

# Terms and Abbreviations

| Term or Abbreviation | Description |
|---|---|
| ADAS | Advanced Driver Assistance System |
| AI | Artificial Intelligence |
| AP | Application Processor |
| Attacker | A human actor, worm or a virus that has the intent to perform malicious activities on a computer system or a network. |
| Backdoor | A method of accessing a computer system by bypassing its security measures. |
| BCM | Body Control Module |
| BIOS | Basic Input/Output System |
| Botnet | A group of compromised computers that can be remotely controlled by an attacker to perform various tasks. |
| BYOD | Bring Your Own Device |
| Boot sector | A special area on a disk used to initiate the loading of an operating system. |
| CAN | Controller Area Network |
| Cryptovirology | A field studying the use of cryptography for malicious purposes. |
| Cryptoworm | Malware that encrypts user's data for ransom and spreads like a worm. |
| DDoS | Distributed Denial of Service |
| DIDS | Distributed Intrusion Detection System |
| DOS | Disk Operating System |
| ECU | Electronic Control Unit |
| FAT | File Allocation Table |
| IDS | Intrusion Detection System |
| Instrument cluster | Dashboard in a car that houses gauges and displays showing important information to the driver (e.g. speed and RPM). |
| IoT | Internet of Things |

| | |
|---|---|
| IVI | In-Vehicle Infotainment |
| Macro virus | A virus that infects files that employ macro functions. |
| MBR | Master Boot Record |
| MCU | Microcontroller Unit |
| Metamorphic virus | A virus that is able to mutate itself syntactically into different forms. |
| MOST | Media Oriented Systems Transport |
| MMS | Multimedia Messaging Service |
| Multipartite virus | A virus capable of employing multiple infection strategies. |
| NFS | Network File System |
| NIDS | Network Intrusion Detection System |
| OBD | On-Board Diagnostics |
| Polymorphic virus | A virus that is able to encrypt itself and mutate its decryptor function. |
| PLC | Programmable Logic Controller |
| Ransomware | Type of malware used to extort a victim e.g. by encrypting their files and demanding money in exchange for decrypting them. |
| RSU | Road-Side Unit |
| Signature scanning | A method to detect viruses by means of matching static binary patterns in files. |
| SMB | Server Message Block. A communication protocol used to share files. |
| SMS | Short Messaging Service |
| SoC | System on Chip |
| SPI | Serial Peripheral Interface |
| Steganography | The practice of hiding information in plain sight, e.g. in image files. |
| Trojan | A malicious program that is embedded in a legitimate program. A trojan does not self-replicate. |
| UART | Universal Asynchronous Receiver-Transmitter |
| UDP | User Datagram Protocol |
| UEFI | Unified Extensible Firmware Interface |
| V2C | Vehicle-to-Cloud |
| V2I | Vehicle-to-Infrastructure |
| V2P | Vehicle-to-Pedestrian |
| V2V | Vehicle-to-Vehicle |
| V2X | Vehicle-to-Everything |
| VCS | Virus Construction Set |

| | |
|---|---|
| Virus | A computer program, that has the ability to infect other programs and to self-replicate. Generally the term is used to refer to malicious programs, but harmless programs also meet the definition. |

# Contents

# 1. Introduction

A *computer virus* is a parasitic program. Typically a virus resides within one or more files on a computer system. Generally, as a clean file gets infected, it is modified so that the virus code is inserted within the original file (the so called *host* program), with the intent that the execution of the host program triggers the execution of the virus at some point in time. However, some viruses execute themselves entirely without user intervention. The execution flow of a virus includes instructions to scan the computer or network for more victims and to infect them. Some types of viruses do not infect files at all, residing entirely in memory. While differences abound, all computer viruses share one vital characteristic: the ability to replicate themselves.

We shall define a computer virus concisely as follows:

> *A computer virus is a program that infects other programs and self-replicates*

Although the name suggests otherwise, it must be emphasized that the definition does not assign any malicious intent to the virus. Many viruses do cause trouble, though. An *anti-virus* on the other hand is a computer program that is intended to stop or remove a working virus from a system.

The history of computer viruses began in the 1970s, although the term was not in general use at the time. The term was officially coined by Fred Cohen in 1985 (Ludwig, 1998). In the late 1980s, the number and variety of computer viruses started rapidly growing. From then on, the struggle between viruses and anti-viruses has been evolving and with the growing sophistication and complexity of both, there is no end in sight.

Since their early days, computer viruses have evolved into multiple different types with different characteristics. Today, the preferred term to use when referring to malicious programs in a general sense is *malware* (short for malicious software). A malicious computer virus, then, is a type of malware. Several other types of malware exist: *worms*,

*trojans*, *backdoors*, *ransomware*, and many more (AV-Test, 2019). Over the years, the malware landscape has shifted and changed. Whereas in the past viruses and worms were some of the most prevalent malware types, in 2018 they comprised 27.7% of all malware on the Windows platform according to the AV Test Security Report 2018/2019 (AV-Test, 2019). According to the same report, the most prevalent kind of malware on Windows were trojans (62.51%). Furthermore, the report points out that Windows remains the primary target for malware writers, with 51.08% of all malware appearing on the Windows platform in 2018. In Q1/2019 the figures were on the rise, with 74.49% targeting Windows. Overall, a staggering 900 000 000 malware samples have been collected over time by AV-Test by mid-May 2019. The number of malware is steadily raising.

In addition to viruses, worms are also self-replicating programs – with one key difference. The distinguishing feature of worms is that they replicate via networks. Usually worms execute themselves on the target machine automatically without any user intervention, for instance using a vulnerability in server software. In the worst case scenario, this makes the spreading of the worm extremely quick. An example of this is the widely spread CodeRed worm. CodeRed uses a buffer overflow vulnerability in Microsoft IIS web server to inject itself into the memory space of the infected web server process. The hijacked process is then used to create new threads and to spread the worm further (Symantec, 2007a). It is estimated that the worm spread to more than 359 000 computers in less than 14 hours, causing major financial damage at that time (CAIDA, 2001).

In addition to the virus engine itself, many viruses carry a *payload*. The payload is usually the part of the virus that is purposefully apparent to the victim: it can be a simple message displayed to the user, some destructive action such as deleting files, a backdoor installation or a number of other things. Even if a virus does not carry an explicitly harmful payload, it may cause problems as a side effect of its propagation. For instance, the previously mentioned CodeRed worm caused massive network traffic due to the target probes initiated by each newly infected host. It also caused some other devices with web interfaces to behave strangely (CAIDA, 2001).

Since many viruses carry a harmful payload (and even in the absence of such a payload) viruses can be generally thought of as uninvited guests, intruding the privacy of the computer user. From the early days of computer viruses, anti-virus software has been used to disinfect computers which have fallen victims to virus attacks. The problem of neutralizing a virus is two-fold: first the virus must be detected. If a virus is already known to anti-virus vendors or if it behaves in ways that measurably deviate from the norm, it may be easily detectable. On the other hand, an unknown virus, especially the kind that actively evades detection, may be difficult or impossible for an anti-virus

program to find before it is too late. If detection is successful, the system must next be restored to a clean state, that is, the virus must be disinfected. The success of this depends on the nature of the virus. If the virus performs destructive actions, such as deleting or encrypting files, complete recovery of the system might be impossible. In many cases though, the virus can be disabled or completely disinfected from the system either with the help of an anti-virus program or by following detailed instructions regularly published by anti-virus companies.

The main contribution of this Thesis is to provide a comprehensive overview to the topic of computer viruses, shedding light to this darker side of information technology that touches more individuals and institutions every year. With a proper understanding of the problem, one is better equipped to defend against it. In addition to viruses, this Thesis covers worms, due to their similar, self-replicating nature. Other malware types are largely out of scope of this Thesis, however in some cases they are mentioned for illustrative purposes.

The study is mainly based on books and research literature on the subject spanning all the way from the early days of viruses to the present day. The basic principles of viruses (and anti-viruses) can be traced way back to the 80s and 90s and are in many ways relevant to this day. In addition, publications by anti-virus companies are often used as a reference, as they generally contain the most up-to-date and in-depth analyses of virus trends as well as the inner workings of specific viruses.

This Thesis is organized as follows: History and background of computer viruses are discussed in Chapter 2. Chapter 3 delves into different types of viruses and analyzes their operating principles. Chapter 4 focuses on virus defense. Chapter 5 scrutinizes two different types of modern computing devices namely mobile phones and car infotainment systems in terms of susceptibility to virus infection and defense against them. Chapter 6 concludes this study by summarizing the key points and taking a more philosophical look at the subject from a higher vantage point, attempting to answer the ultimate question: between viruses and anti-viruses, who will eventually win?

# 2. Background and History

The fundamentals of computer viruses have been known for a long time. From the simplest possible creations that were devised in the era before global networks to the modern advanced computer worms, they are all based on the same basic goal: the creation of self-replicating entities that have the capability to spread as far as possible (or as far as the author wants) using existing computer systems as their breeding ground. Since computer viruses, malicious or not, generally intrude computers without user acknowledgement, they are perceived as pests. Thus, the co-evolution of viruses and anti-virus products is apparent after viruses first appeared in large scale in personal computers.

This chapter is organized chronologically, picking major events in a time span of approximately 70 years. Although a lot of events have been omitted, the following presentation should serve as a useful overview of computer virus evolution. It also attempts to make a cross-section of the currently prevailing virus types and shows when they emerged.

The overview starts in Section 2.1 in the year 1949, introducing *cellular automata* - a theoretical cornerstone for modern computer viruses. In Section 2.2 the year is 1975 and the first actual implementation of a program resembling present-day viruses in terms of functionality, called PERVADE, is presented. In 1982, Xerox Palo Alto Research Center researchers experimented with the first worm implementations; this is the topic of Section 2.3. Section 2.4 presents the first virus for the *DOS* operating system in year 1986, which was also the first virus to feature stealth abilities. Section 2.5 introduces the Morris worm, which was an important milestone in raising awareness of computer security in 1988. In 1990s, viruses started getting more complex: Section 2.6 presents ”1260”, which was the first *polymorphic virus*, challenging anti-virus products of its time. Section 2.7 describes Concept, the first *macro virus*, appearing in 1995. After this, macro viruses started getting more prevalent. In 1999 Melissa was unleashed. Melissa is described in Section 2.8. In 2001 another major worm outbreak hit the headlines. CodeRed spread around the globe quickly and caused major financial damage. CodeRed

is described in Section 2.9. Section 2.10 presents a different development direction in computer viruses: SymOS/Cabir was the first mobile phone virus spreading wirelessly in the mobile environment, discovered in 2004. Section 2.11 covers Stuxnet – the first worm targeting *Industrial Control Systems*. Finally, Section 2.12 presents WannaCry *cryptoworm*. The chapter concludes with a brief summary and a look into the future.

## 2.1 Cellular Automata (1949)

John von Neumann started the study of self-replicating machines in the late 1940s. These ideas provide the foundation for modern computer viruses, which resemble the self-replication described by von Neumann and others later. Cellular Automata (CA) are structures which can be used to model self-replicating machines. CA are simple structures, usually organized in infinite, one or two dimensional arrays, although higher dimensions are also possible (Sarkar, 2000). The array is comprised of cells and each cell has a certain finite state. Furthermore, each cell has a neighborhood, that is, a predefined area around the cell that the cell "interacts" with. In addition to that, the cells change according to predefined rules. The CA evolves in discrete time units, with each cell changing simultaneously.

An example of such a CA is John Horton Conway's Life (Sarkar, 2000). Life is a two dimensional array, with each cell having two possible states (living/dead). Each cell has a neighborhood of eight cells. As the clock ticks, each cell's new state is determined by these predefined rules:

1. If a cell is dead and there are three living cells in its neighborhood, the cell will be alive in the next cycle.
2. If a cell is alive and there are either two or three cells alive in the neighborhood, the cell survives to the next iteration.
3. If a cell is alive and there are less than two neighbors alive, the cell dies (isolation).
4. If more than four cells in the neighborhood are alive, the cell dies (overpopulation).

In certain initial arrangements, Life produces interesting results. For instance, there are configurations that end up producing the same structure (gliders) over and over, gliding over the surface of the array. This is a primitive example of self-replication. There are some web pages on the Internet that host an interactive web application to demonstrate the functionality of Life, such as the one at bitstorm.org (Martin, 2019).

Such applications are useful for building an intuitive understanding of the sorts of patterns that produce interesting results and the sorts that do not.

Although Cellular Automata do not have a direct application in terms of writing or fighting present-day computer viruses, they demonstrate that the foundations for autonomously replicating "entities" in digital form were laid over 70 years ago. The first known actual implementation of a self-replicating computer program that resembles a contemporary virus appeared in 1975, as a creation known as PERVADE.

## 2.2 PERVADE (1975)

In 1975 John Walker programmed PERVADE (Walker, 1996a), a generic module which could be used by any program by plugging it in the instruction flow. The motivation for the module was not malicious, but the implementation had similar characteristics to a primitive computer virus. Combined with a malicious payload, PERVADE could have probably been used to cause damage. However, the motivation behind PERVADE was entirely different. The writer of the module wanted to use it to spread a totally harmless program called ANIMAL quickly to other users (Walker, 1996b). Since copying files at that time had to be done mostly by means of handing over actual storage media and since he was often requested copies of that particular game, he thought of PERVADE as an easy way of distributing software.

PERVADE runs every time the host program, ANIMAL in this case, executes (Walker, 1996b). PERVADE starts as a separate process. It returns control immediately to the host process so that the program the user thought to be running is started. This is similar to modern computer viruses. It is the most obvious way to prevent the immediate detection of a virus: if the virus itself does not initiate any heavy disk I/O, consume lots of CPU time or give itself away by any other means, it might go undetected at the time it spreads. After starting, PERVADE looks for all accessible directories and copies the host program there. As opposite to file infecting viruses, PERVADE only copies the host program, no existing files are overwritten. Eventually the program would spread to other users' directories and removable tapes. New users would find the program, run it, and spread it further. This way ANIMAL spread to other systems and PERVADE accomplished the goal it was made for.

An obvious hinderance to the spreading of ANIMAL was that in the days it was written, there were no widely spread computer networks which it could utilize to propagate itself. The spreading mechanism of PERVADE was written purely with the intent of

physically transferring it between geographical locations. In 1982, researchers at Xerox Palo Alto Research Center created something that would no longer be constrained by this limitation.

## 2.3 Early Worm Experiments (1982)

In a 1982 paper (Shoch & Hupp, 1982) Xerox Palo Alto Research Center researchers John F. Schoch and Jon A. Hupp explored some early worm implementations. In this research, a worm consists of a limited number of segments. Each segment runs on a separate computer. The worm is constructed by running the initial instance on a single machine. After some initialization procedures, the initial segment tries to locate other idle machines in the network by checking each machine one by one. If an idle machine is found, the worm copies itself to the new host and starts executing there. Each segment must be in communication with the other segments in order to keep track of the number of running segments.

An ominous anecdote is included in the Xerox study: in the early phases of their experiments, a worm was left running over night. Next morning, the whole building was full of dysfunctional machines: the worm initialization code had gotten corrupted and crashed each newly allocated segment. The healthy portion of the worm would not be able to spread a segment to a new machine successfully, desperately trying to spread to new computers over and over again. This resulted in a crash of each computer the worm tried to spread to. However, the worm outbreak was stopped in the experiment by using a failsafe procedure which resulted in a shutdown of the entire worm.

In addition to the worm engine itself, the worm could be filled with any kind of payload. The paper describes some applications, such as a distributed alarm clock and distributed computing of graphic animations.

It seems that one important distinction between this early research and many modern malicious Internet worms today is the relationship of individual worm instances. Modern worms often propagate with the sole purpose of distributing copies of itself without a limit. There might be no intent to co-operate with the other instances. In the Xerox study, the worm segments worked co-operatively, with the goal of maintaining a limited functional set of segments at each point in time. These segments would control one another in a distributed manner. This strategy is also used in some modern worms, for instance to initiate *Distributed Denial of Service (DDoS)* attacks using worm infected machines. In these scenarios, a set of computers running the worm would attack a

target in a synchronized manner, for instance by flooding the target with network traffic. As described in the paper, the initially restricted worm can get out of control due to various malfunctions or anomalies in the network or in the worm code itself. Due to these operational problems, the behavior of the individual worm segments might start to resemble modern worms.

Due to the innocuous nature of the programs discussed so far, there was no use in trying to hide any of the creations from users. The Brain, written in 1986, was different, however. Although the virus itself was relatively harmless, it employed mechanisms which would help the virus avert prying eyes.

## 2.4 Brain (1986)

Brain was the first virus on the DOS operating system (F-Secure, 2019a). Brain is a simple boot sector infector and the original virus did not contain any deliberate malicious code to damage the infected machine. However it did cause problems in certain situations resulting in a loss of data (Kurzban, 1989).

*Boot sector viruses* infect boot sectors of floppy disks and/or hard disks. Brain incorporated some stealth methods to hide its code. The original boot sector and the virus body were hidden on the infected floppy disk in sectors marked as bad in the *File Allocation Table (FAT)*. This fooled DOS into thinking the sectors were useless, even though the virus was using them. Another more advanced stealth technique used by Brain was interrupt hooking. Brain intercepts interrupt 13h based sector read/write functions. Normally, when a software interrupt is generated, a special interrupt handler function is called. In the 8086 series processors, the interrupt vector table is located in the lowest 1024 bytes of memory (Ludwig, 1991). Since normal applications could access all memory in the system at the time, the virus could easily replace entries in the interrupt handler table to point to its own handler. Every time a sector read or write using interrupt 13h (INT 13h/AH=02H for read and INT 13h/AH=03H for write (Brown, 2019)) was executed, the Brain virus first inspected the read contents, determined if the sector belonged to the virus and in case it did, returned the original sector stored into a predefined location on the disk. This is a primitive stealth technique which can be easily bypassed by anti-virus products for instance by means of interrupt tunneling (Ludwig, 1991).

Brain also spawned another interesting phenomenon often seen with computer viruses and worms, namely competition (Ször, 2005). There was another virus called Denzuko which

disinfected Brain. If Denzuko detects a Brain infected diskette, it will remove the Brain virus and infect the disk with itself. Additionally, Denzuko uses the same identification code as Brain so it will not be overwritten by Brain. There are more recent examples of virus competition such as the adversary of CodeRed worm, CodeGreen. CodeGreen removes CodeRed infections and patches the target system for the vulnerability exploited by CodeRed. Some worms even use another worm infection on the same machine to serve their own purposes. Other worms exploit vulnerabilities in their competitors' code to propagate themselves.

## 2.5 Morris Worm (1988)

In late 1988 a quickly spreading worm attacked the Internet (E. H. Spafford, 1989). The functionality of the worm was based on a *blended attack* (Ször, 2005). A blended attack can be loosely defined as a spreading strategy of a worm, which consists of the exploitation of multiple vulnerabilities. The Morris worm (also known as the Internet Worm) utilized several security holes in target systems to invade them and eventually spread to new machines. The worm was not fundamentally malicious, since it did not contain a destructive payload. However, it could have been easily modified to do so. The worm did cause some damage due to increased network traffic and burden on the attacked computers, caused by having multiple instances of the worm running on the same machine. As opposed to most worms nowadays, Morris worm spread on UNIX based computers running on Sun Microsystems Sun 3 or VAX systems. It would have been possible to further expand the worm by adding support for more systems, but at the time only these systems were attacked.

The blended attack employed by Morris consists of several steps. A detailed description of the attack can be found for instance in Eugene H. Spafford's thorough analysis (E. H. Spafford, 1989). The following will outline the basic elements of the attack, summarizing Spafford's article.

The worm exploited a buffer overflow vulnerability in *fingerd* and a bug in the debugging option in *sendmail* program to access the target machine. It also tried to access the target with *rsh*, but since *rshd* might not be running or might not accept the connection the other two options were provided as backup. This is the backbone of Morris worm's blended attack. After establishing a connection to the target machine, a bootstrap program is run on it, which in turn tries to download the rest of the worm from the infecting machine. After the whole body of the worm has been transferred to the target machine, the main body of the worm is run. The main body in turn tries to spread the

worm by utilizing information about other machines in the network found on the target machine.

Eventually the creator of the worm, Robert T. Morris, was convicted with community service and a fine. The incident caused a lot of discussion about security issues, ethics around computers and laws and methods to prevent such incidents in the future. Morris worm was an important milestone in the history of computer security, due to its profound impact on the computer user community.

## 2.6 1260 Virus (1990)

1260 was the first polymorphic virus (Ször, 2005). The basic idea of a polymorphic virus is to modify the virus body as it propagates from computer to computer. The virus body can be encrypted simply by using shifting keys, XORs or similar methods. Also more complex encryption can be used, but this increases the size and complexity of the decryptor. The point of polymorphic viruses is to make the decryptor variable as well so that it cannot be used to detect the virus by means of *signature scanning*. The 1260 virus inserted junk instructions in its decryptor. These instructions did not have functional meaning in the code: they were merely used to make the code appear different from generation to generation.

After 1260, many more polymorphic and as well as *metamorphic viruses* started appearing (Ször, 2005). Anti-virus products naturally had to develop their abilities in order to detect these new stealth techniques. Another disturbing development in 1990 was the appearance of the first virus generation kit, *Virus Construction Set* (VCS). Although VCS was a rather simple kit, the idea of a program that can be used even by novices to create viruses was certainly bad news for anti-virus products. In the coming years, the number and sophistication of virus generation kits grew, contributing to the already large number of in-the-wild computer viruses.

## 2.7 Concept (1995)

Until now, three basic types of malware had been detected: file infector viruses, boot sector viruses and Internet worms. In 1995, a new kind of virus was born: Concept was the first macro virus. Concept infects Microsoft Word documents, which means that it is not directly dependent on the platform the word processor runs on: it works

on IBM PC as well as on Macintosh. The virus was written in wordBASIC language, specifically made for Microsoft Word at the time.

The functionality of the virus has been analyzed by F-Secure (2019f). The virus spreads when the user opens an infected document. The auto-execute feature of macros in Word makes it an easy environment to spread in. As is typical for macro viruses, Concept contains multiple macros: AAAZAO, AAAZFS, AutoOpen and Payload. The AutoOpen macro is run automatically when an infected file is opened (unless disabled). This is how Concepts spreads. Concept also infects the global template file, NORMAL.DOT so that future files created with the "Save as"-command will also become infected. NORMAL.DOT is the template file that is opened when Word is started, making it an ideal place to put virus code in. Intercepting typical user commands such as "Save", "Save as", "New File", "Exit File", "Print File", etc. is a common way for macro viruses to take control and replicate.

The PayLoad macro in Concept does not contain any actual payload, only the following code, probably to indicate that the author's intent was indeed only to create a proof-of-concept virus:

```
Sub MAIN
    REM That's enough to prove my point
End Sub
```

Concept opened a whole new channel for spreading viruses, utilizing the idea that users are more likely to exchange data files than executable programs. In the following years, many more viruses using the same concept were created. In 1996, a virus called XM/Laroux was found that was the first virus infecting Microsoft Excel sheets (Symantec, 2012).

Macro viruses pose a new challenge to anti-virus products as described by Bontchev (1998). Since macro viruses tend to be resistant to errors, the macro code can evolve by itself for instance by means of slight corruption or if an anti-virus program only partially disinfects a file (possibly infected by multiple viruses simultaneously). In some cases the virus nevertheless continues to function, but the signature of the virus changes due to a mutation. This might be enough to prevent an anti-virus product from detecting the virus properly.

## 2.8 Melissa (1999)

Melissa is a so-called mass-mailer virus, meaning that its primary means of spreading is via e-mail. Melissa utilizes a technique called *E-Mail Address Harvesting* (Ször, 2005) to collect e-mail addresses from infected systems.

The functionality of the virus is described in detail by Symantec (2007c). The Melissa virus was found on March 26, 1999. The e-mail addresses are collected from Microsoft Outlook 98 or Outlook 2000 installations. The virus sends itself to the first 50 people in each of the Outlook address books. As already seen with the Concept virus in 1995 (and many others), Melissa uses macros to infect Microsoft Word documents. The virus infects the global template file NORMAL.DOT, similarly to Concept, to infect user documents when they are closed (Document_Close macro). User documents are infected with a Document_Open macro, which executes every time the user opens the document. In other words, an infected document must be opened in order to make the virus spread.

Melissa uses a psychological attack to lure victims to open these documents (Symantec, 2007c). Since the virus spreads by using the e-mail addresses from the user's own address book, it can send e-mails pretending that the sender is actually the victim. The virus creates e-mails using the following subject and message body:

```
Subject: Important Message From <username>
Body: Here is the document you asked for ... don't
      show anyone else ;-)
```

This might lure the recipient to believe that the e-mail originates from a legitimate source and to open the infected document. Many more macro viruses such as VBS/LoveLetter.A@mm and VBS/VBSWG.J (the Anna Kournikova virus) have appeared using similar psychological tricks to make the user do what is needed to spread the virus; and also succeeded at it (Ször, 2005).

Melissa uses some stealth techniques to hide its actions (Symantec, 2007c). Melissa disables the following options in Word to facilitate its spreading:

1. Confirm conversions at open (prompts the user when converting a file from one format to another).
2. Macro virus protection (prompts the user when macros are detected in a document).

3. Prompt to save NORMAL.DOT template (any changes to the global template will be confirmed by the user. Since the virus infects the global template, this option needs to be disabled).

The stealth abilities, e-mail address harvesting and the psychological attack employed by Melissa enabled it to spread throughout the globe quickly, quicker than any other virus before it (FBI, 2019). Eventually, the perpetrator was caught and sentenced to 20 months in prison and fined $5000.

## 2.9 CodeRed (2001)

The CodeRed worm is a dramatic example of a quickly spreading Internet worm: more than 359 000 computers running a vulnerable version of the Microsoft IIS Web Server were infected in less than 14 hours (CAIDA, 2001). The worm came in three variants, the first of which was the least successful. It had a fault in its random IP address generation mechanism, which made it infect the same set of machines in each generation, since the same set of IP addresses was always generated. The second version fixed the shortcomings of the first version and spread far and wide.

CAIDA (2001) provides a detailed description of the attack. The attack phase of the worm consists of several steps depending on the date. If the date is between 1. and 19. of the month, the worm tries to propagate to randomly created IP addresses. This is where the first version of the worm had its bug, creating the same list of IPs in every infection. Between 20. and 28. of the month, the worm attacks the White House (www.whitehouse.gov) web page. Due to this synchronization, all of the infected machines (depending on the system clock) attacked the same site, resulting in a Distributed Denial of Service (DDoS) attack. During the rest of the days of the month, the worm goes to sleep. In case the default language of the attacked computer is American English, the worm attacks the web page contents of the server. The following text is displayed:

```
Welcome to http://www.worm.com !
Hacked By Chinese!
```

Another interesting feature of CodeRed (the first variant) is that it resides entirely in memory (CAIDA, 2001). It does not save itself on the computer as a file, which means that a reboot cures the system. However, the web server vulnerability needs to be

separately patched, otherwise the same machine is susceptible to a new infection. The most important aspect of the in-memory feature of the worm regarding this Thesis is that such programs are not detectable by anti-virus products that only scan or monitor files on the disk. In other words, anti-virus products need to employ memory scanning techniques to detect malware that only reside in memory.

## 2.10 SymbOS/Cabir (2004)

Viruses and worms are by no means restricted to personal computers. Cabir is the first worm that spreads in a mobile phone environment via Bluetooth. Ferrie and Ször (2004) provide information on how it operates. The worm runs on Symbian operating system, which was common in smartphones at the time. Symbian based operating systems for smartphones came in many flavors; Cabir runs only on Nokia's S60 phones. The propagation mechanism of Cabir utilizes Bluetooth, which is a short-range, low-power communication protocol ubiquitous in all kinds of devices nowadays, mobile or not.

The propagation mechanism employed by Cabir is fairly simple, as it only tries to contact the first Bluetooth device it can find. On the other hand, if no Bluetooth devices are nearby, Cabir will drain the battery of the mobile phone easily. A more sophisticated worm might use a bit more conservative tactics. The spreading of the worm depends on user actions: the user has to accept the incoming connection request from the infected phone, dismiss a warning dialog alerting about unverified supplier and finally accept to install the application. Thus, the spreading of the worm can be stopped by the user.

While Cabir was not particularly effective in terms of maximizing its spreading, many much more infectious worms were seen in the coming years on mobile phones.

## 2.11 Stuxnet (2010)

Stuxnet is the first worm to infect Industrial Control Systems (ICS). It was discovered in 2010, although some earlier variants are now known. Stuxnet is a sophisticated worm with multiple infection mechanisms, self-preservation capabilities, peer-to-peer as well as remote control functionality and the capability to attack *Programmable Logic Controllers* (PLC). Stuxnet was analyzed in detail by Symantec in the S32.Stuxnet Dossier (Symantec, 2011). What follows is a brief summary of its operation based on that analysis.

ICSes are generally not directly connected to the Internet. Hence, physical access to the target computers is required for the initial infection. Stuxnet infects vulnerable computers through removable media (e.g., USB drive) by exploiting a vulnerability in Windows which allows Stuxnet to execute itself. From there, Stuxnet attempts to spread to other computers through LAN exploiting other vulnerabilities. Stuxnet is also able to spread internally by infecting other removable media.

PLCs are small, hard real-time capable computers that run programs that control industrial processes. PLCs are generally programmed on an external computer, using dedicated programming software. Stuxnet specifically targets PLCs that are programmed with Siemens SIMATIC Step 7 software. Further, Stuxnet checks that the PLC is connected through Profibus to one of two types of frequency converter drivers. If Stuxnet proceeds with the infection, it changes the frequency of the connected frequency converter drives, which in turn changes the speed of motors connected to them. In other words, Stuxnet is able to alter industrial control processes in the target facility. In addition to being able to infect and control PLCs, Stuxnet is also able to hide its presence on the PLC. If an operator uses the Step 7 programmer to inspect the PLC, Stuxnet will intercept the APIs used by the programmer to query the PLC and hides the malicious code blocks from the operator.

The targets of Stuxnet were specifically chosen and most infections occurred in Iran (58.31%), Indonesia (17.83%) and India (9.96%) (Symantec, 2011). Due to its complexity, significant effort and resources were likely required for its creation, prompting some to speculate that a nation state was behind the attack. The creators of Stuxnet would have had to replicate the internal infrastructure of the targeted ICSes to some extent, so that development and testing of the program were possible. Evidently, Stuxnet was designed for industrial espionage and sabotage.

## 2.12 WannaCry (2017)

WannaCry is a type of ransomware, combined with worm-like spreading mechanisms – also knowns as a cryptoworm. In the research literature, cryptoworms were proposed as early as 1996 by Adam Young and Moti Yung in their paper on *cryptovirology* (Young & Yung, 1996).

The attack is analyzed in detail by Symantec (2017). WannaCry spreads on vulnerable Windows computers exploiting a remote code execution vulnerability in the *SMB* server. Once it manages to infect a host, it encrypts files and presents the user with a dialog to

pay ransom in Bitcoin in exchange for decrypting the user's files. The amount requested is between 300 and 600 USD. To maximize the chances of users paying the ransom, the user interface of the ransomware is translated into multiple languages and instructions are provided on how to perform the transaction. WannaCry spreads both within the subnet of the compromised computer as well as to random IP addresses.

Cryptoworms present a formidable threat to Internet users due to their rapid spread. They have the potential to cause financial losses as well as disruptions to services and people's personal lives. Furthermore, as there is a clear financial motivation to write cryptoworms and other types of ransomware, it is likely that these types of attacks will remain prevalent in the coming years unless effective mitigation strategies are widely deployed.

## 2.13 Summary

The number of computer viruses has skyrocketed since the first wide-spread viruses started to appear in the late 1980s. Currently, the numbers are measured in the hundreds of millions (AV-Test, 2019). Internet offers a perfect means for viruses to propagate, especially via exploits of vulnerabilities which require no user interaction at all. Viruses have developed from simple locally operating file infectors to quickly spreading Internet worms capable of executing in stealth, without user intervention and without touching any files on the system. Extremely advanced viruses in the future might be able to work co-operatively (Symantec, 2001) by utilizing standard interfaces to exchange information on vulnerable systems and propagation strategies. Viruses might be able to dynamically evolve into new forms. As new versions of wide-spread operating systems such as Windows, macOS, Linux and various mobile operating systems gain popularity in the market, new opportunities open up for unexpected future exploits, opening doors for virus writers and challenging security software developers in the years to come.

# 3. Different Types of Viruses

Computer viruses come in many flavors. In order to understand the wide range of different viruses and their characteristics, it is useful to break them down into categories. Chapter 2 already briefly introduced some of the most relevant types of computer viruses, but from a chronological perspective. This chapter will take a different approach and describe the exact nature of these types in more detail. Additionally, some new types are discussed. The breakdown performed in this chapter is not based on an infection strategy, platform dependency or other single trait. The categories are more general, attempting to cover major paradigms. Thus, each type described here will significantly differ from the others in nature. In reality, viruses can be further categorized into tens of different, highly detailed categories; however, this will not serve the purpose of this Thesis and so will not be done here.

The organization of the chapter is as follows: Section 3.1 briefly describes the structure of a virus on a general level. Section 3.2 introduces the first major type of virus, called *file infector*. In Section 3.3, boot sector viruses which used to be prevalent in the days before Internet, are introduced. Section 3.4 examines the basics of worms, a common type of malware nowadays. *Source code viruses* take a whole different approach to propagate; these viruses will be described in Section 3.5. Instead of simply cloning themselves as viruses spread, a more nature-like evolution is seen in many viruses. *Mutating viruses* are the theme of Section 3.6. Section 3.7 introduces the *retrovirus*, a type of virus which specifically attacks anti-viruses. Finally, the chapter ends with a summary in Section 3.8.

## 3.1 Structure of a Virus

Every virus, regardless of type, has at least two major functions: search and spread (Ludwig, 1998). The former function searches through a particular environment for victims to infect. The spread function on the other hand, copies the virus into new

victims. Victims can be executable files, data files, source code, memory space of a process and a number of other things depending on the nature of the virus. These two functions derive directly from the definition of the virus and thus a program without them does not qualify as a virus. The internal details of these functions depend on the environment the virus operates on. Locally operating file infecting viruses typically scan hard disks, removable media and network drives they have access to by iterating the devices one way or the other. One way to search for files to infect on a local machine is to systematically scan the current directory or a predefined set of directories and infect all files the virus is capable of infecting. Another method utilized by many boot sector viruses (see Section 3.3) is to check files on-access. That is, the virus intercepts the control flow when another application accesses a file, checks if the file is infectable and not already infected by this particular virus and then proceeds with the infection. Finally, the control is returned to the application that initially tried to access the file.

A common, but optional component of a virus is payload (Ludwig, 1998). Payload can be anything the writer of the virus wants the virus to perform on infection. Common payloads are messages displayed to the user, intentionally destructive actions, installation of additional malware such as backdoors or spyware, among others. Many viruses contain no payload at all. However, since every virus contains search and spread functions, those functions always cause additional workload on the attacked system. Thus, the existence of a virus alone can be considered harmful. Many viruses also accidentally destroy data, for instance due to bugs.

Since the main goal of most viruses in the wild is to spread as far and wide as possible, another common set of features in viruses are anti-detection measures (Ször, 2005). In fact, the two vital components of a virus, search and spread, can be built with anti-detection in mind. Consider for instance a virus which opens and investigates every file on the system on execution as it looks for the next victim to infect. On a system with huge hard drives, this would most likely lead the user to suspect that something strange is going on. A cunning virus might use more careful tactics, for instance activating only when the computer is idle. Another often used self-defense mechanism is self-mutation, which is examined more closely in Section 3.6.

The components discussed in this chapter are basically independent of the details of the underlying platform. Most viruses, no matter how they are implemented, employ some or all of the components discussed here.

## 3.2 File Infector Viruses

File infectors are the oldest type of computer viruses. Several different categories of file infector viruses exist. The following subsections will be organized based on the infection method of the virus, that is, how the virus attaches itself to its victim. Some of these types leave the victim completely operational (assuming that the virus does not accidentally destroy it) and some destroy them beyond repair. Some of the most common types will be described here, starting with the simplest of all, the overwriting virus in Subsection 3.2.1. Subsection 3.2.2 describes appending/prepending viruses. These viruses leave the original file unharmed and add their own code to the beginning or end of the infected file. Inserting viruses will be described in Subsection 3.2.3. Companion viruses leave the original file completely intact, save for the file name, fooling user to think that correct file is executed even though virus code is executed first before the actual untouched original file is finally executed. This type will be described in Subsection 3.2.4. Finally, Subsection 3.2.5 examines how non-executable files can be used by a virus.

### 3.2.1 Overwriting Viruses

Overwriting virus is the most unforgiving virus type: it effectively destroys files it infects (Ludwig, 1998). As the name suggests, the virus overwrites the contents of the infected file with itself. Usually viruses are small, so only a part of the original file will be overwritten. This is enough to destroy it and basically leaves an anti-virus product with no means to restore the system to its original state. Only clean backups might save the system.

### 3.2.2 Appending/Prepending Viruses

Appending/prepending viruses preserve the contents of the infected file and add their own code either to the beginning or to the end of the file. Thus, the size and date of the original file might change, which could trigger an alarm in an anti-virus product unless the virus is careful and restores these attributes to their original state (Ludwig, 1998). In order to get executed at some point in time, an appending virus needs to modify the original code: it will replace the first few bytes of the infected program with a jump instruction to the end of the file, the virus body. This way, the virus gets control when the host is executed. As a new file gets infected, the first bytes of the original file will be stored in the virus body so that the virus can restore them when it is done executing and

return control to the host program. From an anti-virus perspective, jump instructions in the beginning of the execution flow might be suspicious and could alert a heuristic scanner.

### 3.2.3 Inserting Viruses

Inserting viruses utilize slack space in files (Ször, 2005). These kind of viruses add their code into such areas of their victims, which serve no purpose (or very little purpose). Thus, the size of the original file does not change, as with appending/prepending viruses. The virus might also be broken down into several pieces, with each piece filling a hole in the original file. When the virus gets control, the *loader* piece is executed, which assembles the virus from the other pieces scattered around the original file to the memory.

### 3.2.4 Companion Viruses

The basic idea of companion viruses is simple: the original program and the virus reside in different files, but when the user thinks he is executing his own file, the virus will be executed. To understand how this is possible, let us consider a simple example in a by-gone operating system DOS (Stavroulakis & Stamp, 2010). In DOS, there are 3 types of executable files, .COM, .EXE and .BAT files. If a directory contains for instance files a.COM, a.EXE and a.BAT and the user executes command "a", a.COM will be executed. COM has precedence over EXE and EXE has precedence over BAT. The opportunity for an exploit lurks here: a file containing the virus can be created with the same name as the victim file, but with an extension that has precedence over the extension of the victim. Now when the user thinks he is executing his own file, the virus will be executed instead. The virus can then execute the original program after it is done to further fool the user. This is called the *preemptive execution* method.

Another similar companion virus type is the *PATH virus* (Stavroulakis & Stamp, 2010). PATH viruses work similarly to the method described before but utilize the fact that paths are searched in a certain order when executables are run. On UNIX based systems, the PATH environment variable can be used to define search precedence for paths.

A companion virus could also rename the original file's extension to a bogus name, such as .CON or .EX and use the original name for the virus (Stavroulakis & Stamp, 2010). Since all companion viruses need two files to operate, the number of new files

might give away the virus immediately to a user doing a simple directory listing. Some companion viruses hide the original files so that this anomaly would not be detected.

### 3.2.5 Viruses in Non-Executable Files

In order for a virus to execute, it generally resides in an executable file of some sort. However, it is possible for a virus to hide in a non-executable file as well. In this case, the virus is in a dormant state until another program is used to parse said file. Then, the virus may be able to exploit a vulnerability in the parser.

An example of such an attack is detailed by Salomon (2010). In this attack, a vulnerability in the JPEG parser of certain Microsoft products is exploited. The parser was vulnerable to a buffer overflow attack, so that a specifically crafted JPEG file could be used to trick the parser into executing malicious code. This vulnerability was reported in the Microsoft Security Bulletin MS04-028 in 2004. It seems that no wide-spread exploitation of the vulnerability happened at the time of the incident.

Another possible way to utilize non-executables is through *steganography*. Steganography is concerned with hiding information within legitimate files, so that it is difficult to discern a modified file from the original. For example, it is possible to hide information into images by making small modifications to the color values of a pixel. For a 24-bit image, changing the least significant bit of each of the colors (red, green and blue) will allow the encoding of 3 bits of information per pixel (Wikipedia, 2019b). Steganography has in fact been used by malware, such as Gatak/Stegoloader trojan. Gatak/Stegoloader hides control commands encoded in PNG files, which it downloads separately only after the initial installation procedure has been successfully completed and a decision has been made to proceed with the infection (in some cases it will not, such as when it determines to be running under the *Wine* emulator) (Virus Bulletin, 2016).

The motivation for hiding malicious code in non-executables is that it may be difficult for anti-virus programs to discern an infected file from a non-infected one. In some cases, non-executable files or files that are generally not considered suspicious may not be scanned at all in order for the scan to complete faster. Naturally, the infected non-executable files require an executable malware program running in the system to decode and utilize the hidden messages – these programs may be easier for an anti-virus to detect, e.g. by using some of the techniques that will be described in Chapter 4.

## 3.3 Boot Sector Viruses

Boot sector viruses used to be very common in the early days of computer viruses, especially with the DOS family of operating systems. Since floppy disks were often exchanged before the introduction of the Internet, infected floppy disks could easily infect the hard disk when accidentally left in the drive at boot time. While the following description of boot sector viruses contains technical details that have since been largely superseded by newer technological advancements (particularly *UEFI* replacing *BIOS*), the basic operational principles of boot sector viruses remain the same.

A boot sector is a special area on a disk used to initiate the loading of an operating system (Salomon, 2010). When a computer starts, the first steps are performed from BIOS ROM, a read-only memory location. The boot-up code in BIOS contains basic operations necessary for a successful boot, such as checking what hardware is present and performing various tests, such as a memory test (Ludwig, 1998). However, no operating system specific boot code is hard-coded into BIOS, since this would make the system extremely inflexible. Instead, BIOS relies on the fact that instructions to load the operating system are located on the first sector of the hard/floppy disk. If a hard disk has multiple partitions, possibly with multiple operating systems, the first sector of the hard disk contains information about the other partitions. This sector is called the *Master Boot Record* (MBR). The actual operating system dependent boot sector is the first sector in the respective partition. Viruses can infect both MBRs and regular boot sectors.

Single sectors are typically small (512 bytes), which means that a virus might not fit in one sector. Some viruses utilize multiple sectors, which the first sector loads and assembles into memory for execution. Some boot sector viruses store the original boot sector elsewhere on the disk and pass control to it after execution. For instance, Stoned virus which appeared in 1990 stores the original boot sector in Track 0, Head 1, Sector 3 on floppy disks and Cylinder 0, Head 0, Sector 7 on hard disks. The virus works fine with 360K floppy disks but causes problems with larger disks since the sector it uses to store the original boot sector corresponds to a sector in the root directory. If enough files are stored in the root directory, the virus will overwrite these root directory entries (Ludwig, 1998).

Other boot sector viruses do not store the original boot sector at all but instead perform the necessary boot operations themselves (Ször, 2005). This means that the virus cannot be disinfected by restoring the exact original boot sector. Anti-virus products might carry a general purpose boot sector to replace the infected one. MBRs could be easily replaced but operating system dependent boot sectors might be harder to restore.

Typically boot sector viruses stay resident in memory after they are loaded. Strictly speaking a boot sector virus could be able to spread without being memory-resident, simply by infecting media only when it is executed at boot time (Ludwig, 1998). To be more efficient, the boot sector virus can stay in memory, hook disk read/write interrupts, and infect all future media that is accessed from the computer. The virus can also install stealth methods to prevent the user or an anti-virus program from seeing the virus with regular disk access methods.

A special class of boot sector viruses is one that employs regular file infection capabilities. For instance, Tequila.A virus infects the MBR of a hard disk when it is executed for the first time (Symantec, 2007b). After reboot, the virus becomes memory resident and starts infecting .EXE files. Viruses that employ multiple infection strategies are called *multipartite viruses* (Salomon, 2010).

In 2018 a particularly disconcerting type of malware was found: Lojax (ESET, 2018). Lojax is a rootkit that is able to infect UEFI firmware. UEFI infections are difficult to detect and also to disinfect, requiring the user to reflash the firmware.

## 3.4 Worms

A worm can be thought of as a special kind of a virus. The common denominator between worms is that they use networks to propagate. Also, many worms do not require any user intervention to execute: they exploit a vulnerability in the system to execute themselves automatically. This makes the spreading of worms very fast in some cases.

This chapter will introduce the basic components and functions of a typical worm. As mentioned in Section 3.1, every virus has search and spread components. Worms are no different: Subsections 3.4.1 and 3.4.2 will explain the functionality of the search and spread components required by worms. Subsection 3.4.3 will describe some additional components often seen in worms.

### 3.4.1 Search Mechanisms for Worms

Chapter 3.1 introduced some basic components of a computer virus. These components naturally need to be implemented in worms as well. Worms operating on the Internet might scan the infected computer similarly to a locally operating virus and in addition scan the network for more vulnerable machines. IP addresses might be generated

randomly or based on some heuristics. IP addresses and access information could also be collected from the local machine, similarly to Morris worm discussed in Section 2.5.

Another target selection method is to use a hit-list (Zou, Towsley, Gong, & Cai, 2005) (Staniford, Paxson, & Weaver, 2002), which is a predefined set of vulnerable addresses contained within the worm code. When the worm initiates the attack, the first instance of the worm hits one computer on the hit-list. Then, the worm attempts to infect a computer on the hit-list and when successful, divides the list in two. The worm keeps one half and sends the other half to the second infected computer. By dividing the hit-list at the point of infection, multiple infections of the same set of computers are prevented. The division also reduces the size of the hit-list quickly, which further benefits the worm. By using a hit-list combined with a scanning method, a potentially large initial number of infected hosts can be established in the beginning, speeding up the propagation of the worm. The generation of the hit-list is a whole other matter and will not be further examined here. See (Staniford et al., 2002) for further details.

A more conservative method of scanning iterates only definite IP address ranges. A simple example of this was demonstrated by CodeRed II (CAIDA, 2001) worm, which skipped local loopback addresses 127.0.0.1/8 and 224.0.0.0/8 multicast addresses. Reducing the address space to scan, the scan can be completed faster and as a consequence the worm spreads faster.

Another method restricts the address space by utilizing information from Border Gateway Protocol (BGP) routing tables (Zou et al., 2005). BGP data allows the worm to skip non-routable IP addresses. This might significantly reduce the address space to scan. A more targeted way to utilize IP address information is the ability to choose victims selectively, for instance based on geographical location, company or Internet service provider. Furthermore, a worm could harvest information about the computer and network connection it is running on, utilizing this information to control its scanning speed. A worm running on a high-performance server with a fast Internet connection could support more threads to probe other machines.

Depending on the type of the worm, many more search mechanisms exist. For instance, mass-mailer worms such as Melissa seen in Section 2.8 use e-mail address harvesting to search for hosts to infect. Other worms utilize the scripting support of *mIRC* instant messaging client to search for new victims to infect.

### 3.4.2 Spreading Mechanisms for Worms

Since many worms require no user intervention to work, spreading can be as simple as sending a simple *UDP* packet to the infected host. W32/Witty is a good example of this (Ferrie, Perriot, & Ször, 2004) (Weaver & Ellis, 2004). Witty is a small worm, consisting of only 647 bytes. The search mechanism of the worm consists of a (pseudo) random IP address generation. The spreading mechanism on the other hand is extremely simple and effective: Witty uses a buffer overflow vulnerability in *Internet Security Systems* (ISS) security products. These security products inspect the contents of incoming packets for suspicious content. Witty exploits a vulnerability in the *ICQ* instant messaging application traffic analyzer. By sending a specially crafted UDP packet to port 4000, the ICQ traffic analyzer is invoked and a stack overflow is generated by the packet, enabling Witty to run its own code. These kind of spreading mechanisms enable extremely fast spreading: after a successful attack, the only limiting factor is the amount of bandwidth available. For more detailed information on the intricacies of buffer overflow attacks, see (Ször, 2005, pp. 368-384).

### 3.4.3 Additional Components

The most vital components of a worm, the search and spread functions, take care of the rapid propagation of the worm. However, in order to make the worm even more dangerous, more components can be added to it. In this section remote control interfaces, update interfaces and life-cycle managers are examined.

A remote control interface allows the master of a set of worms to control the worm network. A remote control interface might be used for example to initiate a synchronized Distributed Denial of Service (DDos) attack using multiple infected systems to overwhelm a target system with flooded network traffic. As an example, Linux/Slapper has an advanced distributed control interface (Perriot & Ször, 2003). Slapper builds a peer-to-peer network between all hosts it infects. As a new system gets infected, a list of other nodes in the network is delivered to the new system. The infecting machine broadcasts the address of the new instance to the rest of the network. The remote control interface of Slapper allows the attacker to control the entire network from a single instance. Slapper's remote control interface supports several DDoS flood attacks and the execution of arbitrary code on the infected nodes.

Update interfaces on the other hand are used to change the behavior of the worm or to update the infection strategy arsenal of a virus. For instance, upon discovering a new vulnerability, the attacker might build and release an update module for an existing

worm network, that could utilize the vulnerability to spread further. An example of a virus that utilizes a remote server to provide updates is W95/Babylonia (Ször, 2005, pp. 345-346). Babylonia relies on the existence of a single external server, originally located in Japan. As Babylonia infects a computer, it installs itself as a system service. The active virus then checks if an Internet connection is active and tries to download a list containing the available modules for download. The server provides modules to update/reinstall the virus, a payload module, a mIRC infector and a module to keep track of the number of infected machines. As the virus uses a centralized repository for its updates, the updating mechanism was simple to disable by taking the repository offline. A more sophisticated method of providing updates was seen in W95/Hybris worm (Ször, 2005, pp. 346-350). Similarly to Babylonia, Hybris can expand its functionality by means of additional plugins. However, Hybris does not use a single update server. Instead, it can update itself via a web server similarly to Babylonia and via newsgroups (alt.comp.virus). Each instance is able to upload its modules to the newsgroup and download new, possibly updated, modules from the newsgroup. The updates are also signed and encrypted, making it harder to create bogus updates which could harm the worm.

Life-cycle managers change the behavior of the virus as a function of time (Ször, 2005, p. 316). A time-variant behavior could be used for several different purposes: to trigger synchronized attacks, to activate the virus only on a certain date (for example the Michelangelo virus, which activated on his birthday, March 6 (F-Secure, 2019b)), to attack different targets depending on the date, to delay the activation of the virus to obscure its origins, etc. An example of this was seen in Section 2.9 with CodeRed. CodeRed had three stages in its life-cycle: 1) propagation; 2) Distributed Denial of Service attack; and 3) sleep. The behavior of the worm varied depending on the day of the month.

## 3.5 Source Code Viruses

Source code viruses (Ludwig, 1998, p. 291) are file infecting viruses, that target source code files. Although these type of viruses are relatively rarely seen in the wild, their unique nature deserves an introduction. Source code viruses also present new challenges to anti-virus products.

In a 1984 article Ken Thompson presented the basic ideas of a source code virus, although he himself did not call his creation a virus (Thompson, 1984). The paper describes a C compiler, which is modified in such a manner that it recognizes when

either of the following programs is compiled: another C compiler or a login program. The compiler is able to infect them both with different payloads. The actual payload of the virus is aimed at the login program. Every time the login program is compiled with the infected C compiler, it is modified to allow the attacker to log in as any user in the system. On the other hand, when a C compiler gets compiled, the C compiler infector and the login program infector are inserted into the new C compiler, allowing the ”virus” to spread further. The remarkable idea about this is that the infected C compiler contains its own source code in binary form. Thus, it can insert actual C code at compile time into the sources being compiled, propagating the virus further.

A more general kind of source code infector could scan the entire system for source code files, possibly in several languages and insert viral code into the sources (Ludwig, 1998). The idea is to insert one or more function calls of the virus into the instruction flow of the original program. For instance, in a program written in C, the main() function could be located and a call to the virus added as the first call in the whole program:

```
#include <stdio.h>
#include "source_code_virus.h"

int main(void)
{
  call_to_virus();
  printf("Hello world!\n");
  return 0;
}
```

In this example, the whole source code of the virus would be located in the header file source_code_virus.h to make the virus infection a bit less obvious upon inspecting the files. When the virus inserts a function call to its own code, it needs to be careful that the code will really be executed. In the previous example the code will definitely execute with the host, but infecting functions that are never called or accidentally infecting a commented block might prevent the virus from executing at all. Alternatively, the virus might insert the function call to a very frequently called function, possibly severing the performance of the program as a result.

A carefully written source code virus is not strictly dependent on any particular platform. As long as the virus code itself is portable, the virus might function in a variety of different environments as the infected source files are moved to different systems and compiled there. Source code viruses also appear different in binary form as they migrate

to new systems with possibly different compiling tools. The compiler, compiler options, compiler version and language all affect the final binary translation of the virus, making detection in binary form difficult (Ször, 2005, pp. 103-104).

## 3.6 Mutating Viruses

As the main goal of viruses is usually rapid and undetected spreading, viruses need to employ several techniques to counter the attacks of an anti-virus. For instance, the Brain virus introduced in Section 2.4, used interrupt hooking to disable an anti-virus from detecting Brain on an infected disk. The method used by Brain was simple and easily countered by an anti-virus. As anti-viruses get more advanced, more advanced techniques are needed in viruses as well. In this chapter, code evolution techniques are introduced.

Encryption is a code evolution method used to armor a virus against detection and analysis. An encrypted virus is effective against manual disassembly, preventing the analyzer to directly see the contents of the virus. Encryption method can be a simple XOR, resulting in weak encryption with short keys or some stronger encryption algorithm, possibly implemented by the virus writer or utilizing an existing library on the system such as *CryptoAPI* or *OpenSSL* (Symantec, 2006). The main problem with simple encryption is that the decryptor can be used to detect the virus. Also, if the decryption key is present in the virus body, the decryption of the virus body is trivial by an anti-virus. Some viruses indeed leave the key out of the virus entirely. The virus might use a brute-force search to find the right key (Random Decryption Algorithm) or some other technique, such as delivering the key separately (perhaps at a later point in time) in a non-executable file hidden with steganography (see Section 3.2.5). Some of these techniques are explained in more detail by Ször (2005, pp. 253-258). The next step in complexity are polymorphic viruses, which attempt to protect the decryptor as well.

A polymorphic virus creates varying decryptors to prevent an anti-virus from detecting the virus based on the decryptor. The decryptor is changed in each generation by adding random instructions in the decryptor, making it appear different. For instance, adding NOP instructions has no functional effect and they can be added anywhere in the code. PUSH/POP calls push and pop values to/from the stack and can be used to add do-nothing code. Also instructions such as CLC (Clear Carry Flag), CMC (Complement Carry Flag), STC (Set Carry Flag) and similar flag register modifying instructions could be added, paying careful attention that the additions do not change the functionality of the code (for a full instruction set reference for Intel hardware, see (Intel, 2019)).

A polymorphic virus typically contains a *random code generator*, which is responsible for creating random code that can be added to the decryptor. A good random code generator creates classes of random code. For instance, the sample decryptor mentioned by Ludwig (1998) is able to create three classes of random code: code that modifies no registers and no flags (such as NOP or PUSH/POP pairs), code that modifies flags (CLC, CMC, STC, etc.) and code that modifies a single register (for instance MOV reg, value). Breaking the code down into classes is useful, since in some places random instructions that leave registers and flags intact might be required, whereas in some places the state of some flags or registers might be irrelevant. The more versatile the code generator, the more difficult the polymorphic virus will be to handle by an anti-virus. An obvious shortcoming in polymorphic viruses is that if an anti-virus manages to decrypt the virus body, it will always look the same, making exact detection of the virus possible. Metamorphic viruses on the other hand, are able to truly mutate themselves.

A metamorphic virus is syntactically different in each generation. The virus still performs the same functions, i.e preserves the semantics, but changes its appearance. The decryptor part in polymorphic viruses can be thought of as a metamorphic component. Similarly, the same principle can be applied to a virus as a whole: a simple metamorphic virus has a static skeleton of code, which is surrounded by random garbage instructions. Typical methods used by metamorphic viruses are the following:

1. Instruction reordering (Webster & Malcolm, 2006): Non-related instructions can be swapped in the code, without affecting the functionality.

2. Pseudo-branching (Webster & Malcolm, 2006): The code could contain multiple semantically equivalent branches, performing the same functions. The additional branches qualify as garbage, but alter the signature of the virus.

3. Jump instructions (Ször, 2005, p. 274): The instructions of the virus can be arranged in different permutations and linked by jump instructions.

4. Change used registers (Ször, 2005, p. 271): For instance, when doing simple arithmetics, such as A + B or A – B, the registers used to calculate these can be changed, resulting in different binary code.

5. Subroutine permutation (Ször, 2005, pp. 271-272): The code of the virus can be divided into several coherent functions and the locations of these functions can be changed within the body of the virus.

6. Add garbage instructions (Ször, 2005, p. 273).

7. Mutate existing instructions into equivalent, but syntactically different instructions (Ször, 2005, p. 273) (Ludwig, 1998, p. 441). For instance,

   ```
   OR EAX, EAX
   ```

   can be replaced by

   ```
   TEST EAX, EAX
   ```

   to check if EAX register is zero. Another example expands an instruction into two equivalent instructions:

   ```
   MOV EAX, A
   ```

   could be transformed into

   ```
   MOV EAX, B
   ADD EAX, C
   ```

   resulting in EAX having the value A (B + C = A). The possibilities are endless.

Code evolution methods present a challenge to static virus scanning products but many techniques have been developed to fight viruses that use them. Some of these techniques will be presented in Chapter 4.

## 3.7 Retroviruses

A special class of viruses called retroviruses take their self-defense mechanisms one step further. As the saying goes, "best defense is a good offense", these viruses can actively attack anti-virus products instead of passively trying to protect themselves from being detected by one. Retrovirus attacks can range from disruptions in the functioning of the anti-virus to total failure, possibly eliminating the entire product from the system. Similarly to viruses, anti-virus products must protect themselves against virus attacks. For instance the methods described in the previous chapter could be applied to an anti-virus as well. This way, a virus could not easily detect an anti-virus in memory or on disk and eliminate it. The purpose of this chapter is to describe some possible retrovirus attacks and their impacts on anti-virus products.

The first action a retrovirus must take is to locate an anti-virus on the system. Second, it must decide what action to take. A polite virus might cancel its search and spread

operations after detecting an anti-virus, preventing the anti-virus from triggering its alarms (Ludwig, 1998, p. 468). A more aggressive virus might disable the anti-virus by killing the process (if one happens to be running) and/or by deleting the related files on the disk. A more subtle approach is to modify the anti-virus in such a way, that it fails to detect the virus (Hyppönen, 1994). This can be potentially done either by modifying the anti-virus itself, the virus database or other related files.

A virus could also attack other security products compromising its existence or propagation, such as firewalls, Intrusion Detection Systems, behavior blockers and integrity checkers. For instance, integrity checkers must somehow store the integrity information so that file system integrity can be compared against this information at a later point in time. IDEA.6155 virus (Ször, 1998) demonstrates one way to attack an integrity checker, which stores its integrity checksums in a file called ANTI-VIR.DAT. As the virus infects a new file, it patches the corresponding entry in the integrity checksum file, modifying a single character in the file name. This causes the integrity checker to add a new entry for the infected program, effectively missing the modification that occurred to the infected file.

Another real-life example, MTX (F-Secure, 2019h), uses a different approach. As it attempts to infect a machine, it checks if an anti-virus program is running. MTX contains a list of popular anti-virus product names and the check is done against that list. If any of the anti-virus products on the list are detected, the virus politely exits and does nothing. If the virus decides to proceed, it installs a listener to WSOCK32.dll send() function. By doing so, it is able to monitor outbound Internet traffic. The retro-attack is performed at this point: the virus prevents the user from visiting web sites of anti-virus vendors by blocking access to addresses that match an entry in a predefined list of strings. For instance, traffic to all addresses containing strings ”f-se” or ”afee” is blocked (representing the sites for F-Secure and McAfee, respectively).

Not only does the disabling of an anti-virus product benefit the retrovirus itself, it benefits all malicious software since the system becomes more vulnerable to all kinds of attacks. Thus, for the sake of general security, an effective anti-virus product must be able to employ self-defense mechanisms to protect itself and the system it runs on.

## 3.8 Summary

This chapter provided an in-depth look into the structure of a virus to explain its inner workings. In addition, some of the major types of computer viruses were covered.

Different ways for a virus to infect a computer were investigated, along with propagation strategies using files, networks and various other types of media. The granularity of the categorization used in this chapter was intentionally coarse. Indeed, many more types of viruses exist and different variants are found in the wild every year. Due to the scope of this Thesis, this chapter focused only on the most interesting, major types, providing a foundation for the upcoming chapters.

# 4. Principle Techniques of Virus Detection and Defense

A computer virus can be loosely defined as a program that infects other programs and self-replicates to as many new hosts as possible. Typically, the goal of a virus is also to execute a payload of some kind (be it malicious or harmless). The successful completion of these activities depends on two premises (referred to as Premise 1 and Premise 2 from now on):

- Premise 1: The target system(s) must allow the virus to execute, allowing it to drop its payload and to look for new victims. The nature of the victims depends on the nature of the virus: they might be other executable files on the system, data files or remote machines.
- Premise 2: The target system(s) must allow the virus to spread itself, i.e. infect victims with a possibly evolved copy of the virus. The act of spreading the virus can happen in many ways, depending on the nature of the virus. Section 4.1 discusses this premise in more detail.

If a system fulfills Premise 1 but not Premise 2, the infection is contained to a single machine: the virus will not propagate further, but might cause damage on the machine it manages to infect. A natural way to defend against viruses is to attack one or both of these requirements. A system in complete isolation with no information coming in or going out would render spreading of the virus impossible (Cohen, 1987). Thus, the system would be safe from virus attacks. Such systems are abundant in today's embedded applications. On the other hand, a non-isolated system could allow data sharing per se, but it could prevent the virus from executing or otherwise restrict its operation. For instance, in UNIX systems users traditionally cannot access each others files, thus limiting the viruses' ability to operate: other users' and system files will be left uninfected. However the user who executes the virus will grant it permission to infect

the files the user has access to. This example illustrates the problem of access restriction: there is always a tradeoff between accessibility and security. Many real world situations require sharing and executing of foreign data, which limits the effectiveness of this defense approach. This chapter will examine virus detection and defense mechanisms, which can be used in today's interconnected, multi-purpose and multi-user computer systems, aiming for high security but at the same time attempting to minimize the degrading effects on usability.

The scope of the various techniques introduced in this chapter is purposefully broad, ranging from traditional virus signature scanning to more general intrusion detection techniques. The term attacker in this context can not be merely restricted to self-replicating malicious computer programs. The attacks could as well be completely or partly performed by a human actor. For instance, a buffer overflow vulnerability in some server software could be abused to launch a manual attack on only one target (i.e. the attack would not spread to other targets). Similar defense mechanisms may apply for both types of attacks, which is why a comprehensive view of defense mechanisms is needed.

Another reason for selecting a wide scope of defense mechanisms is to better illustrate one of the most central observations of this Thesis: any single virus defense mechanism tends to be inadequate and that no absolute, yet practical, defense mechanism exists. Furthermore, in order to understand the requirements for proper defense, taking a wider perspective to information security field is necessary.

This chapter is organized as follows: Section 4.1 provides an overview of virus propagation mechanisms. Section 4.2 examines the principles of host based static signature scanning techniques. Section 4.3 introduces heuristic methods to find viruses without explicit a priori information about a virus. In Section 4.4 various defense mechanisms based on behaviour monitoring are explored. Section 4.5 introduces firewalls. The social dimension of computer virus defense is examined in Section 4.6. Finally, the chapter concludes with a summary in Section 4.7.

## 4.1 Virus Propagation Mechanisms

In order to better understand the requirements for proper protection against computer viruses, this section summarizes the principle mechanisms of virus propagation – in other words, how a computer gets compromised by a virus. In essence, this chapter will

explore Premise 2 (see introduction of Chapter 4) in more detail, laying a foundation for the subsequent discussion on virus defense mechanisms.

Virus propagation methods can be categorized into several groups. An ideal defense system would naturally cover all categories, thus preventing any kind of virus entering a system. For the purposes of this Thesis, the following groups were identified. The purpose of the categorization is to clarify the main requirements for effective defense measures.

1. Viruses spreading as e-mail attachments: The attachment might have to be opened or executed by the user for the virus to execute. In some cases, merely viewing the e-mail with specific e-mail viewers might execute the virus.
2. Worms spreading through a network: Typically, a worm spreads in a network without user intervention by exploiting a vulnerability that allows the worm to inject itself into a target computer.
3. System breach and manual planting: A malicious program or a human perpetrator might break into a system one way or the other and plant various types of malware, such as viruses.
4. Transferring via physical media: Data on a physical media might be intentionally or unintentionally infected with a virus. Thus, a human actor might cause the virus to spread, by physically transferring the infected media to new locations.
5. Local spreading: The virus can multiply on a host locally within the limits of access permissions. For instance, the virus might infect all user files on a UNIX system it has access to, or if the root account is compromised, other users' files as well. In addition, locally mounted network drives (*NFS* for instance) might allow the virus to spread across systems.

These attack vectors can be protected by various means. The following sections will describe the principle methods of protecting a system by addressing the previously mentioned premises. At the end of each section, pros and cons for the respective defense mechanisms are identified.

## 4.2 Static Signature Scanning

Static signature scanning (Ször, 2005, p. 428) (from now on, simply "signature scanning") is the traditional approach in anti-virus products to detect viruses preemptively in an

accurate manner – in other words, signature scanning can be used to attack Premise 1 (see introduction of Chapter 4). Signature scanning as a technique is quite universal and can be applied in many situations and for many purposes. The following discussion will focus on signature scanning as it is applied to virus defense.

This section is broken down into the following subsections: Subsection 4.2.1 explores host based signature scanning techniques. Subsection 4.2.2 covers signature scanning in Intrusion Detection Systems. Finally, Subsection 4.2.3 summarizes the pros and cons of static signature scanning.

### 4.2.1 Host-Based Static Signature Scanning

Signature scanning is based on the assumption that a certain portion of a computer virus stays static. Thus, the presence of a certain signature in a file or memory indicates the presence of the virus the signature belongs to. In practice, the signature consists of a small sequence of bytes extracted from the virus. For instance, an imaginary signature could be the first 16 bytes of the virus code:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

The virus scanner would scan predefined locations of the computer, open each file in turn (or alternatively only some specific file types, such as executables to boost up scanning efficiency) and compare the contents of the file to each signature in the virus signature database. If a positive match is found, the scanner concludes that the file must be infected with a virus. A virus database can contain additional information such as multiple signatures or checksums of specific byte ranges for each virus to make the detection and especially disinfection more accurate.

In order to enable the virus scanner to detect minor variations of viruses, the signature matching engine can support methods for inexact matches. For instance, wildcards could be used to ignore certain bytes (bytes 2 and 11 in this case):

```
00 ?? 02 03 04 05 06 07 08 09 ?? 0B 0C 0D 0E 0F
```

Another similar method allows a specific number of mismatches (Ször, 2005, p. 432), without defining the exact location of the mismatching bytes. For instance, the following byte sequences could signal a virus, in a five-byte mismatch scheme (mismatching bytes indicated by a caret):

```
00 AA 02 03 04 05 06 01 08 BB 0A 0B 0C 0D 0E 0F
00 01 02 FF 04 05 06 09 08 09 0A FF 0C 0D 0E 0F
   ^     ^           ^     ^     ^
```

In order to improve efficiency, the virus scanner does not need to scan the entire file, but instead specific locations such as the beginning or the end of the file where viruses typically attach themselves. Alternatively a scanner might examine the code at the entry point of the application, where the actual execution starts, since viruses often place a jump instruction to the beginning of the code to move control to the virus body. However, some viruses try to attack anti-virus scanners by obscuring the entry point: the jump instruction to the virus code resides at some random location in the host.

A paper and accompanying source code (Bania, 2005) demonstrates the usage of a simple heuristic to detect and a signature to disinfect an entry point obscuring Win32.CTX.Phage virus, which infects Win32 PE files (see (Pietrek, 2002) for a description of the PE file format). First the scanner looks for jump instructions whose destination address points to the last section of the file (a simple heuristic; such jumps are suspicious since typically execution starts in the .text section of a PE file). The jump instruction itself is located with a simple string scan, looking for a single byte 0xE8, corresponding to the opcode of CALL instruction. The virus overwrites some bytes in the original file to its own entry point code and the replaced bytes are stored in the virus body so that the host can be successfully executed later. The disinfector described in the article uses a byte string to locate the original bytes in the virus body:

```
6A 00 6A 05 E8 05 00 00 00 ?? ?? ?? ?? ?? 50
```

The wildcards correspond to the original code and the disinfector simply copies the bytes to their original location.

In principle, signature scanning is a heuristic method, although generally the term is used to describe virus detection methods which do not rely on signatures, but instead inspect the behavior of the program to reveal suspicious activities. A virus signature typically consists of a small percentage of the bytes of the entire virus (Symantec, 1997). This is an important requirement, since it constraints the size of the virus database, which would grow unacceptably large in case entire viruses would be stored in it. Thus, signature scanning can produce false positives in some cases. For instance, a legitimate program could by chance contain the exact same sequence of bytes that constitute a virus signature; in this case, the virus scanner might give a false alarm to the user. Another drawback of signature scanning is that it can only detect existing viruses and minor

variations of them. More general heuristic methods are needed to discover new families of viruses. Due to its reliance on static code, signature scanning per se is also ineffective against mutating viruses (see Section 3.6): a properly implemented metamorphic virus might not contain a static sequence of bytes at all.

### 4.2.2 Signature Scanning in Intrusion Detection Systems

Signature scanning is also one of the principal building blocks of *Intrusion Detection Systems (IDS)*. An IDS monitors network traffic. Similarly to viruses that reside in files, worms and other exploits that are executed over the network can be detected by an IDS by means of signature scanning. An example of such a system is the popular open source intrusion detection/prevention system Snort (snort.org, 2019).

Snort works by catching network packets as they arrive to the machine on which Snort is running (the *sniffer* component). The packets are then sent to the *preprocessors*, which give Snort extended capabilities, such as support for TCP statefulness, protocol analysis, defragmentation of data transmissions before sending them to the detection engine and many other useful features (see (Caswell, Foster, Russell, Beale, & Posluns, 2003, pp. 197-264) for a thorough account on Snort preprocessors). The *detection engine* does the actual signature scanning, matching the packets against pre-defined rules. An alert/log entry to a log file or database is made in case the detection engine detects something based on the rules.

In case of Snort, the rules are manually created. However, there are some approaches in the related literature which might automate rule generation for IDSes. Other approaches exist in literature as well. For instance, Pillai, Eloff, and Venter (2004) describe a semi-automatic method which is built upon genetic algorithms. Adhering to the survival of the fittest principle, poor rules are eventually discarded by the system as well performing ones are preserved. The system produces new rules by means of mutation. The initial rules must be created by hand.

Real-time inspection of network traffic is a computationally expensive task for a high volume network. Some optimization techniques for existing Intrusion Detection Systems can be found in the literature, see for example (Rubin, Jha, & Miller, 2006).

### 4.2.3 Summary of Static Signature Scanning

Table 4.1 summarizes pros and cons of static signature scanning as a defense mechanism.

**Table 4.1:** Pros and cons of static signature scanning

| Pros | Cons |
|---|---|
| Able to detect specific viruses accurately. | Variations of a known virus might not be detected. |
| Detects a large number of viruses (the more comprehensive the signature database, the more viruses can be detected). | The signature database might grow large and it must be constantly updated by the anti-virus software vendor. |
| A large number of files can be scanned quickly. | Non-effective against mutating viruses. |

## 4.3 Heuristic Scanning

The previously described virus detection method was based on utilizing a priori information about the virus. Consequently, the method is only able to detect known viruses or slight modifications of them. Heuristic scanning takes another approach: instead of comparing exact byte sequences extracted from executable binaries against a virus database, heuristic scanning methods might inspect the structure of the target file, its internal logic and data in a more general sense (Symantec, 1997). Virus infected programs typically perform many unique activities compared to regular programs. After detecting several abnormal aspects from an executable, a heuristic analyzer might conclude that the file is probably infected with a virus of some sort; the more abnormalities are found, the more confident the final conclusion.

Heuristic scanning can be performed in two ways: static and dynamic. Static heuristics resembles the traditional signature based scanning described in Chapter 4.2.1. However, heuristic methods do not rely on signatures extracted from known viruses. Instead, the target code is statically analyzed, looking for other tell-tale signs of potential virus infection. The more suspicious signs are found, the greater the certainty that the file under inspection contains a virus. Similarly to traditional signature based scanning, static heuristic scanning can be targeted to those portions of files where viral code is most likely to be found. Some typical signs that might raise red flags in a heuristic scanner are (Ször, 2005) (Mori, Izumida, Sawada, & Inoue, 2006):

- A jump instruction at the beginning of the execution flow, redirecting the binary execution to the virus code.
- Modifying system settings (for example Windows registry settings for automatically launching programs at start-up time).
- Opening executable files or network connections.

- Scanning the file system or network.
- Writing itself to other files.
- Self-modifying code (for instance a decryptor component in a mutating virus).
- Incorrect values in the binary headers.

Similarly to traditional signature based scanning, code inspecting static heuristic analyzers might not be efficient against metamorphic viruses unless the anti-virus product is capable of unscrambling the mutated parts of the code. However, static analysis has the advantage of being able to cover the entire binary under inspection as opposed to dynamic analysis in which only those portions of the binary are inspected which happen to be executed.

Dynamic heuristic scanning inspects the behavior of the program as it runs. In practice, dynamic analysis is generally performed using an emulated or a virtualized environment. The emulated environment mimics the actual hardware and operating system by offering a software emulated CPU and providing a selected set of system APIs to the program that is being run. This way, the behavior of the program can be closely monitored by the anti-virus program.

An illustrative example of a dynamic virus analyzer is TTAnalyze (Bayer, Moser, Kruegel, & Kirda, 2006) (for implementation details, see (Bayer, Kruegel, & Kirda, 2006)). TTAnalyze is implemented using QEMU CPU emulator running a Windows XP operating system as the environment for the executable under inspection. A custom component is used inside the emulated environment for loading the executable into the virtual environment, as well as getting test results back from the virtual environment to the host system. TTAnalyze uses callback functions to hook interesting activities of the executable being run and composes a report back to the user. The report contains information about file system activity, Windows registry activity, starting and stopping of Windows services, starting and stopping of processes as well as network activity. The actual determination whether the executable in question is a virus or not must be determined by a human or another program, based on the report. An important observation of TTAnalyze and many other dynamic analysis tools is that they are able to handle mutating viruses. This is due to the fact that the analyzer pays no attention to the actual contents of the executable binary, but focuses entirely on monitoring the program's behavior. On the other hand, the virus might contain logic that allows it to execute the tell-tale behaviors only occasionally, for instance every second day, possibly evading detection. Furthermore, as it is difficult to develop an undetectable virtual / emulated environment (Ferrie, 2007), the virus might be able to detect that it is being

executed in an emulated / virtualized environment and refuse to execute. In these cases, the dynamic analyzer might miss the virus.

Table 4.2 summarizes pros and cons of heuristic scanning as a defense mechanism.

**Table 4.2:** Pros and cons of heuristic scanning

| Pros | Cons |
|---|---|
| Effective against mutating viruses. | Prone to false alarms. |
| Able to detect unknown viruses or variations of known viruses. | Dynamic scanning can be computationally expensive. |
| A large database of a priori information on viruses is not required. | Dynamic scanning environment can be detected by a virus and the virus can change its behavior to trick the scanner. |

## 4.4 Behavior Monitoring

Behavior monitoring is another distinct type of virus detection and defense. As the name implies, behavior monitoring techniques are used to inspect the run-time behavior of programs. Here a clear distinction is made to methods that inspect the actual contents of executable files in order to detect suspicious patterns – behavior monitoring in the sense described next happens only when software is being executed and the behavior monitor examines the activities the software exhibits in the system. It is noteworthy, that the technique bears close resemblance to emulated / virtualized heuristics described in Section 4.3. The biggest difference is that behavior monitoring happens in a real environment, not in an emulated / virtualized one. Several approaches to behavior monitoring have been proposed in the literature, including various methods that fall under the terms "behavior blocking" and "intrusion detection". These two domains often overlap and for the purpose of this Thesis the term "behavior monitoring" will cover both areas.

This section is organized as follows: Subsection 4.4.1 introduces different types of behavior monitors. Subsection 4.4.2 covers some possible attacks against behavior monitors. System call, file system and network monitoring techniques are examined in Subsections 4.4.3, 4.4.4 and 4.4.5, respectively. Subsection 4.4.6 summarizes the pros and cons of behavior monitoring methods.

### 4.4.1 Types of Behavior Monitors

Behavior monitoring comes in many flavors, depending on what kind of data constitutes as "behavior" for a given system. Naturally, different combinations are possible as well. In this Thesis, the run-time behavior of programs based on the following behavioral categories will be examined:

- Systems calls made
- File system events generated
- Network traffic

The list is by no means exhaustive, other possibilities exist as well. For instance, in a past research, Teng, Chen, and Lu (1990) describe a behavior monitoring system based on the shell command history of each user in a multi-user computer system: users executing commands that are considered anomalous for their user profile would raise a red flag. Another similar approach using neural networks to classify normal/abnormal user behavior based on command shell history is described by Debar, Becker, and Siboni (1992). However, the most behavior monitoring approaches found in literature fall in one of the three major categories described above.

The scale of the systems the behavior monitor watches may vary. The behavior monitor may be installed on a single computer, concentrating merely on the activities taking place on that particular system or in the particular subnet the computer resides in. On the other hand, there may be a handful of computers with a behavior monitor installed on each and additionally a data collecting entity installed on a yet separate system, to which the individual monitors send data for further processing. There may be behavior monitors in multiple network segments, sending data to a centralized monitor upwards the hierarchy. Furthermore, there may be behavior monitors looking at individual computers as well as network segments which report to multiple centralized data collecting entities. The list goes on. In principle, the behavior monitoring system may consist of multiple layers of monitors, forming a hierarchy and potentially having a human system administrator at the very top of the hierarchy, analyzing the results of the entire monitoring network. Section 4.4.5 explores these possibilities further.

In essence, behavior monitoring consists of defining a normal state of a system and subsequently monitoring the system in order to detect events that deviate from the norm. In a related study, Fawcett and Provost (1999) examined the generality of the problem. The aforementioned study gives behavior monitoring methods (or "activity monitoring"

as it is termed in the study) a useful high-level categorization, worthy of mentioning here:

- *Profiling* is a method in which the behavioral model of a system is built of normal activity. In this method, the interesting events are those that deviate from the normal.
- *Discrimination* is a method in which the behavioral model of a system is built of anomalous activity. The interesting events are then detected by comparing the system behavior to the model.

Being already difficult, the task is further complicated by the fact that many computer systems are dynamic in nature, i.e. the normal state of the system tends to vary in time (Stolfo, Hershkop, Bui, Ferster, & Wang, 2005). Although in some cases behavior blocking can be based on static rules, a general purpose behavior monitor is required to be adaptable to changing environments.

### 4.4.2 Attacks Against Behavior Monitors

A powerful attack against many kinds of behavior monitoring techniques is the so-called *mimicry attack* (for an in-depth study, see Wagner and Soto (2002)). Mimicry attacks attempt to slip under the radar of the behavior monitor by behaving in a way that is considered normal by the monitor, i.e. mimicking normal behavior. As described in the study, it is safe to assume that a dedicated attacker knows how a specific behavior monitor works, i.e. the attacker is free to study it in an arbitrary depth. For instance, the source code for behavior monitors published under open source licenses are available for studying. It is also not far fetched to assume that the source code for commercial products might find their way into the public. Also, reverse engineering and trial-and-error techniques might reveal enough information for the attacker to abuse the behavior monitor program. In the aforementioned study it is then described how an attacker can utilize the information about how a particular behavior monitor classifies normal and anomalous behavior. Some examples mentioned in the study are:

- Avoiding behavior that is observed by the monitor. For example causing damage to a system without executing any system calls, if the behavior monitor merely monitors system calls.

- Waiting until the malicious behavior will go undetected by the behavior monitor. For example, if an attacker gains control of an application, the application may be allowed to execute itself normally until the time would be just right to execute malicious activities. The time window when this might be possible would have to be pre-studied by the attacker.

- Abuse the limitations of the behavior monitor. As pointed out in the study, many system call behavior monitors ignore system call parameters and return values. This can be utilized to perform malicious activities on the system without alarming the behavior monitor.

- Fooling the behavior monitor by masquerading the attack with irrelevant *no-ops*. The idea behind this attack is that many behavior monitors function by building a database of behaviors considered anomalous (or normal, depending on the type of the behavior monitor) and comparing the run-time behavior of the system against the database. For example, a match for a sequence of system calls:

  ```
  <A, B, C, D, E>
  ```

  might be found in the pool of anomalous sequences (or might not be found in the pool of normal sequences), but a sequence of:

  ```
  <A, B, C, D, X, E>
  ```

  where X is some semantically irrelevant system call might be considered normal. Variations of this scheme are replacing system calls with equivalent alternatives or altering the order of the system calls in the sequence.

### 4.4.3 System Call Monitoring

System calls are an interface to the operating system, which can be used by a user space application to request services from the OS. For instance, common system calls that occur in UNIX systems are

```
read(), write(), open(), close(), select(), exec(), fork()
```

and many others. Due to the ubiquity of these calls in applications, system call patterns are a natural target for observation, both on an application level and on a system level. The following examples demonstrate some possible approaches to monitoring system calls on an application level.

DOME (Rabek, Khazan, Lewandowski, & Cunningham, 2003) takes an explicit approach to behavior blocking. The software operates in two phases:

1. Every executable in the system is inspected by DOME. During the inspection, it creates a ”system call behavior signature” for the given executable: it extracts the virtual addresses and names of the Win32 API functions the executable calls during its lifetime.

2. The software starts monitoring the behavior of the executables in the system. Every time a Win32 API call is made, DOME intercepts the API call, checks whether the call came from a valid address (comparing to the ”signature” produced in step 1) and raises a red flag if the call does not match the system call behavior of the executable.

In other words, DOME first determines the expected behavior of each executable, then monitors that the behavior never deviates from the expected. This means that the principle behind DOME is an effective protection against code injection (new code is injected to the memory space of a process during its execution, causing a modified system call behavior), mutating code (encrypted portions of the code are decrypted during execution to reveal viral, and thus new, behavior) and various kinds of code obfuscation attacks (API calls are revealed during run-time that were not detected in phase 1 of DOME execution).

In another study (Summerville, Skormin, Volynkin, & Moronski, 2005), the focus of the behavior monitor is in detecting self-replication. First, it is assumed that self-replication is always an anomalous event, i.e. legitimate programs are assumed never to reproduce. The foundation of the study is in the immune mechanisms of biological entities. The authors define the *Gene of Self-Replication* (GSR) as: ”The GSR is viewed as a specific sequence of commands passed to the computer operating system by certain program code that causes this code to replicate itself through the system or multiple systems”. The GSR consists of multiple building blocks and each building block in turn consists of multiple system calls. The system calls inside of a building block are tied together by means of input/output arguments. For instance, the block dubbed Host Search Block in the study, consists of two system calls: NtOpenFile and NtQueryDirectoryFile. The system calls are tied together by the output argument of NtOpenFile: the handle to the opened file is given as an input argument to NtQueryDirectoryFile. In the same manner, other building blocks are constructed. As the detection algorithm runs, it recognizes building blocks that are a part of the GSR. It is noteworthy to emphasize that the system is indeed not interested in individual system calls, such as the aforementioned two,

unless they form a block by being tied up in a specific manner by their input/output arguments. Some individual building blocks may be found in legitimate programs and that alone will not raise an alarm. An alarm is raised only after the system has detected enough building blocks to conclude that the system call sequence it has observed has resulted in self-replication in the system.

In principle quite similar, yet slightly differently implemented behavior monitoring system is described by Hollebeek and Waltzman (2004). The system is based on the notion of suspicion. Similarly to GSR, individual events are rarely suspicious enough to cause an alarm, unless the attack is very explicit. The system is built upon a set of rules (designed and implemented by human experts), which determine the types of events that are considered suspicious. Suspiciousness is raised for example if an event has a potentially malicious explanation (it might have a non-malicious explanation as well), if an event is linked to other suspicious events or if an entity, for example a process, is causing suspicious events, the entity itself becomes suspicious (more characterization of suspicious events in Hollebeek and Waltzman (2004, p. 4)). The system builds a suspicion graph based on the events it finds suspicious, linking them together and thus producing a model of the suspicious behavior, which can be either interpreted by a human being or processed further programmatically.

### 4.4.4 File System Monitoring

The file system is a frequently accessed part of a computer system by many kinds of software and thus a natural place to observe. Malware often exhibits behavior quite different compared to ”normal” applications. For instance, writing to executable files can be considered suspicious – an act performed by many viruses as they infect their targets.

Many approaches to file system monitoring can be found in the literature. As an example, *The File Wrapper Anomaly Detector* (FWRAP) (Stolfo et al., 2005) builds a database of file access records. Each record contains some important information about the file access, such as *User ID* (UID) of the process accessing the file, *Working Directory* (WD), three previously accessed files (PRE-FILE), FREQUENCY, etc. (for the exact record description see the related study (Stolfo et al., 2005)). Each file access generates a record to the database, creating a model of the host system's file access behavior. First, the system must be trained in order to produce a model of behavior that is regarded as normal. In the study, this was achieved by running the FWRAP sensors on a test machine during normal usage (reading e-mails, basic system administration, etc.). After that, the model was used to monitor the system, with users deliberately attempting to

do malicious activities. The study results reveal the well known: better training of the models lead to higher detection rates of malicious activities as well as to lower false positive rates.

### 4.4.5 Network Monitoring

*Network Intrusion Detection Systems (NIDS)* enable security personnel to monitor network level activities. As the name suggests, the main function of a NIDS is to detect suspicious activities based on pre-defined or learned rules and to log and report these activities to a monitoring entity, be it a system administrator or another computer program filtering the data further. Malicious activities in a network often distinguish from behavior considered normal (*ping sweeps*, *SYN floods*, etc.). Many automated and manual reconnaissance and attack attempts leave a distinct fingerprint, which can be used to create the aforementioned rules for the NIDS to react on.

Intrusion detection systems that monitor network traffic can be roughly divided into three groups: network level systems, host level systems and various kinds of hierarchical systems. The following paragraphs give a brief introduction to each of these systems.

***Network Level IDS***

A NIDS can be deployed to a networked machine, which listens to all traffic on the subnet it resides on, i.e. the network interface of that particular machine operates in promiscuous mode capturing all network packets (Caswell et al., 2003). To illustrate this, a small corporate network is depicted in Figure 4.1. The network consists of multiple workstations (at the bottom of the picture), two server machines, a firewall and a router. In the depicted scheme, the workstations reside in a separate network, separated by the router. NIDSes have been employed to monitor network behavior in two different subnets, monitoring hosts available to the public (the two servers) and a farm of internal work stations. It is also important to point out, that in this scheme each NIDS is independent and unaware of one another, although they reside in the same corporate network – there is no central entity gathering collective information from the NIDS instances.

A NIDS is a useful tool for detecting worms. Worms, by definition, use networks as their operating medium. In addition to the actual attack traffic, the operation of the worm most likely also consists of some sort of active reconnaissance or discovery of potential targets, for instance by pinging random IP addresses.

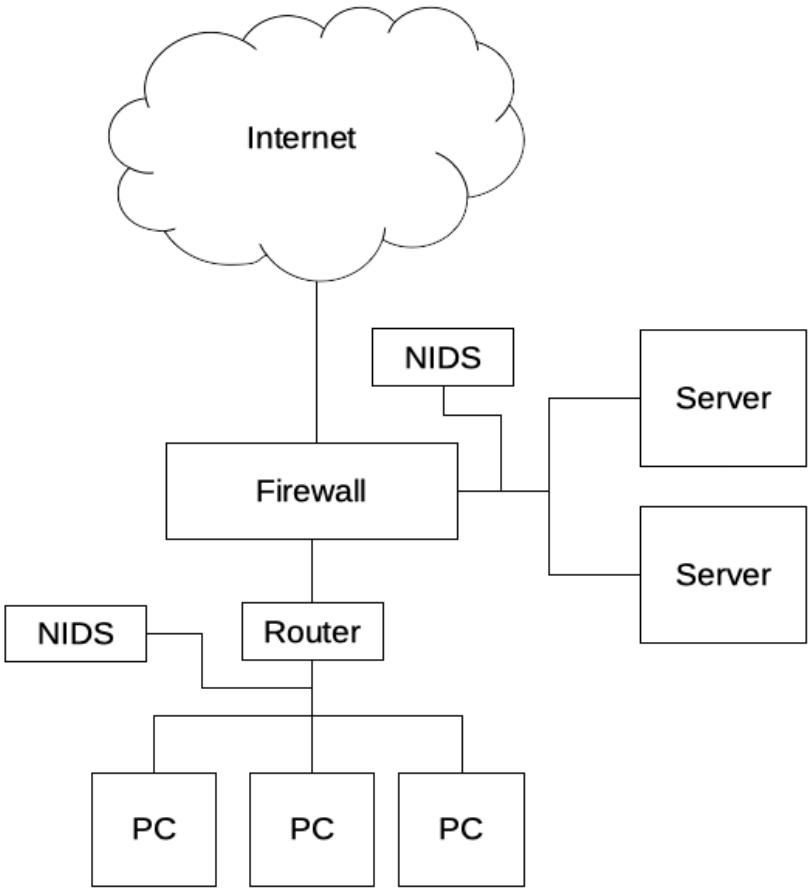



**Figure 4.1:** Several network level intrusion detection systems monitoring subnets of a corporate network (adapted from Caswell, Foster, Russell, Beale, and Posluns (2003)).

A NIDS can be signature based as described in Section 4.2.2 or it can be based on other behavioral aspects of the network. Bielecki and Hajto (2004) describe an implementation of a neural network based network behavior monitor. The motivation behind the study is to have an efficient network intrusion detector that runs in a router, taking appropriate action on packets (for instance dropping them) that are thought to be a part of malicious activities. Running a static signature based virus scanner on all packets that come through would be inefficient. To improve efficiency, neural networks are trained to only allow through normal traffic. Traffic for a computer which resides in a subnet behind the router is considered normal if the traffic volume and the number of contacted hosts fall withing the range of values that the neural network was trained with. During training, a traditional virus scanner is used as a supervisor during the learning process. After training, the virus scanner is no longer needed and the neural network classifies the inputs as viral or non-viral autonomously.

***Host Level IDS***

Figure 4.2 depicts a similar corporate network architecture with *Host level Intrusion*

*Detection Systems* (HIDS) incorporated into its hosts (Caswell et al., 2003). Contrary to NIDS, HIDS monitors events only on a single host.

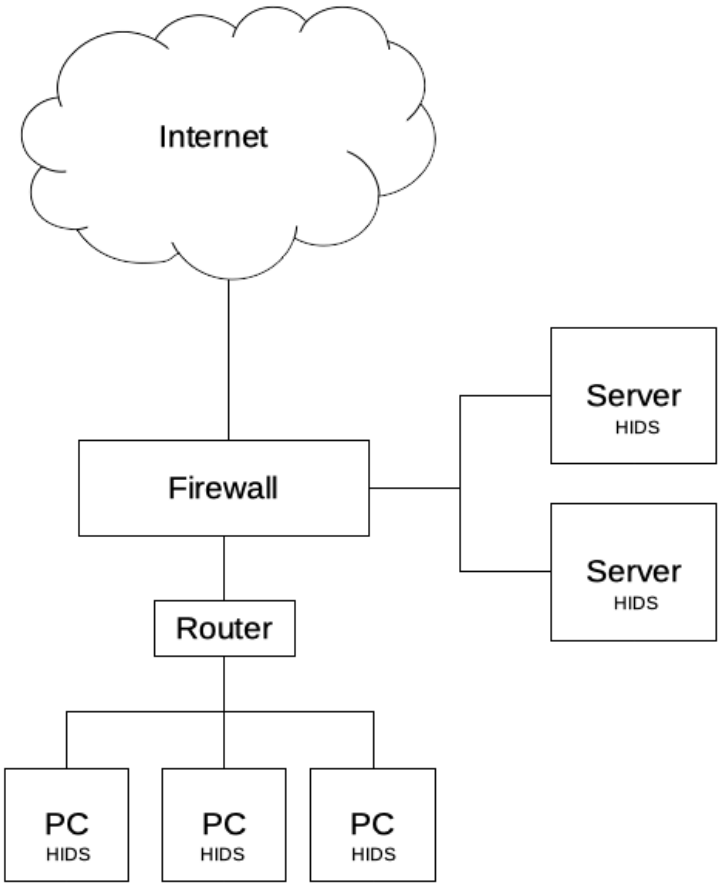



**Figure 4.2:** Host level intrusion detection systems deployed in a corporate network (adapted from Caswell, Foster, Russell, Beale, and Posluns (2003)).

An intrusion detection system dedicated and installed on a single host can monitor that particular system more thoroughly than a remote IDS. Its monitoring capabilities are not restricted to network traffic, but can extent to other areas of the system such as individual files and system calls executed by running processes, thus allowing a more fine-grained control over the system. Without proper automation, a large number of individual HIDS instances can be a burden to manage and maintain.

More and more behavior monitoring nodes (host based or network based) can be combined together to form a hierarchical scheme, where data is collected in the low levels of the hierarchy and sent upwards for further processing. These hierarchical systems are described next.

***Hierarchical Intrusion Detection Systems***

Hierarchical Intrusion Detection Systems consist of an arbitrary number of data collecting entities (the lowest level), for instance NIDS and/or HIDS instances, and additionally one or more layers of data processing nodes. A Hierarchical IDS steps up to a higher vantage point: the point of interest is not in individual nodes, but in the behavior of groups of nodes, not necessarily residing in the same network. One or more higher level nodes collects data from the individual lower level nodes for further analysis, possibly discarding irrelevant data and delivering the rest further. In this Thesis, systems generally called *Distributed Intrusion Detection Systems* (DIDS) (Einwechter, 2002) fall in the Hierarchical IDS category, among some other similar approaches.

A Hierarchical IDS might enable the incident analysts of an organization to detect machines infected by worms or other malware and disable their spreading in the network (Einwechter, 2002). Also, if the malware is launched from within the network (for instance, by a malicious employee), the point of origin might be detected and the perpetrator found. Due to their nature, the data collected by a Hierarchical IDS can be aggregated and used to detect and analyze coordinated attacks occurring in different networks, for instance networks of a corporation in separate geographical locations.

Figure 4.3 illustrates a simple Hierarchical IDS. The picture depicts two separate networks which are geographically distributed into different locations (e.g. different countries). The networks could be a part of some multinational corporation or organization. Each of the networks is monitored by an autonomous agent. The agent has a dual role: it both functions as an intrusion detection system for that particular network and also as a communicator to the Control Center, seen at the bottom of the picture. Advanced agents may be able to learn to function better in their environment over time and communicate with other agents when appropriate. In Figure 4.3, software agents are the basic components of the intrusion detection system, collecting and aggregating data for security personnel employed at the Control Center. The Control Center then has a broad view of the activities of the entire corporation.

Many Hierarchical Intrusion Detection Systems can be found in the literature, with differing architectures. A multilayer, distributed, advanced agent based system is described by Eugene H Spafford and Zamboni (2000), called *Autonomous Agents for Intrusion Detection* (AAFID). The system consists of agents at the lowest level, monitoring various aspects of target systems, transceivers as a middle layer (controlling a group of agents) and monitors, which control transceivers and other monitors.

Another distributed, agent based approach with a peer-to-peer architecture is described by Ramachandran and Hart (2004). The basic idea of the system in this study revolves around the idea of a neighborhood watch. Each node (a single computer) stores

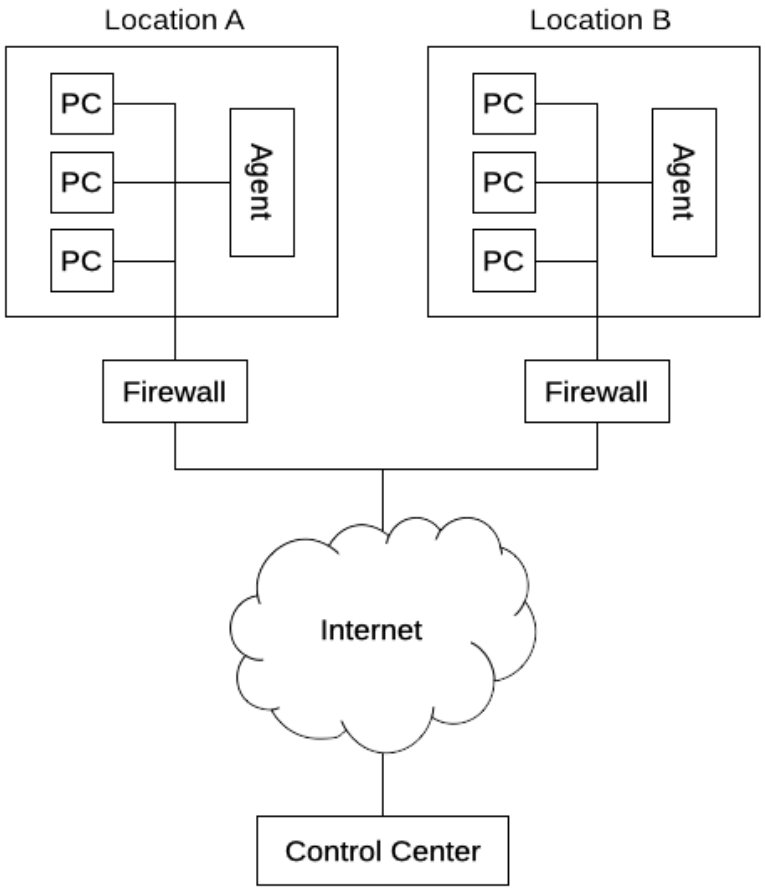



**Figure 4.3:** A hierarchical intrusion detection system (adapted from Einwechter (2002)).

some security related information about its neighbors, such as checksums of critical files, normal CPU activity, configuration files, etc. This information can be anything considered relevant in a given environment. Each node monitors its neighbors and in case something suspicious is detected in a particular neighbor, a voting is organized between the neighbors of the suspect node. If a majority of the voters agree that the suspicious node has indeed been compromised, defensive actions can be taken.

More approaches to Hierarchical Intrusion Detection Systems with varying architectures can be found in the literature, for example by Zou et al. (2005), Jin, Yang, Sun, Tu, and Han (2005) and in many other studies.

### 4.4.6 Summary of Behavior Monitoring

Table 4.3 summarizes pros and cons of behavior monitoring as a defense mechanism.

**Table 4.3:** Pros and cons of behavior monitoring

| Pros | Cons |
|---|---|
| Not based on static signature checking and thus able to detect unknown threats. | Analyzing system calls, file system events and network traffic for anomalous behavior incurs computational overhead. |
| Able to detect and block malicious activities proactively, before any harm is done. | Training (and retraining) models can be time consuming. |
| Able to detect coordinated attacks, when scaled to cover a large number of hosts in a distributed manner. | Defense mechanisms relying on the comparison of normal behavior to abnormal behavior are open to mimicry attacks. |
| An advanced behavior monitor learns to adapt to changing environments automatically. | Poorly trained models may lead to high false positive alarm rates. |

## 4.5 Firewalls

Firewalls are software or hardware devices, which are used between networks having different security policies to secure a trusted network from an untrusted one. For example, a typical scenario is to separate a network internal to a company or home from the Internet with a software or hardware firewall residing between the networks. There are two major types of firewalls: packet filters and application level proxies. Packet filters come in two flavors: stateless and stateful. Stateless packet filters merely monitor individual network packets as they come. Stateful packet filters on the other hand are able to store state information of connections (for example a TCP/IP stream). Packet filters operate by inspecting incoming and outgoing packets for their source/destination IP address and port, used protocol and other information in the packet headers. The actual application level payload is not inspected by packet filters. Application proxies work on the application layer, acting as an intermediary in application level network connections and thus also able to inspect the payload of packets that pass through them. Firewalls provide security on many levels (Nazario, 2003):

- Service protection: Vulnerable services running inside a network may be protected by firewall rules. Also vulnerable services outside the network may be protected, by prohibiting traffic specific to those services from leaving the network.
- Access control: Firewalls can be used to grant/deny access to sensitive destinations.
- Privacy: Services running inside the network may be concealed by a firewall, thus hiding their presence.

- Logging: Firewall logs may be used for incident analysis after suspicious activities have occurred.

Firewalls are useful tools for mitigating the spread of worms in many ways. By restricting the inbound traffic (by IP address, port or protocol), it might be possible to prevent worms from infiltrating a network. For example, blocking the UDP port 1434 (if MS SQL Server 2000 happens to be running behind that port) prevents the propagation of the SQL Slammer (F-Secure, 2019i) worm. In large organizations, firewalls should also be placed inside the organization network, to break down the intranet into separate subnets and thus preventing the propagation of worms that manage to penetrate the perimeter firewalls (Zou, Towsley, & Gong, 2004). Internal firewalls also protect the intranet from attacks originating from inside the network.

Although useful as a defense mechanism in many respects, firewalls are an insufficient defense mechanism alone. Worms may pass through a firewall directly using allowed, legitimate rules. For example the aforementioned worm, SQL Slammer will pass through a firewall if it is configured to allow UDP traffic on port 1434. Worms and viruses may also penetrate the network by other means, for instance via e-mail, compromising the subnet where the infected e-mail is opened or by means of a human being physically carrying an infected storage media inside an organization and allowing the worm or virus to spread from there.

Table 4.4 summarizes pros and cons of firewalls as a defense mechanism.

**Table 4.4:** Pros and cons of firewalls

| Pros | Cons |
|---|---|
| Cost-effective, easy to deploy and maintain. Many easy-to-use consumer level solutions exist in the market. | Insufficient defense mechanism alone. |
| Provides security on many levels. | Firewall configuration must be carefully done, otherwise the level of protection is diminished. |

## 4.6 The Social Dimension

Designing a functioning security policy for an organization of any size is not merely a technical endeavor. In many cases, there is a human factor involved. No matter how carefully crafted the security design technically is, if the human factor is ignored, the

value of the entire security design will be considerably lower. An illustrative example is provided in a news article dated June 2006 (Stasiukonis, 2006). In the article, a security audit performed on a credit union with a special emphasis on the social dimension is described. The auditing party crafts an ”attack” on the client company, by creating a custom trojan that is able to collect sensitive information from the computer it is executed on. The attack is carried out by copying the trojan, along with some other seemingly unsuspicious files on a number of USB drives, and distributing the drives to places that the employees of the client company frequently spend time in. Relying on human curiousness, the security auditors plotted to have the USB drives inserted by the employees themselves into the network of the client company. Indeed, the trojan was activated, sensitive information collected and then sent to the security auditors by e-mail. According to the news article, a majority of the USB drives were found by the employees, who then plugged them into the computers of the client company, exactly as planned.

A functioning security design for an organization, with regards to the human factor, should address at least the following issues:

- Keeping the software used by the company up-to-date (applying necessary patches and upgrades periodically): This requires that somebody in the organization is responsible for monitoring the relevant news channels and taking appropriate actions as critical vulnerabilities become known.
- E-mail filtering and usage policy: Incoming and outgoing e-mails should be pre-filtered for malicious content. In addition to this, a strict policy should be enforced, preventing the employees from opening e-mails and e-mail attachments from unknown origins. Many viruses and worms are known to spread via e-mail.
- The storage on portable computers (laptops, mobile phones and tablets) and portable media should always be encrypted by the user or by somebody in the organization: For example, an employee might forget a company laptop with an unencrypted hard disk containing sensitive data at a public location. Obviously, this could compromise the data on the hard disk. It could also grant access to the computer to a potential attacker for planting malware (backdoors, key loggers, worms, viruses, etc.), which could be activated when the laptop finally finds its way back to the owner, possibly wreaking more havoc inside the organization.
- A password policy for company systems: If the employees are able to choose their own passwords, some basic rules about choosing strong passwords should be taught. Without proper education, wireless networks might also be set-up with

either no encryption, with poorly chosen passwords or with an insecure encryption protocol (such as WEP (Bittau, Handley, & Lackey, 2006)), potentially allowing access to the intranet by an attacker.

- The employees should be wary of phishing attacks (online, phone, e-mail or in-person) and never reveal passwords or other data that might compromise the company network: If an attacker manages to penetrate one computer inside the company network, malware could be planted and further damage caused.
- All of the above mentioned issues should be reviewed on a regular basis on the organizational level: Ideally, the security culture in an organization would be self-learning. For example, if an employee identifies a potential security risk, a process should be in place as to how the newly found security issue will be addressed.

Furthermore, when designing the security policies for an organization, it should be noted that security methods that are too cumbersome will hinder work progress in the organization as well as potentially increase job dissatisfaction of the employees. It might be that eventually these kind of security policies would start to be ignored. Thus, it is important to find a suitable balance between security and efficient functioning of the organization.

Table 4.5 summarizes pros and cons of social dimension as a defense mechanism.

**Table 4.5:** Pros and cons of social dimension

| Pros | Cons |
|---|---|
| Well taught human beings are in some cases more efficient at identifying security problems than any software/hardware method for the time being. | Humans are not machines, so security policies (especially poorly designed ones) might be ignored. |
| Humans learn to adapt to a changing security environment. | Humans are prone to social engineering. |

## 4.7 Summary

This chapter took a wide-angle view on the landscape of defense mechanisms against computer viruses. The central conclusion that can be drawn from the literature is that no single defense mechanism is enough to cover all the possible attack vectors potentially

exploitable by computer viruses. A working defense strategy is a combination of several technical solutions together with an appropriate emphasis on the social dimension.

# 5. Modern Computer Viruses

Chapters 2-4 of this Thesis have examined computer virus evolution, deconstructed their operation as well as provided a look into various defense mechanisms largely from the perspective of personal computers. However, computer viruses are agnostic with regards to the medium they operate on – so long as they are able to execute and spread, they are able to function. In the modern world, computing capabilities and connectivity are built into everything from Bluetooth connected toothbrushes to robotic lawnmowers and more smart devices are entering the market every year. The purpose of this chapter is to examine some of these devices in more detail to gain an understanding on if and how computer viruses could proliferate on them.

In particular, two different types of devices were chosen for a closer examination for this chapter: mobile phones and cars. Mobile phones have been targeted by malware for many years and their attack vectors as well as effective countermeasures are relatively well studied. Hence, they were chosen as the first subject of study. On the contrary, the automotive field is still in its relative infancy in terms of the sophistication of built-in, connected computing devices and has not yet seen widespread malware attacks. However, in the recent years the field has been rapidly changing with increasing digitalization and automation. It is well known in the related research literature that a potential for serious security and safety problems exists. Hence, cars were chosen as the second subject.

Other types of systems, such as Internet of Things (IoT) as well as medical systems, were also considered and would have been interesting subjects of study as both have been targeted by various malware in the recent years. Indeed, if technically feasible and a motivation (monetary or otherwise) for attacking certain types of systems exists, it can be conjectured that sooner or later said systems will come under attack. For IoT and medical systems, this has already happened.

The study subjects, mobile phones and cars, have been broken down into Sections 5.1 and 5.2 with an identical subsection structure, as follows:

- Subsections 5.1.1 and 5.2.1 provide essential background information on mobile phones and cars, respectively
- Subsections 5.1.2 and 5.2.2 cover potential attack vectors on mobile phones and cars, respectively
- Subsections 5.1.3 and 5.2.3 discuss various countermeasures to defend against attacks on mobile phones and cars, respectively
- Subsections 5.1.4 and 5.2.4 examine inherent factors that might mitigate attacks on mobile phones and cars, respectively
- Subsections 5.1.5 and 5.2.5 on the other hand examine inherent factors that might exacerbate attacks on mobile phones and cars, respectively

## 5.1 Viruses in Mobile Phones

Mobile phones have been around for a long time. Since their inception decades ago, wireless networking technologies such as Bluetooth, Wi-Fi, NFC and 4G have become ubiquitous and many smartphones today support them all. This gives the present-day mobile phone a high degree of connectivity to the world around it. New wireless technologies are introduced to mobile phones on a regular basis, such as the Ultra-Wide Band (UWB) chip in iPhone 11 enabling precise location tracking of objects nearby (Wikipedia, 2019c). Mobile phones have converged into fully-featured computers, built on top of advanced software platforms.

Compared to desktop computers and cars (see Section 5.2), mobile phones have some unique characteristics which make them interesting targets for virus writers. The mobile phone is likely the most personal of all devices we use on a daily basis: it stores our private contacts, messages, e-mails, photos and videos. It is used for making purchases online as well as performing banking transactions. It knows our location, and is equipped with a myriad of sensors that, when compromised, may be used against us. Mobile phones are universally used as a pocket-sized computer to get things done on the road by traveling business people. Hence, not only does it store our personal data, it often stores valuable company data as well.

Due to these factors, mobile phones have become an attractive platform for virus writers in the recent years. Although the first mobile phone viruses were seen as early as 2004 (see Section 2.10), the number of attacks has started raising more sharply around 2010. This timeframe roughly coincides with the widespread adoption of app stores (including

3rd party app stores) and downloadable apps. Indeed, app stores remain one of the key attack vectors to this day.

Many cybersecurity companies have reacted to the growing number of viruses by releasing anti-virus software specifically targeted to mobile devices. In theory, a mobile phone allows the usage of the same or similar defense methods as those described in Chapter 4. In practice, many of the defense mechanisms have been shown to be ineffective as the virus landscape is constantly evolving or unpractical due to limitations of the underlying hardware (Suarez-Tangil, Tapiador, Peris-Lopez, & Ribagorda, 2013). Mobile operating system vendors such as Google and Apple have naturally responded by adding new security measures on a regular basis. Despite countermeasures, mobile phones are rife with viruses and other types of malware, causing both personal grief and measurable financial losses. This section will delve deeper into the field of mobile phone viruses and attempt to provide an overview of the current situation.

This section is organized as follows: Subsection 5.1.1 provides essential background information on mobile phones. Based on that understanding, Subsection 5.1.2 explores potential attack vectors. Subsection 5.1.3 discusses some approaches on countermeasures. Finally, some mitigating as well as exacerbating factors regarding mobile phone virus attacks are considered in Subsections 5.1.4 and 5.1.5, respectively.

### 5.1.1 Background

A mobile phone has a complex internal architecture and it is connected to a complex external infrastructure. At the time of writing this Thesis, the prevailing mobile phone operating systems are Apple iOS and Google Android. Both vendors offer a host of services that form a part of the external infrastructure, such as application stores, development and distribution tools, push notification services, streaming services, productivity tools and more. Both vendors also offer a suite of enterprise features, which allow the development and distribution of internal applications, as well as tools for device management. The other part of the external infrastructure is offered by mobile operators as well as third parties offering services of interest to the user. The mobile phone connects to these services through the public Internet, often through an always-on Wi-Fi or cellular connection.

The internal architecture of iOS and Android differs in many respects. Although iOS is built on an open source kernel (Darwin), the upper layers are built largely out of closed source components. Android on the other hand is built on the open source Linux kernel and the main platform is available in source code form through the Android Open

Source Project (AOSP). The proprietary Google Play Services are required to connect to the Play Store and other services provided by Google, however.

Apple and Google operate with significantly different business models in the mobile phone market. Apple sells complete products, building both the software and the hardware in-house. Google on the other hand offers AOSP to device manufacturers, who then customize the platform with their own applications and look and feel. Many device manufacturers also adopt the proprietary Google Play Services. The difference in the business model has implications on security, as will be seen through Subsections 5.1.2-5.1.5.

### 5.1.2 Potential Attack Vectors

Many possible attack vectors to a mobile phone exist. In their study on mobile malware, Suarez-Tangil et al. (2013) determined the following possible attack vectors:

1. App store
2. Applications (particularly web browser)
3. SMS/MMS
4. Wireless network (Bluetooth, Wi-Fi, Cellular, etc.)
5. USB

App stores, especially Play Store by Google as well as various 3rd party stores for Android are a major attack vector. An attack can take many forms. In its most basic form, infected applications are distributed through the official Play Store, despite built-in security measures (McAfee, 2018). An infected application may appear as a legitimate application offering ostensibly useful features. In a *repackaging attack* a known application is implanted with malicious code and then redistributed in a 3rd party store (Suarez-Tangil et al., 2013). It is worth noting for the purposes of this Thesis, that malware distributed through the app store does not necessarily meet the definition of a virus, as they may not self-replicate but spread through the app store instead. Such malware attacks can be more accurately categorized as trojans.

The web browser has been traditionally associated with cyberattacks of many kinds. A particularly insidious attack vector has been demonstrated by Frigo, Giuffrida, Bos, and Razavi (2018): the *GLitch* attack. The researchers show that the web browser of an

off-the-shelf mobile phone can be compromised by an elaborate microarchitectural attack using the GPU through the WebGL API. An attacker can take control of a vulnerable device when a user visits a malicious website, circumventing almost all known defense mechanisms.

*Multimedia Messaging Service* (MMS) and *Short Messaging Service (SMS)* provide another attack vector for mobile viruses. Both messaging types utilize the cellular network, meaning that the distance between the attacker and the victim is irrelevant.

The data payload in an SMS message is rather small for viruses (160 bytes), which makes it non-ideal for this purpose (Bose & Shin, 2006). However, SMSes can be used in malicious ways by a virus. Some examples of malware abusing the SMS facilities have been seen in the past, for example Viver trojan (F-Secure, 2019e). Viver was distributed on a file-sharing site, masqueraded as a harmless utility program. After installing it on an S60 second edition or earlier phone, Viver starts sending SMSes to a premium-rate number which was rented by the malware author. The victim was billed and the author profited. A more recent and fully self-replicating virus using SMS to spread itself, called Selfmite, is presented by Heartfield and Loukas (2015). The virus uses a social engineering attack to lure the user to open a link sent by SMS, which in turn leads the user to download an infected application. The application further spreads by sending itself to the first 20 contacts in the user's phone book.

MMS on the other hand allows sending more data per message. The primary purpose of multimedia messages is to enable the user to send multimedia content (audio, images and video) as well as text content and attachments to their contacts. Viruses and other malware utilizing the MMS have been seen in the past. An example of a MMS based worm that gained some news coverage at the time of appearance is Commwarrior (F-Secure, 2019g). Commwarrior targets Nokia's S60 platform and uses Bluetooth as well as MMS as a propagation method. Commwarrior uses the contact book of the infected device to find new victims – it sends itself in an MMS message to all contacts in the contact book. Furthermore, it attempts to send itself via Bluetooth to devices it happens to find. In addition to causing harm to end users, high volume SMS/MMS based attacks might cause denial or degradation of service for the service provider, as a related study suggests (Fleizach, Liljenstam, Johansson, Voelker, & Mehes, 2007). An example of a more recent MMS based attack is based on exploiting the Stagefright vulnerability (Wikipedia, 2019a). It allows an attacker to send specially crafted MMS messages to victims and take full control of their device – no interaction on behalf of the user is needed. The attacker only needs the phone number of the victim.

USB based attack vectors are rare, but some have been seen in the wild. An example is

the Android/Gepew trojan (F-Secure, 2019d), which infects susceptible Android phones through USB from a connected PC. It then attempts to replace legitimate banking apps with trojanized versions.

This section only scratched the surface of possible attacks on mobile phones. As is evident, mobile phones have been widely targeted with a huge variety of different attacks and numbers are rising every year. A point worth stressing here is that the most prevalent spreading mechanism today is in fact not self-replication. Instead, malware spreads mostly through app stores and applications.

### 5.1.3 Countermeasures

The leading mobile phone operating system vendors, Google and Apple, take quite a different approach on securing their systems. Google has traditionally taken a relatively open stance, giving users and developers more freedoms. Apple on the other hand is known for a tightly controlled device as well as application distribution process. It appears that these different approaches also lead to different outcomes in terms of the prevalence of malware, as Google's Play Store has been plagued with more malware than Apple's App Store. It stands to reason then, that Apple's so called walled-garden approach leads to improved security (F-Secure, 2019c). Google has over the years improved their application review process and recently also managed to clean up the Play Store of malicious applications, at least to some extent.

In addition to app store reviews, both platforms require applications to be signed cryptographically. This helps to ensure the authenticity of applications. It is worth noting that on Android it is trivial to side-load applications, bypassing possible Play Store protection. Indeed, this is an attack vector that is exploited by malware, as described in Section 5.1.2. On iOS side-loading is not possible, however jailbreaking procedures that enable it do exist.

Both iOS and Android employ a host of platform level protections. These include secure boot chain, hardware enabled key protection, full-disk encryption, application sandboxing and application permission controls. In addition, users are provided with user friendly ways to unlock their devices using facial recognition and fingerprint scanners to encourage the use of full-disk encryption and to discourage the use of easy-to-guess PIN codes. This multilayered security model on both platforms enables defense in depth, making it harder for attackers to infiltrate the device as well as curtailing the impact of a potential breach. However, no security measure is perfect and built-in defenses can in some cases be weakened either accidentally or knowingly by users.

For example, the application permission model on Android requires the user to grant a blanket permission to use all of the services an application requests at the time of installation (Suarez-Tangil et al., 2013). However, the user may not know or care about the eventual impact of granting these permissions, especially since applications may not explain why the permissions are needed. For an in-depth description of the security models of both platforms, see (Apple, 2019) and (Google, 2018).

Active defenses similar to those described in Chapter 4 are also in use. Many vendors offer mobile security products with features such as signature based scanning. The big difference to desktop and server environments is, of course, the hardware. Mobile hardware is generally equipped with a less efficient CPU, less memory and less storage space in comparison to desktop and server environments. As typical defense methods such as signature scanning, emulation and many heuristic scanning methods tend to be computationally expensive, using the same virus defense techniques on a mobile device may not be feasible without a major impact on battery life and usability.

Some optimization techniques for anti-virus programs for mobile devices can be found in the literature. For example, in the study by Venugopal (2006), the traditional signature scanning method is improved to reduce the memory footprint to make the method more suitable for mobile devices. The method uses a double hashing mechanism to speed up the look-up of signatures during scanning, as well as to reduce the amount of data that needs to be stored in memory at run-time. In a study by Polakis, Diamantaris, Petsas, Maggi, and Ioannidis (2015) the researchers found major differences in terms of battery life between different anti-virus vendors depending on the detection technique used. Interestingly, the visual design of the application also has a major impact on battery life, especially when the virus scan is performed by the scanner application running on the foreground. Some simple techniques to preserve battery life were proposed, for example terminating the scan of a target file early after a certain threshold of certainty has been reached. In addition, using mobile anti-virus products only to scan new applications instead of enabling continuous scanning may be the preferable approach. Another possible approach is to move the main bulk of computational work required for virus scanning to the cloud (Hamzah, Khattab, & El-Gamal, 2014). With this solution, the client software running on the mobile device does not perform expensive operations, as its main function is only to upload files to the cloud and to communicate results back to the user. The tradeoff of cloud based scanning is the potential risk to user's privacy.

### 5.1.4 Mitigating Factors

Keeping devices up-to-date is crucial to ensure their security. Apple is known for providing updates for their line of mobile devices for up to five years after purchase. Furthermore, updates are delivered to compatible devices immediately after Apple releases the update. Both factors are largely due to the fact that Apple produces both the software and the hardware for their products, which shortens the supply chain and essentially eliminates the need for porting software to different hardware platforms. This long support time helps with keeping older devices secure as well. In contrast, devices running Google's Android generally get updates slower and older devices may not get updates at all (F-Secure, 2019c). Google's own line of mobile phones fares better, though.

Google is attempting to make it easier for device vendors to keep Android up-to-date through project Treble (Google, 2019a), which was made available from Android Oreo onwards. In essence, project Treble allows device vendors to re-use the hardware adaptation layer of the previous version and update only the Android framework on top. However, it will likely take several years until the effects of project Treble will be seen on a world-wide scale.

### 5.1.5 Exacerbating Factors

As mentioned in the previous subsection, software updates have posed a problem for Android for a long time. Although project Treble is aiming to improve the situation on Android, not only will it take time until it is widely adopted, but there are many devices in circulation today that are essentially abandoned by their vendors and will never receive software updates. These devices are prime targets for malware authors. In their State of Cyber Security Report for year 2017 (F-Secure, 2019c), F-Secure provided some numbers to illustrate the severity of the situation. The report compares the adoption rate of updates, in this case iOS version 10.2 and Android Nougat. Whereas iOS 10.2 enjoyed an adoption rate of over 50% in a month, Android Nougat was adopted by only 1% of devices four months after release. According to official numbers from Google's Android development site (Google, 2019b), Android Pie install base is only 10.4%, although the platform was released in August 2018 (14 months ago as of this writing).

The key difference between iOS and Android is the length of the supply chain. In the case of engineering software updates for an Android device, the *SoC (System on a Chip)* manufacturer first adapts a new Android version released by Google to their SoC, then the device vendor makes their modifications and possibly those from carriers and only

then can the software update be provisioned to users. Before project Treble, this process had to be done in a serial fashion, further prolonging the update cycle. Apple on the other hand controls the entire supply chain and is thus able to bring updates to users faster.

Another issue that specifically affects mobile devices are so called *BYOD (Bring Your Own Device)* policies in many companies (Zahadat, Blessner, Blackburn, & Olson, 2015). BYOD is a term used to refer to the fact that many companies today allow their employees to use their personal mobile devices for work purposes. In the past, mobile devices (especially laptops) were generally under company responsibility and management. IT departments installed curated software to employees' machines and policies were in place to disallow the use of personal devices and software at the work place. In many companies these practices are still in place, however the general sentiment, especially in the IT sector, has been shifting towards allowing the use of personal devices. While this brings several benefits to both the employer and employee, it raises security concerns as pointed out by Zahadat et al. (2015): personal devices may not be as strictly controlled by security policies or policies are inconsistent (e.g. leading to the use of weaker passwords), data leakage during use or after the device is eventually discarded by the user and the potential compromise of corporate networks.

## 5.2 Viruses in Cars

In the past decade, the number of computers in cars has skyrocketed. Many new cars sold today come with built-in computers that offer full-featured infotainment applications to drivers and passengers. Some car makers have gone further, replacing traditional instrument clusters that used mechanical gauges to show speed, RPM and other essential info to the driver with real-time rendered equivalents on digital displays. Some car manufacturers have gone further yet, forgoing almost all mechanical controls in favor of a touch screen based user interface. Furthermore, many cars sold today come with built-in wireless capabilities for the car occupants, such as Bluetooth enabled multimedia and phone connectivity. Some cars offer a Wi-Fi hotspot for connecting passengers' devices to the Internet through a cellular connection built into the vehicle.

In addition to connecting vehicle occupants to the car and to the Internet, modern cars are also getting more and more connected to the world around them. *Connected car* is a term that has gained traction in the recent years to capture this phenomenon. Reasons for the increased connectivity are manifold, however improved road safety and traffic efficiency are currently the main motivators behind this emerging technology

(ETSI, 2009). The umbrella term *V2X (Vehicle-to-everything)* covers the various ways a car connects to its environment. Under this umbrella, the following terms are often used to further specify the nature of the connection: *V2V (Vehicle-to-Vehicle)*, *V2I (Vehicle-to-Infrastructure)*, *V2P (Vehicle-to-Pedestrian)* and *V2C (Vehicle-to-Cloud)* (Zhang et al., 2018). V2X technologies are currently not yet in wide-spread use and standardization as well as initial deployments are on their way. However, the trend is clear: the degree of connectivity of cars will increase in the coming years.

As the connected car industry is still in its relative infancy, no wide-spread virus attacks are yet known. However, multiple incidents of vulnerabilities and product recalls by car manufacturers to fix security problems have already been witnessed. Furthermore, potential attack vectors for viruses have been identified by researchers. This chapter will explore the related research and provide an overview of the current state of affairs in the automotive space.

This section is organized as follows: Subsection 5.2.1 provides essential background information about modern cars. Based on that understanding, Subsection 5.2.2 explores potential attack vectors. Subsection 5.2.3 discusses some approaches on countermeasures. Finally, some mitigating as well as exacerbating factors regarding automotive virus attacks are considered in Subsections 5.2.4 and 5.2.5, respectively.

### 5.2.1 Background

Cars today consist of many, in some cases several dozens, embedded computers controlling various parts of the vehicle. In the literature, these computers are generally referred to as *Electronic Control Units* or ECUs. Each ECU has a specific function: for example, the *Body Control Module* (BCM) is an ECU that controls various body functions, for example power windows. The ECUs communicate with one another by means of a *vehicle bus*. The most prevalent vehicle bus in use today is the *CAN bus*. Other vehicle buses are sometimes used in parallel with CAN, for example *MOST bus* for streaming content within the car for use cases that require a high bandwidth, such as multimedia. Safety critical data transfer happens on the CAN bus.

Cars generally have multiple CAN buses for different purposes. The most safety critical functions reside on a separate CAN bus running at a high baud rate, such as those related to braking and steering. Other, parallel CAN buses may be running at lower baud rates and dedicated to less safety critical functions such as the previously mentioned body control. On-board computers running *In-Vehicle Infotainment (IVI)* systems or *instrument cluster* software naturally need to be connected to some or all of the CAN

buses, either directly or through a CAN gateway, in order to allow the user to send control messages to connected ECUs (e.g. by pressing a button in the IVI system to open a window) or to render status information on the digital displays (e.g. speed).

A standardized diagnostic port or *On-Board Diagnostics (OBD)*, can be found in most cars. It is a physical connector accessible from the cockpit, usually from underneath the steering wheel. OBD provides diagnostic access to CAN. A technician connects an OBD reader device to the port to get diagnostic data out of ECUs connected to the CAN bus. It is generally used during vehicle inspection and maintenance operations.

At this junction, an important technical observation must be made: there is no standard architecture for cars. Generally speaking, every car manufacturer builds cars with a unique architectural design. While there are certainly similarities, no two manufacturers build cars that are exactly alike.

The connected nature of the vehicle architecture lends itself to multiple possible attack vectors. Furthermore, as even the most safety critical functions of the vehicle, such as steering and braking, are in principle accessible through CAN, a potential exist for inflicting serious harm to the vehicle occupants as well as fellow road users. Indeed, it has been shown by researchers that many vulnerabilities exist in modern vehicles that could be exploited by either a human attacker or a virus. In terms of the topic of this Thesis, the most interesting attack vectors are those that have the greatest potential to maximize the spreading of the virus. These attack vectors will be examined next.

### 5.2.2 Potential Attack Vectors

A modern car has multiple potential attack vectors for both self-replicating as well as human initiated attacks. In their groundbreaking series of studies, Miller and Valasek demonstrated many vulnerabilities in series-production cars. In their study on remote attack surfaces (Miller & Valasek, 2014), they posited the following attack vectors:

1. Passive Anti-Theft System (PATS)
2. Tire Pressure Monitoring System (TPMS)
3. Remote Keyless Entry / Start (RKE)
4. Bluetooth
5. Radio Data System

6. Telematics / Cellular

7. Wi-Fi

8. Internet / Apps

The first three attack vectors are relatively short-range and as such unlikely candidates to be targeted by viruses. On the other hand, the last five of these attack vectors, generally found in the IVI system, are different. What makes these attack vectors especially dangerous is their long range and wireless nature.

The attack vector with the most potential for a widespread virus outbreak is the cellular connection. This attack vector is illustrated in Figure 5.1. The figure depicts an IVI system connected to two CAN buses, which in turn connect to multiple ECUs. The IVI system hosts two processors: an *Application Processor* that runs the main IVI software (denoted by AP) and a discrete *Microcontroller Unit* (denoted by MCU) that runs the CAN software stack. The two processors are connected, usually either through *Serial Peripheral Interface (SPI)* or *Universal Asynchronous Receiver-Transmitter (UART)* connection. While details may vary, this is a common architecture in automotive computers.

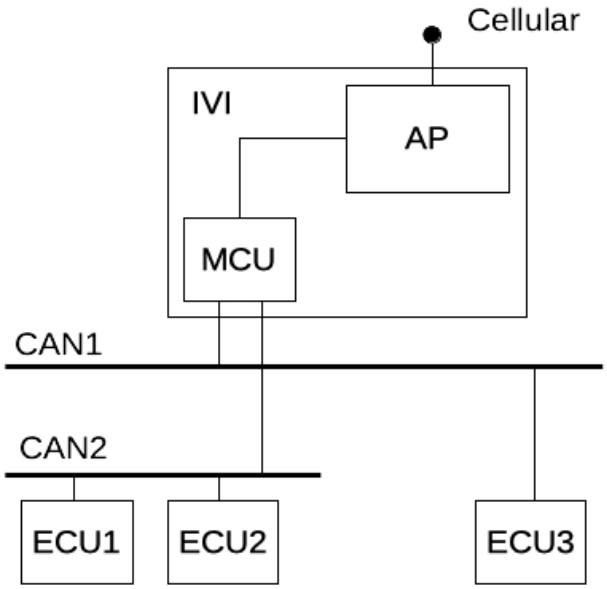



**Figure 5.1:** Attack vector through a cellular connection.

The general anatomy of an attack through a cellular connection is explained in detail by Miller and Valasek (2015). In the following description, some of the details have been simplified for brevity and terminology modified to fit the terminology used elsewhere in this Thesis. The first step in the attack sequence would be to determine the IP address of

the IVI system. After that, a vulnerability in the IVI system would have to be exploited to gain access to the software running on the Application Processor. Already at this stage a virus could cause nuisance, or at worst endanger the driver by causing a distraction, for instance by turning the heat up in the cockpit, or by turning the volume to the maximum. In order to gain a deeper access into the vehicle, the virus would have to find a way to send CAN messages to the CAN buses. Theoretically, a vulnerability could be found in the IVI software running on the Application Processor, which would allow the virus to send arbitrary CAN messages directly. If not, a vulnerability could be exploited in the MCU to allow it to be updated with a firmware that would allow communication to CAN bus.

The aforementioned attack vector was, in fact, demonstrated by Miller and Valasek (2015) on an unaltered, series production 2014 Jeep Cherokee. This eventually lead to a recall of 1.4 million potentially vulnerable vehicles. Although the purpose of the research was not aimed at specifically demonstrating a viral attack, the authors do point out that the exact same attack path could have been exploited by a virus.

Car service shops offer another potential attack vector as shown by Kleberger, Olovsson, and Jonsson (2012). As cars are brought to service, a diagnostic device is used to connect to the OBD-II port to get a reading on ECU status as well as update ECUs, if needed. A compromised diagnostic device or the internal network of the service shop may be a potential target for an attacker. As service shops are visited by a number of vehicles on a daily basis, a potential exists for a large-scale virus outbreak. In a study by Checkoway et al. (2011), the researchers were in fact able to demonstrate both the existence of a vulnerable diagnostic device, as well as the ability to send malicious CAN messages to the car. It is worth pointing out that cars are particularly vulnerable when they are at the service center in diagnostic mode as it allows an unfettered access to the vehicle ECUs. Furthermore, the researchers were able to demonstrate that a virus could spread from one diagnostic device to another at the car service center. In principle, an infected diagnostic device would then be able to attack any vehicle that comes in for a service.

V2X based security threats are extensively studied by Engoulou, Bellaïche, Pierre, and Quintero (2014). Various potential ways of causing harm are presented, either by attacking V2X *Road-Side Units* (RSUs), so that the RSUs send false information to passing cars (or delay, or stop sending information altogether). With regards to V2X, attack vectors can be found either from the infrastructure to the car, or from the car to the infrastructure or in the case of Vehicle-to-Vehicle communications (V2V), from one car to another. V2X attack vectors are wireless in nature, but not necessarily long range.

However, the multi-hop nature of V2X technologies opens up a possibility for a virus to spread, in case a suitable vulnerability is found and exploited.

### 5.2.3 Countermeasures

The basic strategy to secure a connected vehicle is based on a layered approach, where safety critical parts are isolated from the rest. As seen in Section 5.2.2, the IVI unit specifically is a prime target for an attack due to a wide attack surface and in many cases a direct access to the CAN bus. Thus, defending the periphery by hardening the IVI unit should be one of the first steps. In addition, defenses should be built within the IVI unit to minimize the impact of potential vulnerabilities. Unnecessary access to the CAN bus should be limited, for example, by means of software or hardware based isolation or both.

In their study on attack surfaces, Miller and Valasek (2014) propose that a mechanism to detect an on-going attack directly from the CAN bus could be a feasible approach. It turns out that an attack on the CAN bus has a distinct signature which could be detected either by a software based algorithm running on one or more ECUs or a hardware based system connected to the CAN bus or through the OBD-II port.

After vulnerabilities are found and fixes issued by car manufacturers, software updates including those fixes must find their way into cars. Software updates to patch found vulnerabilities pose a real problem to the car industry, as the lifetime of cars is relatively long and users may be running outdated software for a long time. Furthermore, older cars do not necessarily offer mechanisms for over-the-air updates or user friendly USB-based updates, making a visit to a repair shop necessary for many people. Hence, raising awareness of the problem and making the update process as easy as possible is going to be key in keeping cars secure going forward.

### 5.2.4 Mitigating Factors

Compared to the desktop computer or mobile phone space, there are some differentiating factors that may mitigate the spread of viruses. The primary difference is fragmentation. The IVI market is extremely heterogeneous: many car manufacturers have their own proprietary systems built from ground up, with functionalities and user interfaces specific to their vehicles. Even the underlying operating systems vary. Linux, Android and QNX are some of the most used, but many variants and versions of these base systems are in circulation. Some unification efforts have been under way for many years,

most notably *GENIVI* and *Automotive Grade Linux*, however the automotive market remains fragmented.

As discussed in the previous subsection, not only are the IVI computers different between many cars, so is generally speaking the entire vehicle architecture between cars of different manufacturers. Furthermore, the degree of connectivity between vehicles varies wildly: some cars are always connected to the Internet, whereas some do not have any Internet connectivity at all. This variegated environment in itself has a chance of mitigating or at least slowing down the spread of viruses.

Compared to desktop computers and mobile phones, the initial cost of discovering new, undisclosed vulnerabilities is higher with cars. The virus writers may have to purchase or get access to expensive equipment and perhaps even the target vehicle, before a proper understanding of the underlying architecture can be gained and security vulnerabilities found.

A hypothesis could be made that whatever viral attacks we may see in the near future, they will probably be extremely narrowly targeted at a specific manufacturer, perhaps even at a specific model and year. Naturally, this does not diminish the seriousness of potential viral outbreaks, even if the scale of the attacks may remain limited for the time being compared to some of the attacks we have seen in recent years in desktop computers. Cars are special in that they may put human beings directly in harm's way.

### 5.2.5 Exacerbating Factors

The automotive industry has several special characteristics that may work in favor of virus writers. The most notable factor is the fact that the modern vehicle is becoming exceedingly connected and computerized, meaning that the attack surface is growing over time.

Another emerging trend are various *Advanced Driver Assistance Systems* (ADAS) as well as self-driving capabilities. The combination of ubiquitous connectivity and increasing computer control over safety critical functions is potentially hazardous, as vulnerabilities may enable attackers to endanger the safety of the car occupants.

The upcoming V2X technologies may bring further challenges. As one of the key functions of V2X is increasing safety, some features may allow the car to react automatically to safety related warnings from the V2X systems (e.g. engage an emergency braking procedure in case of a perceived danger on the road). This, in turn, may open up new attack vectors allowing attackers to gain physical control over cars.

Compared to desktop, server and mobile phone markets, the lifetime of a car is long. It is not unusual to find over a decade old vehicles in active use. This means that bringing software updates to older vehicles is critical, especially those with vulnerabilities open to remote attacks. However, as over-the-air updates are not a standard feature especially in older vehicles, provisioning updates quickly and efficiently to car owners may not be an easy task.

## 5.3 Summary

This chapter examined the security of two broad product categories, mobile phones and cars, in order to show that neither is inherently safe from attackers. Quite the contrary, mobile phones have already seen a plethora of malware and their numbers are rising steadily every year. Cars on the other hand have not yet been the target of wide-scale attacks, however it is clear that a potential to cause bodily harm and financial damage is real. Thus, car manufacturers should take the necessary steps to harden onboard computers and raise security awareness amongst their customers in order to keep car owners and other road users safe.

# 6. Conclusion and Future Work

This Thesis has provided four different perspectives to computer viruses. First, a *retrospective* look into viruses was taken in Chapter 2, going back 70 years all the way to the beginning and from there step by step to the present. Second, viruses were examined from an *offensive* point of view in Chapter 3, iterating different types of attack mechanisms viruses employ to infect targets and to spread. The third viewpoint was a *defensive* one: Chapter 4 covered the basic principles of defense mechanisms. Finally, an *applied* perspective was taken in Chapter 5, where two modern types of computers, mobile phones and cars, and their susceptibility to viral attacks was examined.

Why do viruses exist, then? One tempting viewpoint to this question is to look at life out in the nature, as well as natural evolution. Living organisms require an operational environment where they are able to reproduce and evolve. It turns out that these operational environments may range from those familiar to us humans, to some of the most hostile places on Earth - wherever we look, life always seems to find a way to carve a niche to flourish in. This seems to be true of computer viruses (and other types of malware) as well. If the computing platform can support it, one can be sure to find a malware specimen on it sooner or later. Of course, the difference to the natural environment is that viruses and malware are specifically written by programmers. Another parallel to nature can be drawn by considering predators and their prey. As a predator gets better at catching their prey, so does the prey get better at evading the predator. Over time, both sides evolve to employ ever more ingenious mechanisms of survival. Similarly, both malware and various security products evolve hand in hand through a constant tug of war that forces both sides to become better at their game.

Today, writing malware can be a profitable business. As seen in Section 2.12, one popular way to make money is through ransomware: a victim's personal files are encrypted and the attacker demands money in exchange for decrypting them. This attack is particularly insidious when combined with a worm-like propagation mechanism as was the case with WannaCry. Other business models exist as well, such as selling exploits, or selling capacity in a *botnet*. The raise of cryptocurrencies such as Bitcoin in

recent years has made it easier for money to change hands without compromising the identity of either the selling or the buying party.

A topic that was deliberately left out of the earlier chapters was self-reproducing programs in a more philosophical and speculative sense. Could there be a benevolent or even beneficial kind of a computer virus? As was discussed in Chapter 2, the first viruses and worms were in fact made for beneficial purposes. The evolutionary biologist Richard Dawkins speculates in his article Viruses of the Mind (Dawkins, 1991) that in the future the digital ecosystem, or "silicosphere" as he puts it, might be filled with both good and bad viruses, possibly co-operating or fighting one another. Similar to nature, viruses could mutate as they reproduce and then evolve by means of natural selection. New species would be born and unsuccessful species would become extinct. Viruses could evolve to learn to benefit from other viruses, creating synergetic communities. Although currently the most prolific means for viruses to evolve is by human hands, some glimpses into Dawkins' ideas have already been encountered in reality. As the silicosphere grows ever larger and more interconnected, and as hardware and software gets more powerful and sophisticated every year, Dawkins' ideas might not sound too far-fetched after all.

While this Thesis has provided a comprehensive overview on the topic of computer viruses, many things were necessarily left unsaid due to the enormity of the field. Hence, the field is ripe for exploration on future topics of study through a multitude of perspectives. One possible perspective is to expand the work started in Chapter 5 by looking at various types of computer systems that may be susceptible to virus attacks, such as Internet of Things (IoT), medical systems, military equipment or electronic voting systems. Another perspective is to look at emerging technologies - i.e. technologies that are today only in their infancy, but may one day be ubiquitously used by a large amount of people. Devices that will be used to augment the capabilities of human biology in the future could fall into this category. Yet another perspective is to consider how advances in the field of *Artificial Intelligence (AI)* might make both viruses and their counterparts smarter and more autonomous. Could one envision a nearly, or fully, autonomous *supervirus* (be it malicious or benevolent) that would cut the navel cord with its creators and start to live a life of its own?

At the end of the introductory chapter of this Thesis, a question was posed: between viruses and anti-viruses, who will eventually win? Given the multidimensionality of both the virus problem as well as proposed defense techniques, as seen in Chapters 2 through 5, an observant reader may have already conjenctured that perhaps this question does not have a satisfactory answer. Indeed, in his seminal paper Computer viruses: Theory and experiments (Cohen, 1987), Fred Cohen showed that the detection of a virus

is an undecidable problem. Furthermore, he posited that the only way to fully protect a system from viruses is to isolate it completely from its environment. Obviously, this is not a practical solution for most people. It is thus clear, that the cat and mouse game between viruses and anti-viruses will continue unabated and unresolved - indefinitely.

# References


Apple. (2019). Ios security (ios 12.3). Retrieved October 19, 2019, from https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf

Bania, P. (2005). Fighting epo viruses. Retrieved October 17, 2019, from https://www.symantec.com/connect/articles/fighting-epo-viruses

Bayer, U., Kruegel, C., & Kirda, E. (2006). Ttanalyze: A tool for analyzing malware.

Bayer, U., Moser, A., Kruegel, C., & Kirda, E. (2006). Dynamic analysis of malicious code. *Journal in Computer Virology*, *2*(1), 67–77. doi:10.1007/s11416-006-0012-2

Bielecki, A., & Hajto, P. (2004). A neural-based agent for ip traffic scanning and worm detection. In L. Rutkowski, J. H. Siekmann, R. Tadeusiewicz, & L. A. Zadeh (Eds.), *Artificial intelligence and soft computing - icaisc 2004* (pp. 816–822). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bittau, A., Handley, M., & Lackey, J. (2006). The final nail in wep's coffin. In *2006 ieee symposium on security and privacy (s p'06)* (15 pp.–400). doi:10.1109/SP.2006.40

Bontchev, V. (1998). Refereed paper: Macro virus identification problems. *Comput. Secur. 17*(1), 69–89. doi:10.1016/S0167-4048(97)80275-1

Bose, A., & Shin, K. G. (2006). On mobile viruses exploiting messaging and bluetooth services. In *2006 securecomm and workshops* (pp. 1–10). doi:10.1109/SECCOMW.2006.359562

Ralf Brown's Interrupt List. (2019). Retrieved October 16, 2019, from https://www.cs.cmu.edu/~ralf/files.html

CAIDA. (2001). CAIDA Analysis of Code-Red. Retrieved October 16, 2019, from http://www.caida.org/research/security/code-red/

Caswell, B., Foster, J. C., Russell, R., Beale, J., & Posluns, J. (2003). *Snort 2.0 intrusion detection*. Syngress Publishing.

Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., ... Kohno, T. (2011). Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th usenix conference on security* (pp. 6–6). SEC'11. San Francisco, CA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=2028067.2028073

Cohen, F. (1987). Computer viruses: Theory and experiments. *Computers & Security*, *6*(1), 22–35. doi:https://doi.org/10.1016/0167-4048(87)90122-2

Dawkins, R. (1991). Viruses of the mind. Retrieved October 17, 2019, from https://www.inf.fu-berlin.de/lehre/pmo/eng/Dawkins-MindViruses.pdf

Debar, H., Becker, M., & Siboni, D. (1992). A neural network component for an intrusion detection system. *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, 05/04-06/92*.

Einwechter, N. (2002). An introduction to distributed intrusion detection systems. Retrieved October 20, 2019, from https://www.symantec.com/connect/articles/introduction-distributed-intrusion-detection-systems

Engoulou, R. G., Bellaïche, M., Pierre, S., & Quintero, A. (2014). Vanet security surveys. *Computer Communications*, *44*, 1–13. doi:https://doi.org/10.1016/j.comcom.2014.02.020

ESET. (2018). Lojax. Retrieved October 17, 2019, from https://cdn1.esetstatic.com/ESET/US/resources/datasheets/ESETus-datasheet-lojax.pdf

ETSI. (2009). Etsi tr 102 638 v1.1.1 intelligent transport systems (its). Retrieved October 19, 2019, from https://www.etsi.org/deliver/etsi_tr/102600_102699/102638/01.01.01_60/tr_102638v010101p.pdf

F-Secure. (2019a). Brain. Retrieved October 16, 2019, from https://www.f-secure.com/v-descs/brain.shtml

F-Secure. (2019b). Michelangelo. Retrieved October 17, 2019, from https://www.f-secure.com/v-descs/michel.shtml

F-Secure. (2019c). State of cyber security 2017. Retrieved October 19, 2019, from https://www.f-secure.com/content/dam/f-secure/en/labs/whitepapers/Cyber_Security_Report_2017.pdf

F-Secure. (2019d). Trojan:android/gepew. Retrieved October 19, 2019, from https://www.f-secure.com/v-descs/trojan_android_gepew.shtml

F-Secure. (2019e). Trojan:symbos/viver.a. Retrieved October 19, 2019, from https://www.f-secure.com/v-descs/trojan_symbos_viver_a.shtml

F-Secure. (2019f). Virus:w32/concept. Retrieved October 16, 2019, from https://www.f-secure.com/v-descs/concept.shtml

F-Secure. (2019g). Worm:symbos/commwarrior. Retrieved October 19, 2019, from https://www.f-secure.com/v-descs/commwarrior.shtml

F-Secure. (2019h). Worm:w32/mtx. Retrieved October 17, 2019, from https://www.f-secure.com/v-descs/mtx.shtml

F-Secure. (2019i). Worm:w32/slammer. Retrieved October 18, 2019, from https://www.f-secure.com/v-descs/mssqlm.shtml

Fawcett, T., & Provost, F. (1999). Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the fifth acm sigkdd international conference on*

*knowledge discovery and data mining* (pp. 53–62). KDD '99. doi:10.1145/312129.312195

FBI. (2019). The melissa virus. Retrieved October 16, 2019, from https://www.fbi.gov/news/stories/melissa-virus-20th-anniversary-032519

Ferrie, P. (2007). Attacks on more virtual machine emulators. *Symantec Technology Exchange*, *55*.

Ferrie, P., Perriot, F., & Ször, P. (2004). Chiba witty blues. Retrieved October 17, 2019, from http://pferrie.tripod.com/papers/witty.pdf

Ferrie, P., & Ször, P. (2004). Cabirn fever. Retrieved October 17, 2019, from https://www.virusbulletin.com/uploads/pdf/magazine/2004/200408-analysis-PeterFerrie-PeterSzor.pdf

Fleizach, C., Liljenstam, M., Johansson, P., Voelker, G. M., & Mehes, A. (2007). Can you infect me now?: Malware propagation in mobile phone networks. In *Proceedings of the 2007 acm workshop on recurring malcode* (pp. 61–68). WORM '07. doi:10.1145/1314389.1314402

Frigo, P., Giuffrida, C., Bos, H., & Razavi, K. (2018). Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *2018 ieee symposium on security and privacy (sp)* (pp. 195–210). doi:10.1109/SP.2018.00022

Google. (2018). Android enterprise security white paper. Retrieved October 19, 2019, from https://source.android.com/security/reports/Google_Android_Enterprise_Security_Whitepaper_2018.pdf

Google. (2019a). Android architecture documentation. Retrieved October 19, 2019, from https://source.android.com/devices/architecture

Google. (2019b). Android distribution dashboard. Retrieved October 19, 2019, from https://developer.android.com/about/dashboards

Hamzah, A. A., Khattab, S. M., & El-Gamal, S. S. (2014). Cloud antivirus cost model using machine learning. In *2014 9th international conference on informatics and systems* (PDC-1-PDC–8). doi:10.1109/INFOS.2014.7036708

Heartfield, R., & Loukas, G. (2015). A taxonomy of attacks and a survey of defence mechanisms for semantic social engineering attacks. *ACM Comput. Surv. 48*(3), 37:1–37:39. doi:10.1145/2835375

Hollebeek, T., & Waltzman, R. (2004). The role of suspicion in model-based intrusion detection. In *Proceedings of the 2004 workshop on new security paradigms* (pp. 87–94). NSPW '04. doi:10.1145/1065907.1066041

Hyppönen, M. (1994). Retroviruses - how viruses fight back. Retrieved October 17, 2019, from https://archive.org/details/Mikko_Retroviruses_1994_06

Intel. (2019). Intel® 64 and ia-32 architectures software developer manuals. Retrieved October 17, 2019, from https://software.intel.com/en-us/articles/intel-sdm

Jin, H., Yang, Z., Sun, J., Tu, X., & Han, Z. (2005). Cips: Coordinated intrusion prevention system. In C. Kim (Ed.), *Information networking. convergence in broadband and mobile networking* (pp. 89–98). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kleberger, P., Olovsson, T., & Jonsson, E. (2012). An in-depth analysis of the security of the connected repair shop. In *Proceedings of the seventh international conference on systems and networks communications, icsnc* (pp. 99–107).

Kurzban, S. (1989). Viruses and worms - what can they do? *SIGSAC Rev. 7*(1), 16–32. doi:10.1145/70951.70954

Ludwig, M. A. (1991). *The little black book of computer viruses: The basic technology*. American Eagle Publications.

Ludwig, M. A. (1998). *The giant black book of computer viruses* (2nd). American Eagle Publications.

Martin, E. (2019). John conway's game of life. Retrieved October 16, 2019, from http://www.bitstorm.org/gameoflife/

McAfee. (2018). Mobile threat report q1, 2018. Retrieved October 19, 2019, from https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf

Miller, C., & Valasek, C. (2014). A survey of remote automotive attack surfaces. *black hat USA*, *2014*, 94.

Miller, C., & Valasek, C. (2015). Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, *2015*, 91.

Mori, A., Izumida, T., Sawada, T., & Inoue, T. (2006). A tool for analyzing and detecting malicious mobile code. In *Proceedings of the 28th international conference on software engineering* (pp. 831–834). ICSE '06. doi:10.1145/1134285.1134426

Nazario, J. (2003). *Defense and detection strategies against internet worms*. Norwood, MA, USA: Artech House, Inc.

Perriot, F., & Ször, P. (2003). An analysis of the slapper worm exploit. Retrieved October 17, 2019, from http://www.symantec.com/avcenter/reference/analysis.slapper.worm.pdf

Pietrek, M. (2002). An in-depth look into the win32 portable executable file format.

Pillai, M. M., Eloff, J. H. P., & Venter, H. S. (2004). An approach to implement a network intrusion detection system using genetic algorithms. In *Proceedings of the 2004 annual research conference of the south african institute of computer scientists and information technologists on it research in developing countries* (pp. 221–221). SAICSIT '04. Stellenbosch, Western Cape, South Africa: South African Institute for Computer Scientists and Information Technologists. Retrieved from http://dl.acm.org/citation.cfm?id=1035053.1035080

Polakis, I., Diamantaris, M., Petsas, T., Maggi, F., & Ioannidis, S. (2015). Powerslave: Analyzing the energy consumption of mobile antivirus software. In *Proceedings of the 12th international conference on detection of intrusions and malware, and vulnerability assessment - volume 9148* (pp. 165–184). DIMVA 2015. doi:10.1007/978-3-319-20550-2_9

Rabek, J. C., Khazan, R. I., Lewandowski, S. M., & Cunningham, R. K. (2003). Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 acm workshop on rapid malcode* (pp. 76–82). WORM '03. doi:10.1145/948187.948201

Ramachandran, G., & Hart, D. (2004). A p2p intrusion detection system based on mobile agents. In *Proceedings of the 42nd annual southeast regional conference* (pp. 185–190). ACM-SE 42. doi:10.1145/986537.986581

Rubin, S., Jha, S., & Miller, B. P. (2006). Protomatching network traffic for high throughputnetwork intrusion detection. In *Proceedings of the 13th acm conference on computer and communications security* (pp. 47–58). CCS '06. doi:10.1145/1180405.1180413

Salomon, D. (2010). *Elements of computer security* (1st). Springer Publishing Company, Incorporated.

Sarkar, P. (2000). A brief history of cellular automata. *ACM Comput. Surv. 32*(1), 80–107. doi:10.1145/349194.349202

Shoch, J. F., & Hupp, J. A. (1982). The "worm"programs - early experience with a distributed computation. *Commun. ACM*, *25*(3), 172–180. doi:10.1145/358453.358455

snort.org. (2019). Snort. Retrieved October 18, 2019, from https://www.snort.org

Spafford, E. H. [E. H.]. (1989). Crisis and aftermath. *Commun. ACM*, *32*(6), 678–687. doi:10.1145/63526.63527

Spafford, E. H. [Eugene H], & Zamboni, D. (2000). Intrusion detection using autonomous agents. *Computer Networks*, *34*(4), 547–570. Recent Advances in Intrusion Detection Systems. doi:https://doi.org/10.1016/S1389-1286(00)00136-5

Staniford, S., Paxson, V., & Weaver, N. (2002). How to own the internet in your spare time. In *Proceedings of the 11th usenix security symposium* (pp. 149–167). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=647253.720288

Stasiukonis, S. (2006). Social engineering, the usb way. Retrieved October 18, 2019, from https://www.darkreading.com/attacks-breaches/social-engineering-the-usb-way/d/d-id/1128081

Stavroulakis, P., & Stamp, M. (2010). *Handbook of information and communication security* (1st). Springer Publishing Company, Incorporated.

Stolfo, S. J., Hershkop, S., Bui, L. H., Ferster, R., & Wang, K. (2005). Anomaly detection in computer security and an application to file system accesses. In M.-S. Hacid, N. V. Murray, Z. W. Raś, & S. Tsumoto (Eds.), *Foundations of intelligent systems* (pp. 14–28). Berlin, Heidelberg: Springer Berlin Heidelberg.

Suarez-Tangil, G., Tapiador, J., Peris-Lopez, P., & Ribagorda, A. (2013). Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, *16*. doi:10.1109/SURV.2013.101613.00077

Summerville, D., Skormin, V., Volynkin, A., & Moronski, J. (2005). Prevention of information attacks by run-time detection of self-replication in computer codes. In V. Gorodetsky, I. Kotenko, & V. Skormin (Eds.), *Computer network security* (pp. 54–75). Berlin, Heidelberg: Springer Berlin Heidelberg.

Symantec. (1997). Understanding heuristics: Symantec's bloodhound technology. Retrieved October 17, 2019, from http://www.symantec.com/avcenter/reference/heuristc.pdf

Symantec. (2001). Hunting for metamorphic. Retrieved October 17, 2019, from http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf

Symantec. (2006). Malicious cryptography. Retrieved October 17, 2019, from https://www.symantec.com/connect/articles/malicious-cryptography-part-two

Symantec. (2007a). CodeRed Worm. Retrieved October 16, 2019, from http://www.symantec.com/security_response/writeup.jsp?docid=2001-071911-5755-99&tabid=2

Symantec. (2007b). Tequila.a. Retrieved October 17, 2019, from https://www.symantec.com/security-center/writeup/2000-121813-3642-99

Symantec. (2007c). W97M.Melissa.A. Retrieved October 16, 2019, from https://www.symantec.com/security-center/writeup/2000-122113-1425-99

Symantec. (2011). W32.stuxnet dossier. Retrieved October 17, 2019, from https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

Symantec. (2012). XM.Laroux. Retrieved October 16, 2019, from https://www.symantec.com/security-center/writeup/2000-121913-4933-99

Symantec. (2017). Ransom.wannacry. Retrieved October 17, 2019, from https://www.symantec.com/security-center/writeup/2017-051310-3522-99

Ször, P. (1998). Bad idea. Retrieved October 17, 2019, from https://www.virusbulletin.com/uploads/pdf/magazine/1998/199804.pdf

Ször, P. (2005). *The art of computer virus research and defense*. Addison-Wesley Professional.

Teng, H. S., Chen, K., & Lu, S. C. (1990). Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Proceedings. 1990 ieee computer*

*society symposium on research in security and privacy* (pp. 278–284). doi:10.1109/RISP.1990.63857

AV-Test. (2019). AV-Test Security Report 2018/2019. Retrieved October 17, 2019, from https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf

Thompson, K. (1984). Reflections on trusting trust. *Commun. ACM*, *27*(8), 761–763. doi:10.1145/358198.358210

Venugopal, D. (2006). An efficient signature representation and matching method for mobile devices. In *Proceedings of the 2nd annual international workshop on wireless internet*. WICON '06. doi:10.1145/1234161.1234177

Virus Bulletin. (2016). How it works: Steganography hides malware in image files. Retrieved October 21, 2019, from https://www.virusbulletin.com/virusbulletin/2016/04/how-it-works-steganography-hides-malware-image-files/

Wagner, D., & Soto, P. (2002). Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th acm conference on computer and communications security* (pp. 255–264). CCS '02. doi:10.1145/586110.586145

Walker, J. (1996a). Pervade source code. Retrieved October 16, 2019, from http://www.fourmilab.ch/documents/univac/pervade.html

Walker, J. (1996b). The animal episode. Retrieved October 16, 2019, from http://www.fourmilab.ch/documents/univac/animal.html

Weaver, N., & Ellis, D. (2004). Reflections on witty. Retrieved October 17, 2019, from https://www.usenix.org/system/files/login/articles/1104-weaver.pdf

Webster, M., & Malcolm, G. (2006). Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, *2*(3), 149–161. doi:10.1007/s11416-006-0023-z

Wikipedia. (2019a). Stagefright (bug). Retrieved October 19, 2019, from https://en.wikipedia.org/wiki/Stagefright_(bug)

Wikipedia. (2019b). Steganography. Retrieved October 21, 2019, from https://en.wikipedia.org/wiki/Steganography

Wikipedia. (2019c). Ultra-wideband. Retrieved October 19, 2019, from https://en.wikipedia.org/wiki/Ultra-wideband

Young, A., & Yung, M. (1996). Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings of the 1996 ieee symposium on security and privacy* (pp. 129–). SP '96. Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=525080.884259

Zahadat, N., Blessner, P., Blackburn, T., & Olson, B. A. (2015). Byod security engineering: A framework and its analysis. *Computers & Security*, *55*, 81–99. doi:https://doi.org/10.1016/j.cose.2015.06.011

Zhang, S., Chen, J., Lyu, F., Cheng, N., Shi, W., & Shen, X. (2018). Vehicular communication networks in the automated driving era. *IEEE Communications Magazine*, *56*(9), 26–32. doi:10.1109/MCOM.2018.1701171

Zou, C. C., Towsley, D., & Gong, W. (2004). A firewall network system for worm defense in enterprise networks.

Zou, C. C., Towsley, D., Gong, W., & Cai, S. (2005). Routing worm: A fast, selective attack worm based on ip address information. In *Proceedings of the 19th workshop on principles of advanced and distributed simulation* (pp. 199–206). PADS '05. doi:10.1109/PADS.2005.24