DEVELOPMENT OF A VIRUS DETECTION SYSTEM FOR DOS ENVIRONMENT

A DISSERTATION
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

MASTER OF ENGINEERING
IN
COMPUTER TECHNOLOGY & APPLICATIONS

SUBMITTED BY
SUNITA VERMA
(Roll No: 3009)

UNDER THE GUIDANCE OF
Mrs. RAJNI JINDAL

DEPATRTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
NEW DELHI-110042
2005

Department of Computer Engineering
Delhi College of Engineering, Delhi

CERTIFICATE

This is to certify that the project report entitled **"Development of a Virus Detection System for DOS environment"** being submitted by **Ms. SUNITA VERMA (Roll No: 3009)** is a bonafide record of her own work carried under our guidance and supervision in partial fulfillment for the award of the degree of Master of Engineering in Computer Technology and Applications from Delhi College of Engineering, Delhi.

Mrs. Rajni Jindal
Lecturer, Project Guide
Delhi College of Engg.
Delhi-42

Dr. D. Roy Choudhury
Professor & HOD
Delhi College of Engg.
Delhi-42

ACKNOWLEDGEMENT

I wish to express my deep sense of gratitude and veneration to my project guide **Mrs. Rajni Jindal**, Lecturer, Department of Computer Engineering, Delhi College of Engineering, Delhi, for her perpetual encouragement, constant guidance, valuable suggestions and continuous motivation which has enabled me to complete this work.

I would like to express my sincere thanks to **Dr. P. B. Sharma**, Principal, Delhi College of Engineering, Delhi, to allow me to perform this study and for providing all the necessary facilities to carry out this work.

I am deeply indebted to **Dr. D. Roy Choudhury**, HOD, Department of Computer Engineering, Delhi College of Engineering, Delhi, for his constant encouragement, valuable guidance, resourceful suggestions and alignment evaluations throughout the course of this project.

I am also thankful to Mrs. Goldie Gabrani, Assistant Professor, Delhi College of Engineering, Delhi and to all the teachers of Computer Engineering department for their kind co-operation and enormous support.

I am also grateful to my parents, for being a constant source of inspiration, and for enabling me to reach at this stage. A special appreciation also goes to all my friends for their love and constant support.

SUNITA VERMA

ABSTRACT

This dissertation presents an implementation of a Virus Detection System to detect, known as well as unknown viruses. It is an integration of two detection tools, Integrity Checker and Signature Scanner. Integrity checker tool uses SHA-1 algorithm to generate the checksum of a file to avoid forgery. SHA-1 gives 160-bit checksum, which is larger than that of CRC-32 (32-bit) that makes it more resistant to brute force attacks, such as Birthday attacks, which choose messages at random in an attempt to generate the same checksum. By using integrity checking technique virus detection system is able to detect all the infections whether it is due to known virus or unknown virus. It generates the list of all infected files, which are then scanned by signature scanner to get more details about the infection like name of the virus, location of infection in file etc. The signature database can always be updated by using database maintaining program, which is provided as a part of signature scanning tool. To reduce the scanning time, signature scanner uses Boyer-Moore-Horspool (BMH), a fast pattern-matching algorithm. It showed the best performance among commonly used pattern matching algorithms like Boyer Moore and Turbo-Boyer-Moore algorithms.

CONTENTS

CERTIFICATE i

ACKNOWLEDGEMENT ii

ABSTRACT iii

TABLE OF CONTENTS iv

LIST OF FIGURES viii

LIST OF TABLES viii

CHAPTER 1 INTRODUCTION 1

1.1 Motivation 1

1.2 Objective 1

1.3 Contribution 1

1.4 Organization of dissertation 3

CHAPTER 2 WHAT IS A COMPUTER VIRUS? 4

2.1 Related Software 7

2.2 Classification of Computer Viruses 7

2.2.1 Type of infection technique 8

2.2.1.1 Shell viruses 8

2.2.1.2 Add-on viruses 8

2.2.1.3 Intrusive viruses 9

2.2.1.4 Inserting virus 10

2.2.2 Type of Host Victim 11

2.2.2.1 System Sector Virus 11

2.2.2.2 File Virus 12

2.2.2.3 Companion Virus 12

2.2.2.4 Multipartite Virus 13

2.2.3 Type of virus activation 13

2.2.3.1 Direct action virus 13

2.2.3.2 Memory resident virus 13
2.2.4 Special Virus Features 14
2.2.4.1 Stealth Virus 14
2.2.4.2 Polymorphic Virus 15

CHAPTER 3 ANTIVIRUS SOLUTIONS 17
3.1 Signature scanner 17
3.2 Integrity checkers 18
3.3 Activity monitor 19
3.4 Heuristics Scanner 20

CHAPTER 4 SPECIFICATIONS OF VIRUS DETECTION SYSTEM 23
4.1 Generic Approach: An Integrity Checker 23
4.2 Specific Approach: A Signature Scanner 24
4.3 Combining Generic and Virus-Specific Approaches 25

CHAPTER 5 INTEGRITY CHECKER 26
5.1 Checksumming 27
5.2 Implementation Modes 28
5.2.1 On-demand 28
5.2.2. Resident 28
5.2.3. Self-test 29
5.3 Criteria for Choice of Hash Function 30
5.4 Description Of SHA-1 31
5.4.1 Bit Strings and Integers 32
5.4.2 Operations on WORDS 33
5.4.3 Message Padding 34
5.4.4 Functions Used 35
5.4.5 Constants Used 36
5.4.6 Computing the Message Digest 36
5.5 Description of Integrity Checker Tool 37

5.5.1 Implementation 37
5.5.2 The Code 39
5.5.2.1 Building the Checksum File 39
5.5.2.2 Calculating the File Checksum 40
5.5.2.3 Checking the Files 41
5.6 Comparison of SHA-1 with CRC-32 41

CHAPTER 6 SIGNATURE SCANNING TECHNIQUE 43
6.1 Pattern Matching Algorithms 44
6.1.1 Sequential pattern matching algorithm 44
6.1.2 Boyer Moore Algorithm 44
6.1.2.1 Performance analysis of BM 47
6.1.3 Turbo Boyer Moore Algorithm 47
6.1.3.1 Performance analysis of TBM 49
6.1.4 The Boyer-Moore-Horspool Algorithm 49
6.1.4.1 Description of the algorithm 50
6.1.4.1.1 Implementation and data structures 50
6.2 Description of the Signature Scanner 52
6.2.1 Implementation 52
6.2.1.1 Signature Database 52
6.2.1.2 Virus detection engine 53
6.3 Performance analysis 54
6.3.1 Measures 54
6.3.2 Choice of Patterns 55
6.3.3 Search for Boot sector viruses 55
6.3.4 Search for Partition table viruses 55
6.3.5 Search for File type viruses 56
6.3.6 Performance According to Pattern Size 57
6.4 Summary of results 58

CHAPTER 7 CONCLUSION AND FUTURE WORK 60

REFERENCES **62**
APPENDIX A: OUTPUT SNAPSHOT **66**
APPENDIX B: SOURCE CODE **74**

LIST OF FIGURES

Figure 2.1 Shell Virus Infection 8

Figure 2.2 Prepending virus 9

Figure 2.3 Appending Virus 9

Figure 2.4 Intrusive Virus Infection 10

Figure 2.5 Inserting virus 10

Figure 6.1 The good-suffix shift, *u* re-occurs preceded by a character *c* 45

Figure 6.2 The good-suffix shift, only a suffix of *u* re-occurs in *x*. 45

Figure 6.3 The bad-character shift, *a* occurs in *x*. 46

Figure 6.4 The bad-character shift, *b* does not occur in *x*. 46

Figure 6.5 A turbo-shift can apply when $|v|<|u|$. 48

Figure 6.6 $c \neq d$ so they cannot be aligned with the same character in *v* 48

Figure 6.7 Performance graph on the basis of Signature Database size 56

Figure 6.8 Performance graph on the basis of the pattern size 58

LIST OF TABLES

Table 6.1 Performance on the basis of Signature Database Size 56

Table 6.2 Performance according to the pattern size 57

CHAPTER 1
INTRODUCTION

1.1 Motivation

Today, we have become dependent on computing infrastructure, which is becoming more and more vulnerable to viruses with time. There are many reasons behind this. First is, nowadays our computing systems are connected enabling a virus to infect large number of machines by spreading. Second reason is, today machines are more programmable, so it does not need a very experienced programmer to write a malicious code etc.

Considering the devastating effect of viruses in our computing systems, it has become very important to detect them, so that we can remove them as well as disinfect the infected files. Therefore any defense mechanism should have a component that detects the presence of any kind of malicious code. There are four basic types of virus detection techniques: Signature Scanning, Integrity Checking, Activity Monitoring and Heuristic technique. Each has pros and cons of its own.

1.2 Objective

The aim is to develop a virus detection system that will help in detecting known as well as new viruses appearing daily in a large number.

1.3 Contribution

In this dissertation, a virus detection system has been implemented, that consists of two techniques, Integrity checking and Signature Scanning technique. Because no single detection method works for all cases, it has to be accomplished by a combination of detection techniques. Integrity checking technique detects presence of a virus in a file by comparing the checksum of that file with the stored checksum, when file was supposed to be clean. Thus it is able to detect any modification in the file, which may

have been caused by a known or an unknown virus. While in signature-based detection technique, a pattern string is searched for in a target text (a possibly infected program). This pattern identifies a specific virus and is called virus signature in technical terms. It consists of sequences of bytes in the machine code of the virus and is unique to a particular virus, or a family of viruses. Signature scanning technique uses a database of signature of all known viruses, which it tries to match against every file suspected to have a possible virus. This method is very good at detecting the viruses whose signatures are already known. But they are unable to detect the attack of any new virus. The main reason behind this is that the signature of this new virus is not known.

To detect viruses, this system will first use integrity checking technique, which will give all the infected files. Now to get more details about the infection, signature scanner scans those infected files. It will give the name of virus, offset where its signature is found in file etc. In case if file is declared clean by scanner, it implies some new virus has infected the file. This file can be sent to software houses so that they can perform more research on that virus. In this way this virus detection system is able to detect both known viruses as well as new viruses.

The Integrity Checking Technique uses SHA-1 algorithm to generate the checksum (secure hash) of file. It generates a 160-bit checksum and therefore provides more security than CRC-32, which generates 32-bit checksum. The probability of forgery in SHA-1 is 2^{-160}, which implies that it is very difficult to generate a different message having the same checksum.

While implementing signature detection technique, a fast pattern-matching algorithm was needed to improve its performance. Since in this technique a pattern, that can be anywhere within the file, is searched for, so if pattern-matching algorithm is not fast, users may find it annoying. Also, it is asserted that pattern matching is the most computationally expensive test that this technique commonly performs. For these reasons, in this work, a fast string-matching algorithm named Boyer-Moore-Horspool (BMH) has been used. During comparison with other algorithms like Boyer-Moore and Turbo-Boyer-Moore algorithms BMH proved the fastest pattern-matching algorithm.

Thus by using the combination of both these virus detection techniques i.e. integrity checking and signature detection technique, the virus detection system will be able to detect any kind of old or new viruses in the computer.

1.4 Organization of dissertation

The rest of the dissertation is organized as follows:

Chapter 2 gives an overview of computer viruses and their working.
Chapter 3 is well-organized exposition of anti-virus technologies.
Chapter 4 gives the specifications of Virus Detection System.
Chapter 5 moves on to explain how a signature-based detection system could benefit from the use of better string matching algorithm. As well as the operation of the popular Boyer-Moore, Turbo Boyer-Moore and Boyer-Moore-Horspool algorithms and these theoretical performances has been dissected, too. The results of some performance testing of signature detection tool have been discussed as well.
Chapter 6 explains how integrity-checking technique works and its implementation using SHA-1 algorithm for generating checkcodes of files. Also its performance is compared with CRC-32 algorithm.
Chapter 7 summarizes the results and gives a short future outlook.

CHAPTER 2
WHAT IS A COMPUTER VIRUS?

The development of a solution to any security problem initially requires understanding two areas relating the security issue: what the security threat is and what solutions have been used previously for that problem and similar security problems. This is particularly true for antivirus research also. Therefore a solution to problem of developing a method of detecting virus requires an insight in both areas. These areas are: the different types of viruses, the mechanisms that work within them and the security threats associated with them and what methodologies and technologies have been employed to combat security problems within the fields of antivirus research. This chapter discusses both these areas.

Computers are designed to execute instructions one after another. Those instructions usually do something useful — calculate values, maintain databases, and communicate with users and with other systems. Sometimes, however, the instructions executed can be damaging and malicious in nature. When that happens by accident, we call the code involved a software bug — perhaps the most common cause of unexpected program behavior. If the source of the instructions was an individual who intended that the abnormal behavior occur, then we consider this malicious coding. In recent years, occurrences of malware have been described almost uniformly by the media as *computer viruse*s.

A computer virus is a segment of machine code (typically 200-4000 bytes) that will copy itself (or a modified version of itself) into one or more larger “host” programs when it is activated. When these infected programs are run, the viral code is executed and the virus spreads further [1]. Sometimes, what constitute “programs” are more than simply applications: boot code, device drivers, and command interpreters also can be infected. Computer viruses cannot spread by infecting pure data; pure data files are not executed [2]. However, some data, such as files with spreadsheet input or text files for

editing may be interpreted by application programs. For instance, text files may contain special sequences of characters that are executed as editor commands when the file is first read into the editor. Under these circumstances, the data files are "executed" and may spread a virus. Data files may also contain "hidden" code that is executed when the file is used by an application, and this too may be infected. Technically speaking, however, pure data itself cannot be infected by a computer virus [2].

Fred Cohen [2] formally defined the term *computer virus* in 1983 as:

We define a computer 'virus' as a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows."

Although Cohen and others, including Len Adleman [14] have attempted formal definitions of *computer virus*s, none have gained widespread acceptance or use. Stubbs and Hoffman quote a definition by John Inglis that captures the generally accepted view of computer viruses:

"He defines *a virus as a piece of code with two characteristics:*

1. At least a partially automated capability to reproduce.

2. A method of transfer, which is dependent on its ability to attach itself to other computer entities (programs, disk sectors, data files, etc.) that move between these *systems."*

In other words *"Viruses are malicious segments of code, attached into legitimate programs, which execute when the legitimate program is executed"*. To attach might mean physically adding to the end of a file, inserting into the middle of a file, or simply placing a pointer to a different location on the disk somewhere where the virus can find it. The primary characteristic of a virus is that it replicates itself when it is executed and inserts the replica into another program, which will replicate the virus again when it executes.

Generally speaking [14], a computer virus consists of three parts :

- the infection mechanism,
- the trigger,
- the payload.

A computer virus must at least have the infection mechanism part.

The Infection Mechanism

As the name implies, the infection mechanism searches for one or more suitable victims and checks to avoid multiple infections, if the host is already infected or not (not every virus does this; some viruses infect a host multiple times due to bugs). After that, simply speaking, the virus body is copied into the victim. The easiest method to do so is by overwriting the code of the victim. Other methods are putting the code in front of or at the end of a file.

The Trigger

A trigger is used for starting the possible payload, i.e. on a particular event the payload is executed. Such an event could be a special day (Friday, 13th) or when the infection counter has reached a pre-defined value. Viruses also may be triggered based on some random event. One common trigger component is a counter used to determine how many additional programs the virus has succeeded in infecting. Of course, the trigger can be any combination of conditions, too.

The Payload

A possible payload causes transient or permanent damage, e.g. displaying an animation on the screen or formatting the hard disk drive or manipulation of data.

For a computer virus to work, it somehow must add itself to other executable code. The viral code is usually executed before the code of its infected host (if the host

code is ever executed again). Of course, damage may even happen unintentionally, e.g. due to a programming error. Damage may be caused by over-reaction by the user, too.

Over time, the problem of viruses has grown to significant proportions. After the first infection by the *Brain* virus in January 1986, generally accepted as the first significant MS-DOS virus [6], the number of known viruses has grown to several thousand different viruses, most of which are for MS-DOS. The problem has not been restricted to the DOS machines, however, and now affects all popular operating systems. It may be a reflection on the more technical nature of the user population of these machines.

2.1 Related Software

Worms are another form of software that is often referred to as a computer virus [16]. Unlike viruses, worms are programs that can run independently and travel from machine to machine across network connections; worms may have portions of themselves running on many different machines [19]. Worms do not change other programs, although they may carry other code that does, such as a true virus.

It is their replication behavior that leads some people to believe that worms are a form of virus, especially those people using Cohen's formal definition [2] (which incidentally would also classify standard network file transfer programs as viruses). The fact that worms do not modify existing programs is a clear distinction between viruses and worms, however.

2.2 Classification of Computer Viruses

The classification of computer viruses can be done via several ways:

- Type of infection technique,
- Type of host victim,
- Type of virus activation

- Special virus features.

2.2.1 Type of infection technique

One form of classification of computer viruses is based on the ways a virus may add itself to host code: as a shell, as an add-on, as intrusive, and as an inserting code [15].

2.2.1.1 Shell viruses

A shell virus is one that forms a "shell" around the original code. In effect, the virus becomes the program, and the original host program becomes an internal subroutine of the viral code. An extreme example of this would be a case where the virus moves the original code to a new location and takes on its identity. When the virus is finished executing, it retrieves the host program code and begins its execution. Almost all boot sector viruses are shell viruses.

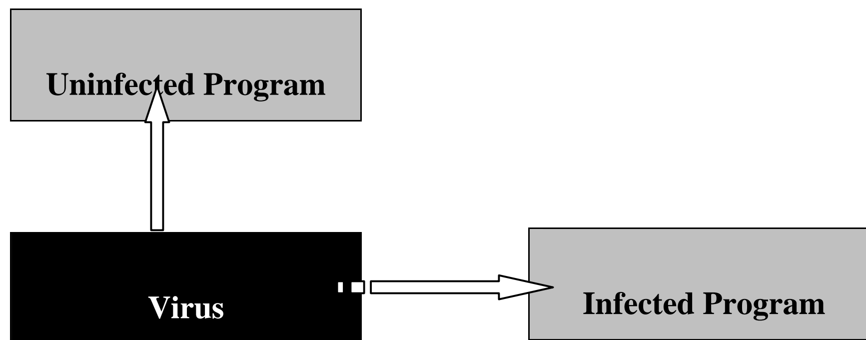

Figure 2.1 Shell Virus Infection

2.2.1.2 Add-on viruses

Most viruses are add-on viruses. They function by appending their code to the host code, and/or by relocating the host code and inserting their own code to the beginning. The add-on virus then alters the startup information of the program, executing the viral code before the code for the main program. The host code is left

almost completely untouched; the only visible indication that a virus is present is that the file grows larger, if that can indeed be noticed.

The pure **prepending virus** may simply place all of its code at the top of your original program. When you run a program infected by a prepending file virus, the virus code runs first, and then your original program runs.

Figure 2.2 Prepending virus: virus code runs first then executables

An **appending virus** places a "jump" at the beginning of the program file, moves the original beginning of the file to the end of the file, and places itself between what was originally the end of the file and what was originally at the beginning of the file. When you try to run this program, the "jump" calls the virus, and the virus runs. The virus then moves the original beginning of the file back to its normal position and then lets your program run.

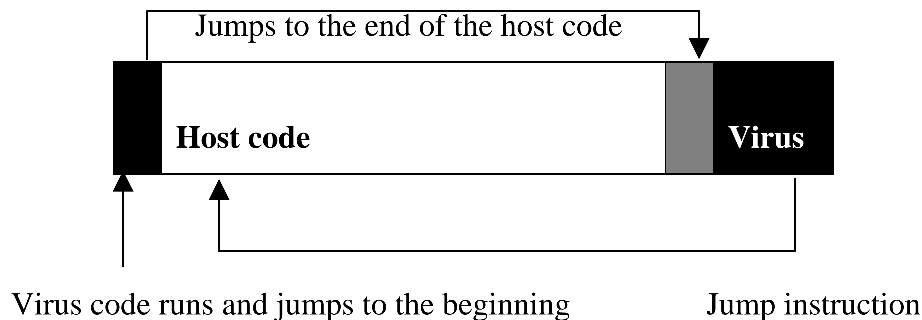

Figure 2.3 Appending virus

2.2.1.3 Intrusive viruses

Intrusive viruses operate by overwriting some or all of the original host code with viral code. The replacement might be selective, as in replacing a subroutine with

the virus, or inserting a new interrupt vector and routine. The replacement may also be extensive, as when large portions of the host program are completely replaced by the viral code. In the latter case, the original program can no longer function properly. Few viruses are intrusive viruses.

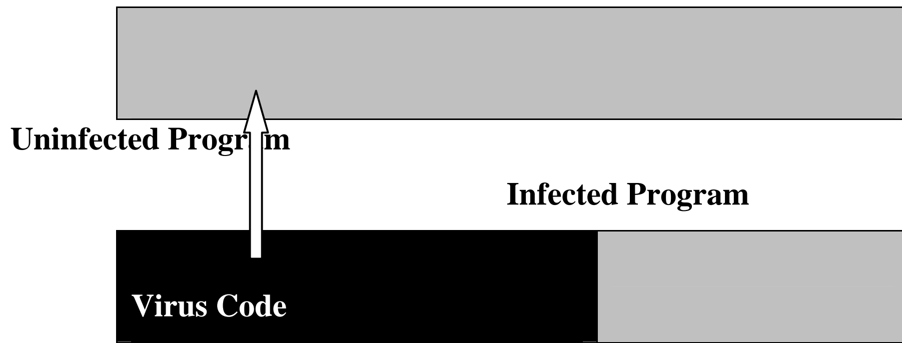

Figure 2.4 Intrusive Virus Infection

2.2.1.4 Inserting virus

An inserting virus copies itself into the host program. Programs sometimes contain areas that are not used, and viruses can find and insert themselves into such areas. The virus can also be designed to move a large chunk of the host file somewhere else and simply occupy the vacant space.

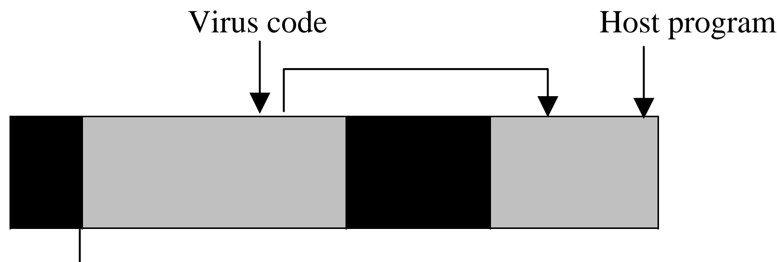

Figure 2.5 Inserting virus: Virus takes unused space and is run first. Then it jumps back to the host program

2.2.2 Type of Host Victim

A second form of classification is to divide viruses on the basis of type of Host victim. This is not particularly clear, however, as there are viruses that spread by altering system-related code that is neither boot code nor programs [14]. Some viruses target file system directories, for example. As of type of host victim we can distinguish between:

- System Sector virus,
- File Virus,
- Companion Virus,
- Multipartite Virus.

2.2.2.1 System Sector Virus

A system sector virus infects the boot sector of a floppy disc and/or partition table and boot sector of a hard disk [16]. Such a virus can infect the computer system, when the computer is booted from an infected floppy disk.

As the code in the Disk Boot Record (DBR) is started by the BIOS after it does the POST (Power On Self Test), the virus gets activated even before the Operating System has been started and most likely ”hooks” some particular Interrupts (e.g. BIOS INT13h or DOS INT 21h) for performing its tasks. Most viruses of this type save a copy of the original boot sector/master boot record in an unused sector of the disk. A boot virus may be ”placed” into the computer system by a so-called ”dropper”, i.e. a program that simply drops the boot virus.

Most boot sector viruses are memory-resident, so they can easily infect every non-write protected floppy when it is accessed. If we attept to boot another machine with this floppy, the first sector containig the virus would get loaded in memory. Now the virus acts intelligently. It knows that it has been loaded from the floppy and hence proceeds to copy itself in the first physical sector of the hard disk,that is the partition table sector. Instead of copying itself in the partition table sector, some viruses may

copy itself in the first logical sector of the DOS partiotion, that is the boot sector. In either case, before copying itself the virus would first displace the original contents of the sector to some other location. Once this is done , it reduces the RAM size and steals the interrupt. Then back again to the floppy disk to load the original boot sector in memory.

Even if the infected disk is not a bootable disk, if attempt is made to boot from the floppy, the virus still manages to enter into the machine. This is becase DOS flashes the ‘Non System Disk’ error message only when it fails to load the file IO.SYS. by thius time virus has already reached the memory and takes over the control. Thus a non bootable floppy may also infect the machine.

2.2.2.2 File Virus

The simplest file viruses’ work by locating a type of file they know how to infect (usually a file name ending in .COM or .EXE) and overwriting part of the program they are infecting. When this program is executed, the virus code executes and infects more files. These overwriting viruses do not tend to be very successful since the overwritten program rarely continues to function correctly and the virus is almost immediately discovered. Most appending viruses put their virus code at the end of the file and put a jump to the virus code at the beginning of the file, so that the virus code is started first upon execution.

2.2.2.3 Companion Virus

A companion virus looks for programs with the extension .BAT or .EXE and then creates a .COM file with the same name (i.e. EXAMPLE.COM, if a program EXAMPLE.EXE exists). If only the program name is entered, DOS per default looks up first for a matching .COM, .EXE and then .BAT file. So, EXAMPLE.COM will be started (instead of EXAMPLE.EXE, which was originally the intention of the user). Therefore, the companion virus is started first and can then start EXAMPLE.EXE.

2.2.2.4 Multipartite Virus

A multipartite or hybrid virus uses more than one infection technique, e.g. a combination of a boot sector and file virus and therefore infects DBR / MBR and files.

2.2.3 Type of virus activation

Yet a third form of classification is related to how viruses are activated and select new targets for alteration [29].

- Direct action (non-memory resident),
- Memory resident.

2.2.3.1 Direct action virus

A direct action virus is the simplest virus that runs when its "host" program is run, selects a target program to modify, and then transfers control to the host. These viruses are *transient* or *direct* viruses, known as such because they operate only for a short time, does not stay in memory and they go directly to disk to seek out programs to infect. In most cases, a direct action virus does not spread as fast as a memory resident virus.

2.2.3.2 Memory resident virus

The most "successful" PC viruses to date exploit a variety of techniques to remain resident in memory once their code has been executed and their host program has terminated. This implies that, once a single infected program has been run, the virus potentially can spread to any or all programs in the system. This spreading occurs during the entire work session (until the system is rebooted to clear the virus from memory), rather than during a small period of time when the infected program is executing viral code. These viruses are *resident* or *indirect* viruses, known as such because they stay resident in memory, and indirectly find files to infect, as the user

references those files. DOS provides a mechanism called ”terminate-and-stay-resident” (TSR), so these viruses are also known as TSR (Terminate and Stay Resident) viruses.

Only a memory resident virus may use some ”modern” virus techniques like stealth capabilities. For the memory resident virus, one can differentiate between a fast infector and slow infector. Both got their name due to the speed they spread. The first one infects every program that is being accessed (read/write) or even all files being listed in a directory listing (e.g. when the ”dir” command is being executed). The latter one, in contrast, infects only a file, when it's being written (e.g. during compilation of a new program or some older programs stored their configuration settings directly into the executable file). Therefore, a slow infector may bypass file integrity checkers.

If a virus is present in memory after an application exits, how does it remain active? That is, how does the virus continue to infect other programs? The answer for personal computers running software such as MS-DOS is that the virus alters the standard interrupts used by DOS and the BIOS (Basic Input/Output System). When an interrupt is raised, the operating system calls the routine whose address it finds in a special table known as the *vector* or *interrupt* table. Normally, this table contains pointers to handler routines in the ROM or in memory-resident portions of the DOS. A virus can modify this table so that the interrupt causes viral code (resident in memory) to be executed. Once a virus has infected a program or boot record, it seeks to spread itself to other programs, and eventually to other systems.

2.2.4 Special Virus Features

The following special virus features will be explained briefly:

- Stealth technique,
- Polymorphism.

2.2.4.1 Stealth Virus

Some special virus features can only be used by memory-resident viruses. A stealth virus tries to hide itself by hooking several interrupts like BIOS Int 13h or DOS

Int 21h. Assumed, an anti-virus program reads the MBR via BIOS Int 13h to scan for viruses, the virus can intercept this and ”redirect” the read call to the saved copy of the original, uninfected MBR. Therefore, the anti-virus program won’t find any virus. Or, if a virus scanner scans a file, this file must be opened first [4]. The open call, ”redefined” by the virus, will first remove the virus from the file and then call the original open call. After the scanning of the file has been finished, the file will be closed by the virus scanner. And the modified close call will infect the file again.

2.2.4.2 Polymorphic Virus

A polymorphic virus is being ”encrypted” and changes infection its shape and structure of the encryption/decryption routine by each infection but the basic functionality is always the same [4]. For example: a CPU has a set of registers, e.g. the accumulator register AX. If this register is set to be zero, then this can be done by setting the register to zero, i.e. MOV AX, 0. Or by subtracting the current value of the AX register with itself, i.e. SUB AX, AX. Or by the exclusive-or operation, i.e. XOR AX, AX. In short, the effect is just the same, but each operation will result in a different opcode. This technique is also known as ”mutation”.

Computer viruses can infect any form of writable storage, including hard disk, floppy disk, etc [16]. Infections can spread when a computer is booted from an infected disk, or when an infected program is run. This can occur either as the direct result of a user invoking an infected program, or indirectly through the system executing the code as part of the system boot sequence or a background administration task. With the presence of networks, viruses can also spread from machine to machine as executable code containing viruses is shared between machines. Once activated, a virus may replicate into only one program at a time, it may infect some randomly chosen set of programs, or it may infect every program on the system. Sometimes a virus will replicate based on some random event or on the current value of the clock [15].

Traditional boot viruses and file viruses prosper in MS-DOS machines because MS-DOS has no inherent security features. Viruses, therefore, have free rein to infect system sectors and program Files. In starting, most infections were seen in .COM files or system sectors, but later, the application world switched to .EXE format executables [10]. Infecting .EXE files is done in any number of ways, from prepending, or appending code to a file, to splitting up the virus and hiding it in holes within the unused segments of the host application. These viruses are written mostly at an assembly language level to have access to the innermost workings of DOS.

CHAPTER 3
ANTIVIRUS SOLUTIONS

Computer viruses pose an increasing risk to computer data integrity. They cause loss of valuable data and cost an enormous amount in wasted effort in restoration/duplication of lost and damaged data. Each month many new viruses are reported. As the problem of viruses increases, we need tools to detect them and to eradicate them from our systems. Defense against viruses generally takes one of four forms:

3.1 Signature scanner

Scanners have been the most popular and widespread form of virus defense. A scanner operates by reading data from disk and applying pattern matching operations against a list of known virus patterns. If a match is found for a pattern, a virus instance is announced.

Scanners are fast and easy to use, but the list of patterns must be kept up-to-date. In the MS-DOS world, new viruses are appearing by as many as several dozen each week. Keeping a pattern file up-to-date in this rapidly changing environment is very important.

To the advantage of scanners, however, is their speed. Scanning can be made to work quite quickly. Scanning can also be done portably and across platforms, and pattern files are easy to distribute and update. Furthermore, of the new viruses discovered each week, few will ever become widespread. Thus, somewhat out-of-date pattern files are still adequate for most environments. Scanners equipped with

algorithmic or heuristic checking may also find most polymorphic viruses. It is for these reasons that scanners are the most widely used form of anti-virus software.

A further benefit to scanning is that it can also be used against embedded trojan horse code, logic bombs, and other malicious software in addition to simple viruses. All that is needed to detect these pieces of code are appropriate signatures generated from a disassembly of virus code. These signatures can be added to the search set and used without any further change to the scanning software.

Cohen argues that signature scanning is not worth pursuing against computer viruses [11]. He (correctly) observes that scanning cannot find new viruses before their patterns are known, nor will such methods work against polymorphic viruses. He attempts to show that an integrity shell (i.e., checksumming) is the most cost-effective approach to virus protection.

It is believed that the cost-benefit ratio for scanners, either by themselves or in addition to other mechanisms, is much higher than he calculates [14]. This is because of scanners low impact on existing practice and because of their flexibility. Their widespread use and continued effectiveness in the commercial world affirm this view. Almost all currently available commercial anti-virus tools use signature scanning as their primary detection method.

3.2 Integrity checkers

Integrity checkers are programs that generate checkcodes (e.g., checksums, cyclic redundancy codes (CRCs), secure hashes, message digests, or crypto-graphic checksums) for monitored files [21]. Periodically, these checkcodes are recomputed and compared against the saved versions. If the comparison fails, a change is known to have occurred to the file, and it is flagged for further investigation. Integrity monitors run continuously and check the integrity of files on a regular basis. Integrity shells recheck the checkcode prior to every execution.

Integrity checking is an almost certain way to discover alterations to files, including data files [11]. As viruses must alter files to implant themselves, integrity checking will find those changes. Furthermore, it does not matter if the virus is known or not, the integrity check will discover the change, no matter what caused it. Integrity checking also may find other changes caused by buggy software, problems in hardware, and operator error.

Integrity checking also has drawbacks [21]. On some systems, executable files change whenever the user runs the file, or when a new set of preferences is recorded. Repeated false positive reports may lead the user to ignore future reports, or disable the utility. It is also the case that a change may not be noticed until after an altered file has been run and a virus spread. More importantly, the initial calculation of the checkcode must be performed on a known-unaltered version of each file. Otherwise, the monitor will never report the presence of a virus; probably leading the user to believe the system is uninfected. Some integrity checkers now use other anti-virus techniques along with their intelligence and ease of use.

While using an integrity checker is an excellent way to monitor changes to in system, with today's operating systems so many files change on a regular basis, it's imperative integrity checking to improve that we also use a good up-to-date scanner along with the integrity checker or for the integrity checker to have that capability built in.

3.3 Activity monitor

Activity monitors are programs that are resident on the system. They monitor activity, and either raise a warning or take special action in the event of suspicious activity. Thus, attempts to alter the interrupt tables in memory, or to rewrite the boot sector would be intercepted by such monitors. This form of defense can be circumvented (if implemented in software) by viruses, which activate earlier in the boot sequence than the monitor code [11]. They are further vulnerable to virus alteration if

used on machines without hardware memory protection —as is the case with all common personal computers.

Another form of monitor is one that emulates or otherwise traces execution of a suspect application. The monitor evaluates the actions taken by the code, and determines if any of the activity is similar to what a virus would undertake. Appropriate warnings are issued if suspicious activity is identified.

A monitoring program assumes that viruses perform actions that are in its model of suspicious behavior and in a way that it can detect. These are not always valid assumptions. New viruses may utilize new methods, which may fall outside of the model [20]. The monitoring program would not detect such a virus.

The techniques used by monitoring tools to detect virus-like behavior are also not foolproof [22]. Personal computers lack memory protection, so a program can usually circumvent any control feature of the operating system. As a part of the operating system, monitoring programs are vulnerable to this as well. There are some viruses, which evade or turn off monitoring programs.

Finally, legitimate programs may perform actions that the monitor deems suspicious (e.g., self-modifying programs). Monitoring software may be difficult to use but may detect some new viruses that scanning does not detect, especially if they do not use new techniques. These monitors produce a high rate of false positives. Monitors can also produce false negatives if the virus does not perform any activities the monitor deems suspicious. Worse yet, some viruses have succeeded in attacking monitored systems by turning off the monitors themselves.

Monitoring packages are integrated with the operating system so that additional security procedures are performed. This implies some amount of overhead when any program is executed. The overhead is usually minimal, though.

3.4 Heuristics Scanner

Heuristic techniques are used to find unknown viruses and threats that have not yet been cataloged with signatures. Heuristics looks at characteristics of a file, such as size or architecture, as well as behaviors of its code to determine the likelihood of an infection [20]. Heuristics can sometimes find and stop many new viruses before they execute. It is also used to find known viruses that do not lend themselves to signatures, like some of the new metamorphic viruses that can obscure their entry points, have obfuscated code structures (that can shrink or expand themselves through their metamorphic engines), and are often encrypted as well.

A heuristic scanner looks for dozens if not hundreds of behaviors and indicators that viruses use, and assigns a weighted score to each. Some red flags may include code to check a date, an oversized file, or attempts to access your address book. Any of these red flag indicators could simply be an innocent application, but a scorecard of sorts is created, and when a number of such occurrences are combined, it may indicate a virus.

Heuristics can also be used in a reverse manner, looking for behaviors that couldn't possibly be viruses, or in certain instances couldn't possibly be specific viruses with known properties. Therefore, in some cases, it's faster to determine that a file couldn't contain a virus, then finding one that could. For example, with a virus, by which the infection is 32k to 130k in length, then a file that is 25k can not contain that virus.

A big plus for heuristics is the ability to detect viruses in files and boot records before they have a chance to run and infect the machine. Other anti-virus technologies, such as behavior blocking or integrity checking, actually require the virus to execute on the host computer and exhibit suspicious and potentially harmful behavior before the virus can be detected and stopped.

Heuristic techniques, on the other hand, are working on the probabilities of a file being infected. Heuristics is not an exact science. Currently, the industry claims a 70%-80% detection rate of new and unknown viruses with heuristic scanning, which clearly demonstrates the complexity of virus detection problem.

As given here there are several methods of defense against viruses. Unfortunately, no single defense mechanism is perfect, but the combination of integrity checking with signature matching technique seems the more reliable and more economical approach to detect viruses.

CHAPTER 4
SPECIFICATIONS OF VIRUS DETECTION SYSTEM

In previous chapter, the definitions of various antivirus solutions are given, including: signature scanning, integrity checking, behavior monitoring and heuristic scanning method. The advantages and disadvantages of these individual methods are also presented. Which is better: specific, precise detection or generic way of handling viruses? A theoretical battle is constantly going on.

There are pros and cons of both approaches:

- Specific methods are good at determining exact nature of infection and can serve as an excellent base for producing detailed descriptions and discussing small differences between minor variations.
- Generic methods are good at detecting new viruses but they are not good at determining the type of infections.

So the best way to detect known as well as unknown viruses is to combine both these approaches. However, there are many different ways to combine two approaches, but the combination of integrity checking with signature matching technique seems the more reliable and more economical approach to detect viruses. This virus detection system uses the similar approach.

4.1 Generic Approach: An Integrity Checker

Integrity checkers are the most reliable tool for unknown viruses [27]. It is a program that determines whether another program has been altered or changed. It searches for such changes and flags them as suspicious.

The integrity checker tool used in this detection system is an on-demand tool and uses SHA-1 algorithm to generate a cryptographic checkcode for verifying the

integrity of information in computer systems. This technique produces a 160-bit checksum. The chances of having two random documents hash to the same value is very small, which is 1 in 2^{160} , Which implies forgery is almost impossible.

4.2 Specific Approach: A Signature Scanner

Scanner speed is as important as its ability to detect viruses. Slow scanners simply cannot be used effectively; they increase the computing cost and should not be used.

The first and probably the most important step to enhance the speed of scanner is to use a fast pattern-matching algorithm because searching for signatures in files is its main task. For this purpose, the signature-scanning tool uses Boyer-Moore-Horspool algorithm, which significantly improves the performance.

The second important thing is to ensure that the virus definitions are applied to the right type of files. It does not make sense to look for a virus infecting only boot sectors in EXE files. To achieve this, the virus detection database has been designed properly. The viruses have been categorized in three types: boot sector, partition table and file types.

The third step to achieve high data processing rate is to read as little from the file as possible before staring to analyze it seriously. The scanner divides file in slices of 1024 bytes and reads one slice at a time. Optimally, if the scanner can read the first chunk of the file and determine whether file is clean or not, that would be exceptionally quick. It is not usually possible to decide if the file is clean after analyzing just the first chunk but certainly the less disk I/O tat is performed during the analysis, the better. The reason for this is obvious; disk operations are relatively slow and amount to approximately 50% of the total time spent performing a scan.

4.3 Combining Generic and Virus-Specific Approaches

When a scanner is analyzing the file, it should make sure that before it starts scanning, it performs some kind of elimination. Good elimination is important but achieving this is sometimes tricky. However, the integrity checking, a generic virus detection technique can be used for this purpose. Integrity checker checks all the files for the modifications and generates a report of all the modified files. Now only these files need to be checked by signature scanner. That would be a good elimination. Generally people prefer to know which particular virus they have detected. So, after finding a virus-infected file by using integrity checker it would be advantageous to switch to signature scanner and report the exact details. This virus detection system uses the same approach, which is very similar to specific detection using elimination by generic method approach.

CHAPTER 5
INTEGRITY CHECKER

Integrity Checking is a generic technique known by several names, particularly Modification Detection, Differential Detection, and (Message or File) Authentication. This is the most reliable detection technique for unknown viruses [27]. An integrity checker (also known as a checksummer) is a program that determines whether another program has been altered or changed. It searches for such changes and flags them as suspicious.

The basic principles behind this technique [21] are as follows:

1. The distinguishing feature of a virus is that it *replicates*.
2. In order to replicate, a virus must attach itself to existing executable files1, which necessarily causes some modification in the file.
3. Almost all viruses postpone their damaging effects for a long time (typically months) in order to give themselves a chance to propagate as widely as possible without being noticed.
4. Therefore to detect all infections, it suffices to examine all such files, either periodically or just before execution, to determine whether they have been modified. If a file has not been modified, one can be certain that it is not infected (assuming it was not infected to begin with).

The advantages of this technique [21] are that:

1. it seems capable of detecting *all* viral infections which occur after its installation, even by viruses which have never been encountered before;
2. it cannot be fooled, neutralized, or circumvented by stealth viruses or other hostile software;
3. it is not affected by polymorphic techniques or compression;

4. there is no need for the user to obtain updates when new viruses or even entirely new types of viruses are discovered.

Though an integrity checking detects infection only after it has taken place. Nevertheless, the fact that it detects all infections is very important. If the user has backed up his files before they became infected (a procedure which is recommended in any case), he can restore them from backups. If the virus is known, it can be eradicated and in the case of unknown virus, one can send the infected file to any software house, to get this virus registered.

5.1 Checksumming

The almost universal way of implementing integrity checking is to compute, for each file, a small fixed-sized value, at a time when system is assumed to be “clean” (i.e. uninfected). Then modifications can be detected by changes in these values and it suffices to store only these small original values (instead of copies of the entire original files) for comparison.

The most common names for this value and the process of computing it are **checksum** and **checksumming**, respectively [21]. A few write “checkcode” or “checkvalue”, but these terms are not in wide use. A much more common alternative, especially in cryptological circles, is **hash value** or sometimes **hashcode**, in which case the checksumming function is called a **hash** function. Other alternatives for “checksum” are **message digest**, **fingerprint**, **file signature**, and **MIC (Message Integrity Code)**.

Instead of computing checksums and looking for changes in them in order to detect modifications in files, there are few programs, which attempt to save time by looking for changes in the size (length) of the file and/or in its date/time stamp. The argument concerning the file-length check is as follows: When a program is infected with a virus, it must continue to perform all its usual functions, so that the user does not notice the modification. Moreover, new functions associated with the virus (mainly

replicating itself and performing some destructive action at some future date) must be added. The result is that these functions must increase the size of the program. This argument overlooks several things. First of all, checking the size is not effective against boot sector or master-boot-record viruses. Secondly, even among file viruses, it is ineffective against:

(a) stealth and semi-stealth viruses if the virus is in memory when checksumming is performed and the checksumming program is not capable of bypassing the stealth mechanism,
(b) viruses which hide the increase behind the end-of-file mark,
(c) "cavity" viruses, i.e. those which place themselves only in files which contain enough consecutive unused binary-zero characters so that they can replace these characters, and
(d) viruses which compress the original program and then pad it to preserve the original length.

On the other hand, a file-length check can be quite useful if used, not instead of, but with checksumming, file creation date as well as last modified date. For if any of these has got changed, this guarantees that the file itself has changed.

5.2 Implementation Modes

There are essentially three ways of implementing checksumming:

5.2.1 On-demand

By a non-resident program activated by explicit program call, in order to check all files or a specified subset of them all at once.

5.2.2 Resident

By a memory-resident program to automatically checksum, just before execution, each program which is about to be executed.

5.2.3 Self-test

By code attached to each program, which is executed just before the program itself is executed.

The first two methods usually use a single table to hold the name of every file, which is to be checksummed and its corresponding checksum, although some implementations create a separate table for each directory. Such table is called the **CST** (CheckSum Table) and additional information on each file, such as the size, creation date and time, is also included. With the later two methods one hardly notices the extra execution time required for the checksum comparison, and because infrequently executed programs don't get checksummed any more than is necessary. Other things being equal, these considerations would be significant. However, other things rarely are equal, and both the resident and the self-test implementations have disadvantages. These methods can be applied only to ordinary files, not to boot records or to certain system files [27]. Also, a malicious program can easily neutralize the resident and self-test types of programs.

Additional disadvantages of the self-test implementation are:

(a) a program may already have its own internal self-test, in which case the addition of the new code to the file will cause the internal self-test to sound an alarm,
(b) the self-test code may get added to an already infected file, making it non-disinfectable by many virus removal programs,
(c) adding code to many executable files wastes much disk space,
(d) the extra code may be misinterpreted as a virus by many users, and
(e) many users simply don't like having their programs altered. For these reasons, use of this implementation is to be discouraged.

Perhaps most important, if either of the last two methods is used alone, there is no way of guaranteeing that memory is "clean" (uninfected) when the checksumming is performed. They are therefore at the mercy of stealth viruses; when such a virus is in

memory, any attempt to checksum an infected file will result in checksumming the original uninfected file, and the infection will go unnoticed. Another danger is that if the checksum program is activated while RAM is infected by a “fast infector”, then every time a file is opened for checksumming, it will immediately be infected by that virus.

On-demand checksummers, on the other hand, have the advantage that they can be executed when memory is known to be clean. This is achieved by activating them immediately after cold booting from a system diskette, which is known to be clean because its write protection has never been removed since it was created. Theoretically, even an original DOS diskette could be infected, but this is highly unlikely if the diskette has been kept write protected.

5.3 Criteria for Choice of Hash Function

Three obvious criteria, which a hash function must satisfy [12], are:

1. For any given file F, computation of H (F) should be fast.
2. The length of the checksums should not be too great, so as not to take up too much storage.
3. If two files are chosen at random, the probability of their having the same checksum should be very small.

The fastest algorithms can compute checksum and compare it with the stored value more quickly than a comparison of the complete file with a stored copy of it can be made. There is a probability, that two randomly chosen files have the same checksum. It is therefore important that the checksum should be reasonably large. Increasing the length of checksum will increase storage requirements somewhat and perhaps also the computation time. Therefore, unless storage and time are not at a premium, checksum is usually chosen large enough to give security in the situation in which the checksum program is used.

This is an on-demand integrity cheker tool and uses SHA-1 algorithm to generate a cryptographic checkcode for verifying the integrity of information in computer systems with no built-in protection. This technique produces a 160-bit condensed representation of the message called a message digest.

5.4 Description Of SHA-1

SHA (The Secure Hash Algorithm) is a cryptographic message digest algorithm specified in the Secure Hash Standard (SHS, FIPS 180), and was developed by NIST [30]. SHA-1 is a revision to SHA that was published in 1994. During the revision an unpublished flaw present in SHA was corrected [33]. Later, after the selection of Rijndael as the Advanced Encryption Standard, were announced new algorithms SHA-1, SHA-256, SHA-384 and SHA-512.

These new algorithms were published as "Secure Hash Standard" (in FIPS PUB 180-2) are issued by National Institute of Standards and Technology, announced on 2002 Aug 1.

This new standard specifies four secure hash algorithms – SHA-1, SHA-256, SHA-384 and SHA-512 – for computing a condensed representation of electronic data (message) [31]. When a message of any length less than 2^{64} bits (for SHA-1 and SHA-256) or 2^{128} bits (for SHA-384 and SHA-512) is input to an algorithm, the result is an output called message digest. The message digest ranges from 160 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits). The four hash algorithms described in this standard called secure because, for a given algorithm, it is computationally infeasible to find a message that corresponds to the given message digest, or to find two different messages that produce the same message digest.

The Secure Hash Algorithm (SHA-1) is required for use with the Digital Signature Algorithm (DSA) as specified in the Digital Signature Standard (DSS) and whenever a secure hash algorithm is required for federal applications. For a message of length < 2^{64} bits, the SHA-1 produces a 160-bit condensed representation of the message called a message digest. The message digest is used during generation of a signature for the message. The SHA-1 is also used to compute a message digest for the received version of the message during the process of verifying the signature. Any change to the message will, with very high probability, result in a different message digest, and the signature will fail to verify.

The SHA-1 is designed to have the property that it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest [38].

5.4.1 Bit Strings and Integers

The following terminology related to bit strings and integers will be used:

a. A hex digit is an element of the set {0, 1, 9, A, ..., F}. A hex digit is the representation of a 4-bit string.

Examples: 7 = 0111, A = 1010.

b. A word equals a 32-bit string, which may be represented as a sequence of 8 hex digits. To convert a word to 8 hex digits each 4-bit string is converted to its hex equivalent as described in (a) above.

Example: 1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

c. An integer between 0 and 2^{32} - 1 inclusive may be represented as a word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation.

Example: the integer 291 = $2^8+2^5+2^1+2^0$ = 256+32+2+1 is represented by the hex word, 00000123.

If z is an integer, $0 <= z < 2^{64}$, then $z = 2^{32}x + y$ where $0 <= x < 2^{32}$ and $0 <= y < 2^{32}$. Since x and y can be represented as words X and Y, respectively, z can be represented as the pair of words (X, Y).

d. Block = 512-bit string. A block (e.g., B) may be represented as a sequence of 16 words.

5.4.2 Operations on WORDS

The following logical operators will be applied to words:

a. Bitwise logical word operations

X Λ Y = bitwise logical "and" of X and Y.

X \/ Y = bitwise logical "inclusive-or" of X and Y.

X **XOR** Y = bitwise logical "exclusive-or" of X and Y.

~ X = bitwise logical "complement" of X.

Example:

```
     01101100101110011101001001111011
XOR  01100101110000010110100110110111
     --------------------------------
  =  00001001011110001011101111001100
```

b. The operation X + Y is defined as follows: words X and Y represent integers x and y, where $0 <= x < 2^{32}$ and $0 <= y < 2^{32}$. For positive integers n and m, let n mod m be the remainder upon dividing n by m. Compute $z = (x + y) \bmod 2^{32}$. Then $0 <= z < 2^{32}$. Convert z to a word, Z, and define Z = X + Y.

c. The circular left shift operation $S^n(X)$, where X is a word and n is an integer with $0 <= n\ ^{32}$, is defined by $S^n(X) = (X << n)$ OR $(X >> 32-n)$. Here, $X << n$ is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the

right (the result will still be 32 bits). X >> n is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus $S^n(X)$ is equivalent to a circular shift of X by n positions to the left.

5.4.3 Message Padding

The SHA-1 is used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. The purpose of message padding is to make the total length of a padded message a multiple of 512. The SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The following specifies how this padding shall be performed. As a summary, a "1" followed by m "0"s followed by a 64-bit integer are appended to the end of the message to produce a padded message of length 512 * n. The 64-bit integer is l, the length of the original message. The padded message is then processed by the SHA-1 as n 512-bit blocks.

Suppose a message has length $l < 2^{64}$. Before it is input to the SHA-1, the message is padded on the right as follows:

a. "1" is appended.

Example: if the original message is "01010000", this is padded to "010100001".

b. "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length l of the original message.

Example: Suppose the original message is the bit string

01100001 01100010 01100011 01100100 01100101.

After step (a) this gives

01100001 01100010 01100011 01100100 011001011.

Since l = 40, the number of bits in the above is 41 and 407 "0"s are appended, making the total now 448. This gives (in hex)

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000.

c. Obtain the 2-word representation of l, the number of bits in the original message. If l < 2^{32} then the first word is all zeroes. Append these two words to the padded message.

Example: Suppose the original message is as in (b). Then l = 40 (note that l is computed before any padding). The two-word representation of 40 is hex 00000000 00000028. Hence the final padded message is hex

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000028

The padded message will contain 16 * n words for some n > 0. The padded message is regarded as a sequence of n blocks M_1, M_2, ..., M_n, where each M_i contains 16 words and M_1 contains the first characters (or bits) of the message.

5.4.4 Functions Used

A sequence of logical functions f_0, f_1,..., f_{79} is used in the SHA-1. Each f_t, 0 <= t <= 79, operates on three 32-bit words B, C, D and produces a 32-bit word as output. f_t(B,C,D) is defined as follows: for words B, C, D,

f_t(B,C,D) = (B AND C) OR ((NOT B) AND D) (0 <= t <= 19)

f_t(B,C,D) = B XOR C XOR D (20 <= t <= 39)

f_t(B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t <= 59)

f_t(B,C,D) = B XOR C XOR D (60 <= t <= 79).

5.4.5 Constants Used

A sequence of constant words K (0), K (1), ... , K (79) is used in the SHA-1. In hex these are given by

K = 5A827999 (0 <= t <= 19)

K_t = 6ED9EBA1 (20 <= t <= 39)

K_t = 8F1BBCDC (40 <= t <= 59)

K_t = CA62C1D6 (60 <= t <= 79).

5.4.6 Computing the Message Digest

The message digest is computed using the final padded message. The computation uses two buffers, each consisting of five 32-bit words, and a sequence of eighty 32-bit words. The words of the first 5-word buffer are labeled A, B, C, D, and E. The words of the second 5-word buffer are labeled H_0, H_1, H_2, H_3, and H4. The words of the 80-word sequence are labeled W_0, W_1,..., W_{79}. A single word buffer TEMP is also employed.

To generate the message digest, the 16-word blocks M_1, M_2,..., M_n defined in Section 4 are processed in order. The processing of each M_i involves 80 steps. Before processing any blocks, the $\{H_i\}$ are initialized as follows: in hex,

H_0 = 67452301

H_1 = EFCDAB89

H_2 = 98BADCFE

H_3 = 10325476

H_4 = C3D2E1F0.

Now M_1, M_2, ... , M_n are processed. To process M_i, we proceed as follows:

a. Divide M_i into 16 words W_0, W_1, ... , W_{15}, where W_0 is the left-most word.

b. For t = 16 to 79 let $W_t = S^1 (W_{t-3} \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16})$.

c. Let A = H_0, B = H_1, C = H_2, D = H_3, E = H_4.

d. For $t = 0$ to 79 do

$TEMP = S^5 (A) + f_t(B,C,D) + E + W_t + K_t$;

$E = D; D = C; C = S^{30} (B); B = A; A = TEMP$;

e. Let $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$.

After processing M_n, the message digest is the 160-bit string represented by the 5 words H_0 H_1 H_2 H_3 H_4.

SHA-1 is a cryptographically strong checksumming technique with reasonable performance characteristics, and it performs as intended. It is a method by which performance may be improved over previous similar systems without substantially increasing the complexity [35]. It appears to be genraly applicable and well suited for integrity checking applications, allowing it to verify its own integrity and the integrity of data files as well.

5.5 Description of Integrity Checker Tool

This tool consists of two parts. The first part uses SHA-1 algorithm to generate the hash-codes i.e. message digests for files and then develops a catalog of message digests for all the files in a directory tree. Since this is an on-demand integrity checker tool, so it should be executed, only when memory is known to be clean. Later on the second part can check the files in the same directory tree against the catalog.

5.5.1 Implementation

CHECKER is written as a dual-purpose program. It has two operating modes, one for building a list of message digests, and another for checking files against that list. The command line for CHECKER has one of two forms. The first form is used to create a Message Digest listing file that has the message digest called checksum in this case, and file name of every file in a directory tree, time, date and file size. The syntax for invoking CHECKER in this mode is:

CHECKER -c directory checksum-file-name

The directory parameter passed to CHECKER is the name a of a root directory. CHECKER will calculate the checksum for every file in and under that directory, and store the results in the checksum file named as the second parameter. The checksum file created is an ordinary ASCII text file that can be edited and manipulated using any text editor. All it contains is a sequence of lines that contain a checksum value followed by a file name, date, time and its size.

For example, CHECKER was run on the F: directory that holds all the work for this dissertation on MS-DOS machine with the following command:

CHECKER –c F: TEST.txt

This created a checksum file named TEST.txt, which had the following contents:

fbe3e486a4238eda55b9da0ceedc14a6b0152056 F:\INTEGR~1\CHECKER.C 0 57 8 4 20 2005 58392578

d1b2a0dd3ffa34c396aebada1e6bef2a59473467 F:\INTEGR~1\SHA.H 16 5 46 9 10 2003 58392578

0e32540de0106bc120563ce13e72653513088458 F:\INTEGR~1\GLOBAL.H 16 5 38 9 10 2003 58392578

-

-

-

62f0c7a08aad4016a76db98dd8802a54e056cd88 F:\ SIGNAT~1\TESTFILE.C 11 19 8 4 1 2005 58392578

ad77c85a619bb1e4076bb7ea7e3873d2c8028aa1F:\SIGNAT~1\SCANNER.C 14 0 34 4 1 2005 58392578

Later on, one can check the integrity of these files by running CHECKER in its second mode, which takes this command line:

CHECKER TEST.txt

In this mode, CHECKER just reads in each line of the Checksum file, calculates the checksum of the file, and determines if it matches the stored checksum. Now if a file is modified, CHECKER will produce the error that checksum, last modified time as well as file size differs.

CHECKER correctly detected the changes in the file. In the current implementation of CHECKER, all that happens when an error is detected is that an error message is printed out to the screen at the same time. At the end a summary is also printed which shows all the modified files in case of any modifications, otherwise it prints, “All files are intact”.

5.5.2 The Code

The complete listing for CHECKER is shown in appendix B. This tool is designed to run under most MS-DOS C Compilers. The main () routine of CHECKER has to first perform checks to see which mode the user has selected, based solely on the number of arguments passed on the command line. If argc is equal to 2, main () assumes that it has been invoked with a single file name as an argument, and it calls CheckFiles(). If argc is equal to 4 and the first argument is "-c", main () assumes it has been invoked to build a checksum file, and then it calls BuildChecksumFile(). If neither of these turns out to be true, a usage message is printed out and the program exits.

5.5.2.1 Building the Checksum File

Of the two possible jobs given to this program, building the Checksum file is the more complex. Both tasks have to calculate the checksum values for one or more files, but building the file has the additional job of navigating through the directory tree.

BuildChecksumFile() sets things up for the task by opening up the output file that is going to receive all the file names and Checksum values. It then makes a call to the routine, recusfile(), that gives all the filenames. This routine takes two arguments, a path name and a checksum file FILE pointer.

The pseudo code for this routine looks like this:

```
recusfile( path )
    dir = OpenDirectory( path )
    while FilesLeftInDirectory( dir )
      filename = GetNextFile( dir )
       if filename is a directory then
```

```
                recusfile( filename )
                else
                ProcessFile( filename )
                endif
          end of while
    end of recusfile
```

Implementing this same function without being able to use recursion would be considerably more difficult. Examining the body of recusfile() shows that near the bottom of the routine a call is made to ProcessFile(). This routine then calls CalculateChecksumFile(), which calculates the Checksum value for the file, using SHA-1 algorithm. The result is then printed out along with the file name, date, time and size to the Checksum log file. Thus by these routines, a complete listing of the entire directory tree is built up, for later use by CHECKER in its checking mode.

5.5.2.2 Calculating the File Checksum

For calculating the Checksum of a given file, SHA-1 algorithm has been used. The CalculateFileChecksum() routine repeatedly reads in blocks of 512 bytes, and passes them to the SHAUpdate() routine. Then routine SHAFinal() is used to give the checksum for the file so far. This process repeats until the entire file has been processed.

5.5.2.3 Checking the Files

The second mode of operation for this program is the Checksum check. Most of the work here is done in the CheckFiles() routine. It gets to bypass the directory tree navigation, since all of the file names it needs to check are already stored in the Checksum log file. All this routine has to do is repeatedly read in a line from the Checksum log file containing a 160-bit checksum value, a file name, date and time of creation, and file size. It then calculates the actual Checksum for the file, and reports on whether the stored and calculated Checksum values match up.

CHECKER can be set up to provide a quick way to check the integrity of any or all of the files on your system. By calling CHECKER with the -c parameter for every directory full of executables, a set of Checksum log files is created that can be periodically checked with a single call to CHECKER. CHECKER operates quickly enough that we can even include it as part of AUTOEXEC.BAT file under MS-DOS, without letting it slow down the work too much.

5.6 Comparison of SHA-1 with CRC-32

CRC-32 generates a 32-bit checksum, while SHA-1 generates 160-bit checksum. The SHA-1 does have attributes that make it very attractive for the verification of files. These include the following:

- Every bit in the message contributes to the message digest. This means that changing any bit in the message should change message digest also.
- Relatively small changes in the message should always result in changes in the message digest. We want to be sure that it would take an extremely unlikely combination of errors to produce an identical message digest.
- The histogram of output message digests for input messages should tend to be flat. For a given input message, we want the probability of a given message digest being produced to be nearly equal across the entire range of possible message digests from 0 to FFFFFFFFH.

Gilmore Systems has a program called PROVECRC that creates a modified version of a file that is different, but that has the same CRC as the original. The program proves that a CRC is not fool-proof for virus detection, for it is possible to write a virus, much like they wrote PROVECRC, which can add code to the program without changing the CRC. When CRC-32 and SHA-1 algorithms were used, PROVECRC created changes undetected by CRC-32, but detected by SHA-1.

SHA-1 gives a larger checksum that makes it more resistant to brute force attacks, such as Birthday attacks, which choose messages at random in an attempt to

generate the same checksum. It is known that CRCs are not cryptographically strong. It fails to0 provide the required integrity protection and not intended to be used in place of SHA-1. CRCs will not protect against intentional damage, because it is fairly easy to fiddle the file to make the checksum come out the same, which is very difficult with SHA-1. In case of SHA-1, the chances of having two random documents hash to the same value is very small, which is 1 in 2^{160}, while in the case of CRC, it is 1 in 2^{32}. This means that while CRC-32 will be an excellent judge of unintentional damage to files, it is possible that an exceptionally clever virus will be able to defeat it.

Though CRC-32 is fast, but not secure. So where security is more important, a slower, but really secure solution is better than an insecure though fast solution.

CHAPTER 6
SIGNATURE SCANNING TECHNIQUE

Among all the methods of virus detection mentioned above, the method that can detect viruses accurately and also can help in removing them is Signature Scanning method. Having every virus signature till date in the database, it is easy to detect majority of viruses. While signature scanning may not be able to detect all possible viruses, it is still simple and cheap enough to be easily available and useful to the public at large, and it has the least impact on existing code and hardware. Moreover, it is simple to add new patterns to an existing scanner whenever new viruses are discovered.

This chapter analyzes the problem of virus detection using *Signature Scanning Technique* and its reliance on fast string matching algorithms. It will show that the problem can be restructured to allow the use of more efficient string matching algorithms that operate on patterns i.e. virus signatures and will introduce and analyze Boyer-Moore-Horspool algorithm, a fast string-matching algorithm. Further, measurement of the actual performance of several search algorithms on a set of virus signatures is given. The results provide lessons on the structuring of string matching algorithms in general, and the importance of performance to security.

Given that string matching is a bottleneck and performance is important, there is at least one way to improve the performance of system that an efficient fast string-matching algorithm could be used to search file for a set of signatures.

In this virus detection tool, a fast pattern-matching algorithm Boyer-Moore-Horspool (BMH) has been used. It has better performance than the iterative use of Boyer-Moore, currently used in some popular virus detection softwares and much better than an efficient sequential string searching algorithm.

Before introducing BMH algorithm, a brief review of the string matching problem, Sequential, Boyer-Moore and Turbo Boyer-Moore pattern matching approaches is given below.

6.1 Pattern Matching Algorithms

Assume a text string T of length n and a pattern string P of length m, each composed of an ordered set of characters from an common alphabet A. The general problem is to determine the location of P within T, or that T does not contain P.

6.1.1 Sequential pattern matching algorithm

This is the simple sequential search algorithm. It uses a linear and sequential character-based comparison at all positions in the text between y_0 and y_{n-m-1}, whether or not an occurrence of the pattern x starts at the current position. In case of success in matching the first element of the pattern x_0, each element of the pattern is successively tested against the text until failure or success occurs at the last position. After each unsuccessful attempt, the pattern is shifted exactly one position to the right, and this procedure is repeated until the end of the target is reached.

This algorithm has only one advantage, that it needs no preprocessing of any kind on the pattern, but at the cost of more search time.

6.1.2 Boyer Moore Algorithm

The Boyer-Moore algorithm is considered as the most efficient pattern-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the *Search* and *Substitute* commands [41].

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two

shift functions are called the *good-suffix shift* (also called matching shift) and the *bad-character shift* (also called the occurrence shift).

Assume that a mismatch occurs between the character $x[i]=a$ of the pattern and the character $y[i+j]=b$ of the text during an attempt at position j. Then, $x[i+1 .. m-1]=y[i+j+1 .. j+m-1]=u$ and $x[i] \neq y[i+j]$. The good-suffix shift consists in aligning the segment $y[i+j+1 .. j+m-1]=x[i+1 .. m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$.

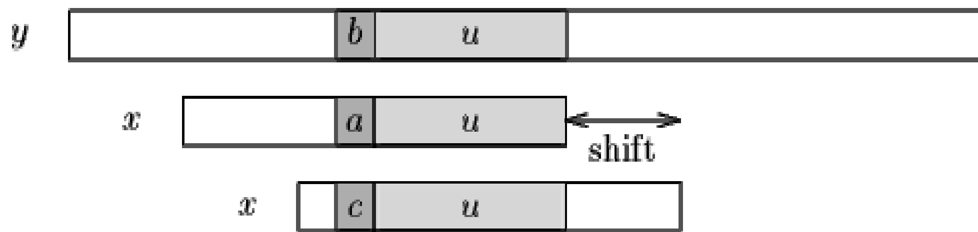

Figure 6.1. The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i+j+1 .. j+m-1]$ with a matching prefix of x.

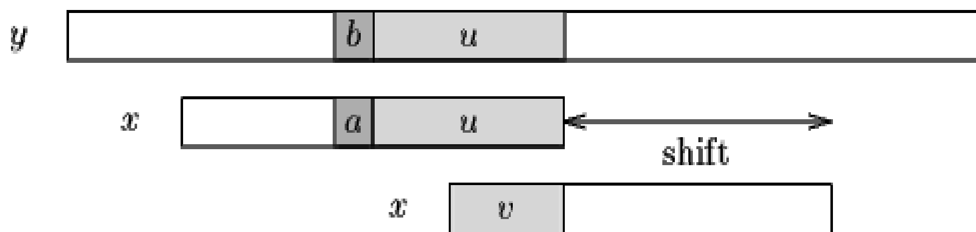

Figure 6.2. The good-suffix shift, only a suffix of *u* re-occurs in *x*.

The bad-character shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0 .. m-2]$.

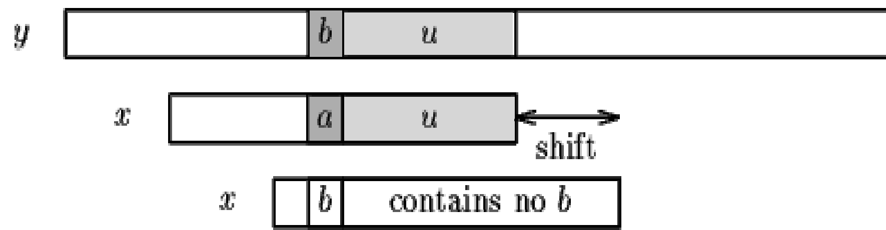

Figure 6.3. The bad-character shift, *a* occurs in *x*.

If *y* [*i*+*j*] does not occur in the pattern *x*, no occurrence of *x* in *y* can include *y*[*i*+*j*], and the left end of the window is aligned with the character immediately after *y*[*i*+*j*], namely *y*[*i*+*j*+1].

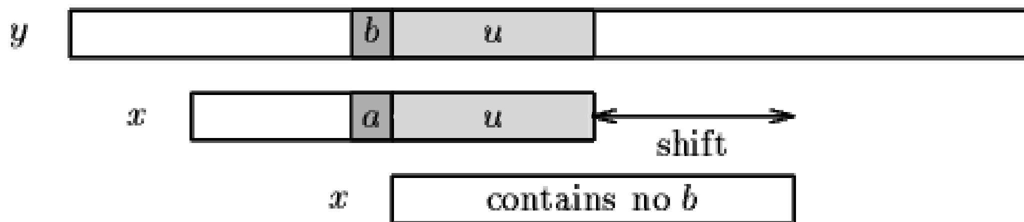

Figure 6.4. The bad-character shift, *b* does not occur in *x*.

Since the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.

The good-suffix shift function is stored in a table *bmGs* of size *m*+1.

Let us define two conditions:

- *Cs* (*i*, *s*): for each *k* such that $i < k < m$, $s \geqslant k$ or $x[k\text{-}s]=x[k]$
- *Co* (*i*, *s*): if $s < i$ then $x[i\text{-}s] \neq x[i]$

Then, for $0 \leqslant i < m$: $bmGs[i+1]=\min\{s>0 : Cs(i, s) \text{ and } Co(i, s) \text{ hold}\}$ and we define *bmGs*[0] as the length of the period of *x*. The computation of the table

bmGs use a table *suff* defined as follows: for $1 \leqslant i < m$, $suff[i]=\max\{k : x[i-k+1 .. i]=x[m-k .. m-1]\}$

The bad-character shift function is stored in a table *bmBc* of size σ. For c in Σ: $bmBc[c] = \min\{i: 1 \leqslant i < m-1 \text{ and } x[m-1-i]=c\}$ if c occurs in x, m otherwise.

6.1.2.1 Performance analysis of BM

Tables *bmBc* and *bmGs* can be precomputed in time $O(m+\sigma)$ before the searching phase and require an extra-space in $O(m+\sigma)$. The searching phase time complexity is quadratic $O(mn)$, but at most $3n$ text character comparisons are performed when searching for a non-periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for $a^{m-1}b$ in b^n the algorithm makes only $O(n/m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

6.1.3 Turbo Boyer Moore Algorithm

The Turbo-BM algorithm is an amelioration of the Boyer Moore algorithm. It needs no extra preprocessing and requires only a constant extra space with respect to the original Boyer-Moore algorithm. It consists of remembering the factor of the text that matched a suffix of the pattern during the last attempt (and only if a good- suffix shift was performed) [41].

This technique presents two advantages:

- It is possible to jump over this factor;
- It can enable to perform a turbo-shift.

A turbo-shift can occur if during the current attempt the suffix of the pattern that matches the text is shorter than the one remembered from the preceding attempt. In this case let us call u the remembered factor and v the suffix matched during the current attempt such that uzv is a suffix of x. Let a and b be the characters that cause the mismatch during the current attempt in the pattern and the text respectively. Then av is a suffix of x, and thus of u since $|v| < |u|$. The two characters a and b occur at distance p

in the text, and the suffix of x of length $|uzv|$ has a period of length $p=|zv|$ since u is a border of uzv, thus it cannot overlap both occurrences of two different characters a and b, at distance p, in the text. The smallest shift possible has length $|u|-|v|$, which we call a turbo-shift as shown in figure.

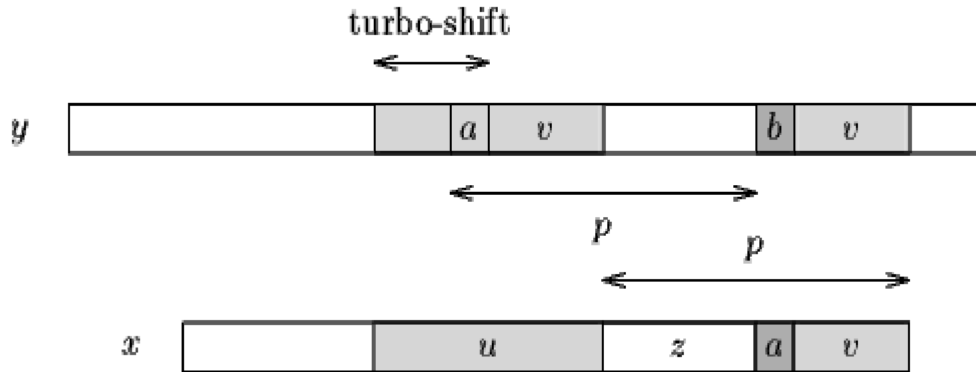

Figure 6.5 A turbo-shift can apply when $|v|<|u|$.

Still in the case where $|v|<|u|$ if the length of the bad-character shift is larger than the length of the good-suffix shift and the length of the turbo-shift then the length of the actual shift must be greater or equal to $|u|+1$. Indeed, in this case the two characters c and d are different since we assumed that the previous shift was a good-suffix shift.

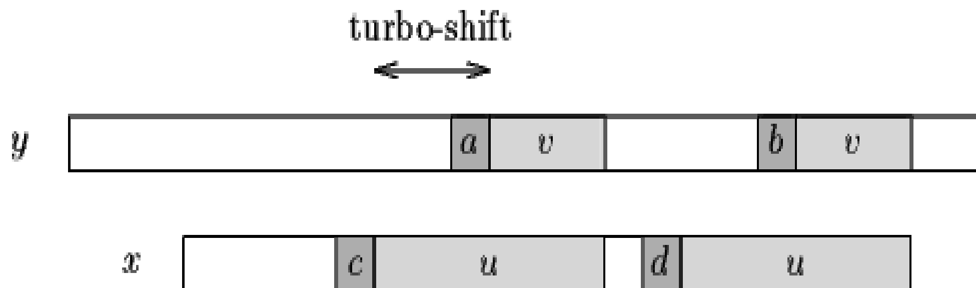

Figure 6.6 $c \neq d$ so they cannot be aligned with the same character in v.

Then a shift greater than the turbo-shift but smaller than $|u|+1$ would align c and d with a same character in v. Thus in this case the length of the actual shift must be at least equal to $|u|+1$.

6.1.3.1 Performance analysis of TBM

The preprocessing phase can be performed in $\boldsymbol{O}$ $(m+\sigma)$ time and space complexity. The searching phase is in $\boldsymbol{O}$ (n) time complexity. The number of text character comparisons performed by the Turbo-BM algorithm is bounded by $2n$.

6.1.4 The Boyer-Moore-Horspool Algorithm

The Boyer-Moore algorithm and its variant Turbo Boyer-Moore are considered the most efficient pattern-matching algorithms. Both these methods can be preprocessed and kept in tables. Their time and space complexities depend only on the pattern.

However, although there is a theoretic advantage in using the BM algorithm, many computational steps in this algorithm are costly in terms of processor instructions. The cost in time of the computational step was not shown to be amortized by the economy of character comparisons. This is particularly true of the function that computes the size comparisons in the skip tables. Simpler search algorithms can often perform better than algorithms, which skip more characters per comparison, but require much more work per skip.

Horspool proposed a simplified form of the BM algorithm that uses only a single auxiliary skip table indexed by the mismatching text symbols i.e. the bad character heuristic [41]. Baeza-Yates showed that the Boyer-Moore-Horspool algorithm (BMH) is the best in terms of average case performance for nearly all pattern lengths and alphabet sizes.

Among all pattern-matching algorithms tested here, the BMH algorithm showed the best performance in time. Despite its apparent conceptual complexity, the BMH algorithm is relatively simple to implement. Its time complexity is $\boldsymbol{O}$ (NM) in the worst case but has better average performance than BM, both analytically and experimentally. Also because of its low space complexity, the use of the BMH algorithm in any case of exact string pattern matching, is recommended, whatever the size of the target. Since a

virus scanner should be very fast and economical, as well as it should be able to handle very large files also, without being slow, hence for this purpose BMH algorithm appears to be the best-choice algorithm.

6.1.4.1 Description of the algorithm

The BMH algorithm requires preprocessing of the pattern, and works by comparing letter by letter the characters of T with the characters of P to find out where the pattern matches.

6.1.4.1.1 Implementation and data structures

The BMH algorithm works as follows: a position indicator j is set up for the text, and a position indicator k is set for the pattern. The matching process starts by aligning the first letter of P under" the first letter of T. It is as if we had opened a window on the text that allows us to see only m characters. Later, this window will slide to the right, to allow us to view other positions. Another position indication i is required to record the position of the rightmost text position viewed through the window. i is initialized to $m-1$. Starting at letter P_{m-1}, letter-by-letter comparisons are done between T_j and P_k. Both j and k are decremented after each successful comparison. These comparisons continue as long as characters match and as there are uncompared elements in the pattern. If all characters in the pattern have been successfully compared (that is $k=-1$); then we have found the pattern in the text. If a mismatch is detected, the algorithm recognizes the failure of the current window, and no pattern can be possibly found. Whether a match is found or not, the window is shifted a certain predesignated distance d to the right: the position indicator is incremented by d, j is reset to i and k to $m-1$. The whole process is repeated until we reach the end of the text.

In order to determine shift distance d, the pattern has to be preprocessed: for each character $a \in A$, a distance d_s is computed. This distance depends only on the position of a in the pattern. When we have to move the window on the right, the right-

most character ***t*** of the windows determines the shift distance: it is natural to align this character with the nearest occurrence of ***t*** in ***P***. The shift distance can be computed this way: $\min\{s \geq 1, P_{m-s}=t\}$. If ***t*** does not appear in the pattern, we cannot have any match containing this letter. That is, we know that any window containing this letter ***t*** will not match. So, we can move the window to the right for ***m*** characters.

This algorithm can be written this way:

```
while i < n  do   {A new valid window is defined}
        j=i
        k=m-1
        while  k >= 0 and  T[j] = P[k]  do
               decrement  j
               decrement  k
         end while
         if   k < 0  then    { Found en occurrence of the pattern}
              process to be realized in case of match
             {print the line}
         end if
         I = I + D[ti]    {shift the window on the right}
End while
```

And the preprocessing step can be written as follow:

```
For ∀a ∈ A do
   D [a]=m
End for
For i=0 to m-2 do
   D [Pi]=m- (i+1)
 End for
```

Data structures used to implement this algorithm are essentially arrays, which are efficient data structure adapted to this algorithm. The array containing the pattern is dynamically allocated, but the array containing the line of the text is statically allocated.

The alphabet A considered contains at most 256 characters (there is a parameter ALPHABET_SIZE). It is the standard ASCII alphabet, without special characters. It is worth noting that in C, there is a complete equivalence between a character and an 8 bits integer (A character is stored in memory by its ASCII value). Hence, it is possible to index an array with characters.

6.2 Description of the Signature Scanner

The tool has two main components: Signature Database and an Engine that scans files for viruses against signatures stored in the database. They are complementary to each other and cannot work independently.

6.2.1 Implementation

The first step to implement signature scanner is to build the signature database of all the viruses known till today. While in the second step, actual virus searching is done for the viruses stored in signature database.

6.2.1.1 Signature Database

It is a database of uniquely identifiable "signatures" that a virus contains. The signature for an executable virus typically is a series of machine code bytes that a virus always contains. In this every virus record has following fields:

1. Virus signature in HEX
2. Type of virus (B: Boot sector, P: partition table, F: file virus)
3. Description of the virus

Whenever a new virus appears, database can be updated through a data entry program. It first asks user to enter the signature. It has to be in HEX and without commas and blank spaces. Next the type of virus has to be entered and finally the description of virus can be entered. The virus description includes the virus name, properties, comments, etc. After all the data has been verified, it is saved to the signature database.

6.2.1.2 Virus detection engine

It scans boot sector, partition table and files of all types for viruses. The scanner starts by reading information about viruses from the signature file. Now it knows the particular sequence of code and is looking for an exact match, which will identify the code as a virus. This program has to be passed certain command line options. For example, -F *.COM checks all .COM files and -P 1 signifies a check of the partition table of the first hard disk.

The database file contains, besides the signature and description of the virus, its type as well. That is, whether it is a Boot sector, Partition or a File type of virus. Depending on the value of the type field, an array of structures representing these virus types is created to represent each virus record. If at the command line user passes option –F C:*.EXE, this means that a check with all file virus signatures for all EXE files in C: directory. Now the signatures of all the file viruses are dumped into the array of structures. And signature by signature all executable files are searched for each virus. The same procedure is followed for boot sector and partition table also.

For efficiency point of view, Firstchar array stores the first nibble of all signatures, and if these first nibble matches while scanning the executables, then only further matching with the respective signature is done. To keep the scanning speed fast, Boyer-Moore-Horspool algorithm has been used, which is a very fast exact string-matching algorithm.

Every file will be scanned from first byte to last byte against the signature database. If certain anomalous patterns are detected, it will notify the user.

6.3 Performance analysis

6.3.1 Measures

All algorithms have been implemented in C. The tests were conducted on a 750 MHz Pentium IV processor PC with 128-Mbytes RAM having 20-GB hard disk divided into four drives of 5 GB each, running under MS DOS 6.0. The 1127 target files, which occupied total size 1.5 GB, were searched.

All preprocessing loads were measured within their respective algorithms. The parameters of the function calls of each algorithm used pointers to zero-terminated strings (target and signature) in order to avoid variance due to memory management. All memory for strings was allocated before the calls. The functions, however, were self-contained. The functions return the value 0 in case of failure, or else they return the position in the target where the first occurrence of pattern was found. No preprocessing or global variables were needed for their execution. Whenever possible, the functions were optimized for speed. Time measures were done using the motherboard's high-resolution performance counter.

The target text was divided into slices of 1024 characters, except the last one, which might have fewer characters. All measurements were done in an incremental manner growing in steps from the size of one slice to the whole target size. All algorithms were tested for the various patterns.

All the algorithms considered in this dissertation incur an approximately constant cost in space. In the worst case, the extra space needed for processing is a linear function of the length of the pattern, which is negligible. Moreover, in this analysis, space complexity is similar for all algorithms tested.

The accuracy with which this tool could detect viruses has been tested by using the standard VB set of viruses.

6.3.2 Choice of Patterns

All tests were performed using three types of signatures. The first type tested was Boot Sector Virus Signature. This word was chosen because it will be found in boot sector of the system. The second pattern was for Partition Table Virus. It infects partition table of the hard disk. The last pattern belonged to file type viruses. These viruses infect files in system; hence the pattern is located somewhere in file.

6.3.3 Search for Boot sector viruses

The first test was done in boot sector using the boot sector virus signatures. In this case size of target text was 512 bytes. The difference in performance of all BM and its variants was subtle, because of the smaller size of target text. But it is worth noting that BMH is 1.72 times faster than the sequential algorithm.

6.3.4 Search for Partition table viruses

In the second test partition table of hard disk was searched for partition table type virus signatures. Since here also the target text size is the same as that of the boot sector, so results were also the same.

6.3.5 Search for File type viruses

In this test, 1127 files occupying 1.5 GB space were searched.

Database Size (No. of Patterns)	**Sequential Algorithm** (Sec)	**TBM** (Sec)	**BM** (Sec)	**BMH** (Sec)
20	8.6	6.8	6.3	5.3
40	10.7	9.4	8.2	7.2
60	11.4	10.2	9.2	8.5
80	15.6	14.8	11.5	9.6
100	18.5	18.1	13.9	11.8

Table 6.1 Performance on the basis of Signature Database Size

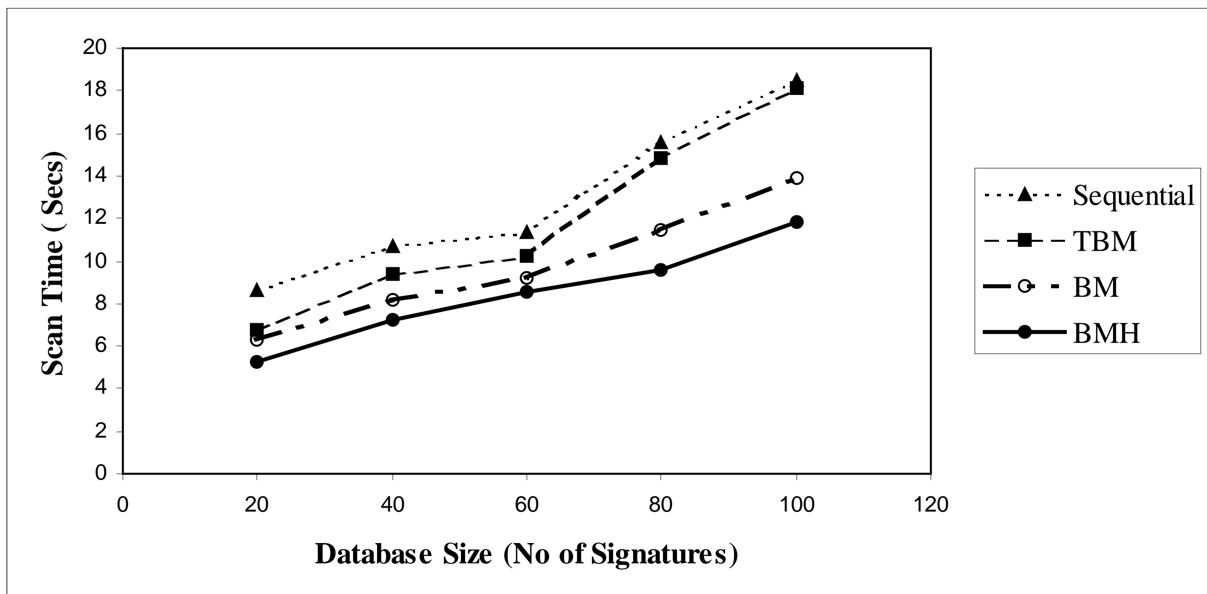

Figure 6.7 Performance graph on the basis of Signature Database size

Table 6.1 and figure 6.7 shows the performance of all the algorithms by varying database size. It is clearly indicated by this table and figure that BMH is the fastest algorithm among others shown here.

6.3.6 Performance According to Pattern Size

Algorithms that do not use a skip table to optimize the shift function are rather independent of the pattern size in their time complexity. This is not the case for the Boyer-Moore algorithm and its variants. In these algorithms, larger pattern size means longer the skip shift in case of mismatch and, therefore, the faster the algorithm. Nevertheless, the BMH algorithm is still fastest among all algorithms tested. The overall results of all algorithms tested are shown in table 6.2 and figure 6.8.

Pattern Size (No. of Chars)	**Sequential Algorithm** (Sec)	**TBM** (Sec)	**BM** (Sec)	**BMH** (Sec)
8	4.34	3.46	3.35	3.29
16	4.50	3.35	3.29	3.24
20	5.27	2.91	2.75	2.08
32	5.34	2.23	2.14	1.89
48	6.26	1.86	1.75	1.70

Table 6.2 Performance on the basis of the pattern size

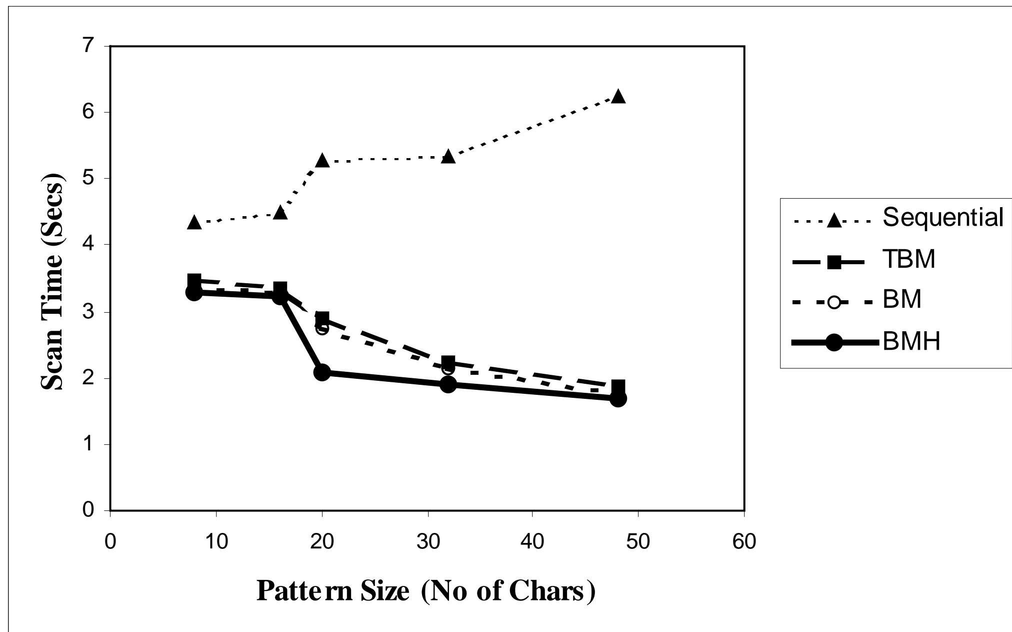

Figure 6.8 Performance graph on the basis of the pattern size

This table emphasizes the extreme speed of the BMH algorithm and the slight advantage to long patterns. It shows also that the sequential algorithm, compared with the BMH algorithm, performs less well with long patterns than with short ones.

6.4 Summary of results

Programs that detect viruses by scanning for known patterns are judged by two criteria: how fast they scan and how well they detect the viruses. The scanning speed of this tool is tested by searching the whole of the hard disk on an IBM PC compatible system for viruses.

The BMH algorithm is a fast and easy-to-implement algorithm. It typically performs better than the Boyer Moore algorithm, which is mostly used for string pattern matching. Considering the growing amount of text, that needs to be handled by a virus detection system, the BMH algorithm is worth implementing in any case. Other algorithms that could theoretically perform better do not, compared with the BMH algorithm under real conditions. If long patterns are used, a more conservative control

of the right-to-left comparison loop can slightly improve the time performance of the BMH algorithm.

This virus detection tool is very efficient. Several techniques are used to keep a handle on performance. First, signatures are classified by the type of infection they represent like boot sector, partition table or file type. Through a process of elimination, when a particular file is scanned, only the signatures that pertain to that file type is used to keep scan times down. For example, a boot sector signature would not be used to scan a file. If a signature matches, with the 'Virus Detected' warning, the name of file, the offset at which it is found in the file and the description of the virus is displayed on the screen.

The initialization overhead of reading in the patterns from signature database and storing them in the internal structure "virus" is very small. The scanner has the potential to scan an almost infinite range of different file types. In practice, however, not all file types need to be scanned because some types of file, e.g. ASCII text files are not capable of being virus carriers. It is very easy to use. Simply execute the scanner and it provides concise results. It has options describing which disk, files, or directories to scan, but the user does not have to be a computer expert to select the right parameters or comprehend the results.

CHAPTER 7
CONCLUSION AND FUTURE WORK

This work represents a prototype of a virus detection system. Therefore there is a lot of work that could still be done from both a research point of view and from a commercial point of view. This chapter will suggest some of the directions that future research could take as well as what would be necessary to make the move from research to commercial viability.

In this virus detection system, viruses are detected by using two virus detection tools that is an integrity checker and signature scanner. Viruses have great difficulty in infecting machine without making some change in it. To detect a change is to begin the process of virus detcetion, that is the approach integrity checker is using. It is using SHA-1 algorithm to generate 160-bit checksum, which is large enough to avoid forgery. It also computes file size, creation date, and last modification date, to ensure that a virus is detected in case of any mismatch. Integrity checker detects all the infections whether it is by known or unknown viruses and its performance is acceptable. Virus Scanner is primarily used to detect if an executable contains virus code or not, but it can also be used to detect resident viruses by scanning memory instead of executables. While being prone to false positives sometimes, it is pretty accurate. Signature-based approaches should not be abandoned, as they are useful for cleaning up infected computers, after getting the information about area of infection and its cause.

The implementation demonstrated here provided good results within acceptable time. By carefully using the BMH algorithm, the performance of virus detection system is improved as compared to the performance of a Boyer Moore algorithm-based system.

Virus detection in general is an undecidable problem. We cannot devise a method that can detect all possible viruses. Our method is also bounded by this theoretical limit. For instance, it cannot detect the presence of virus before infection.

Currently the signature database has 100 signatures, but to use it in the practical world all the existing virus signatures have to be maintained in the database.

If storage and time are not at a premium, then integrity checker can combine two or more techniques to generate the checksum. In this way more security can be obtained but at the cost of speed and memory. Having shown within this work, the approaches to virus detection does demonstrate promising results, the next step would be to include a technique that can detect the viruses before they have any chance to infect the system.

REFERENCES

[1] Leonard Adleman. An abstract theory of computer viruses. In *Lecture Notes in Computer Science, vol 40*3. Springer-Verlag, 1990.

[2] Fred Cohen. *Computer Viruse*s. PhD thesis, University of Southern California, 1985.

[3] Deborah Russell and Sr. G. T. Gangemi. *Computer Security Basic*s. O'Reilly & Associates,Cambridge, MA, 1991.

[4] Alan Solomon. *PC VIRUSES Detection, Analysis and Cur*e. Springer-Verlag, London, 1991.

[5]Andrew S TanenBaum, "*Modern operating System*". Prentice Hall of India, 2003.

[6]Richard D. Pethia, "*Computer Viruses: The Disease, the Detection, and the Prescription for Protection*", Carnegie Mellon University, November 6, 2003.

[7]Jan Hruska, "*Computer virus prevention: a primer*", Oxford University, August 2000.

[8] Jake Ferry. "*A Study and Evaluation of Virus Protection Software Marketed to Average Computer Users*." Dissertation ES200006, Department of Computer Science, University of Virginia, 2000.

[9] Lisa J. Carnahan and John P. Wack. *Computer Viruses and Related Threats: A Management Guid*e. NIST Special Publication 500-166, National Institute of Standards and Technology, 1989.

[10] David Chess. Common viruses. *Virus News and Review*s, 1:106– 107, March 1992.

[11] Frederick B. Cohen. Acost analysis of typical computer viruses and defenses. In *Safe Computing: Proceedings of the 4th Computer Virus & Security Con-ferenc*e, pages 737– 750. DPMA, 1991.

[12] George I. Davida, Yvo G. Desmedt, and Brian J. Matt. Defending systems against viruses through cryptographic authentication. In *Proceedings of the 1989 IEEE Symposium on Computer Security and Privac*y, pages 312– 318, 1989.

[13] Peter J. Denning, editor. *Computers Under Attack: Intruders, Worms, and Viruse*s. ACM Books/Addison-Wesley, 1991.

[14] Frederick B. Cohen; *Computer Viruses, Theory and Experiments*; 7th Security Conference, DOD/NBS Sept 1984.

[15] David Ferbrache. *A Pathology of Computer Viruse*s. Springer-Verlag, Lon-don, 1992.

[16] Lance Hoffman, editor. *Rogue Programs: Viruses, Worms, and Trojan Horse*s. Van Nostrand Reinhold, 1990.

[17] Keith Jackson. Product review: Central Point Anti-Virus. *Virus Bulleti*n, pages 21– 23, May 1992.

[18] Keith Jackson. Product review: SmartScan. *Virus Bulleti*n, pages 16– 18, July 1992.

[19] Keith Jackson. Product review: Vi-Spy Professional Edition. *Virus Bulleti*n, pages 24– 26, August 1992.

[20] Maria M. King. Identifying and controlling undesirable program behaviors. In *Proceedings of the 14th National Computer Security Conferenc*e, pages 283– 294, 1991.

[21] Yisrael Radai. Checksumming techniques for anti-viral purposes. *Virus Bulletin Conferenc*e, 6:39– 68, September 1991.

[22] Eugene H. Spafford, Kathleen A. Heaphy, and David Ferbrache. *Com-puter Viruses: Dealing with Electronic Vandalism and Programmed Threat*s. ADAPSO, Arlington, VA, 1989.
[23] Steve R. White, David M. Chess, and Chengi Jimmy Kuo. Coping with computer viruses and related problems. *International Business Machines Corporatio*n, 1989.

[24] Dark Angel's Phunky Virus Writing Guide, URL: http://vx.netlux.org/lib/static/vdat/tuda0001.htm

[25] GIAC Code of Ethics, URL: http://www.giac.org/COE.php

[26] ZDNet UK. New page. 9 May 2000. ZDNet UK. 9 May 2000 <http://www.zdnet.co.uk/news/2000/18/ns-15265.html>.

[27] F. Cohen, "A Complexity Based Integrity Maintenance Mechanism", Conference on Information Sciences and Systems, Princeton University, March 1986.

[28] M. Pozzo and T. Gray, "An Approach to Containing Computer Viruses", Computers and Security, IFIP-SEC V6#2, 1987.

[29] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", CACM V21 (1978) pp 120-126.

[30] National Institute of Standards and Technology, Specifications for the SECURE HASH STANDARD, August 2002.

[31] National Institute of Standards and Technology, Descriptions of SHA-256, SHA-384, and SHA-512, August 2002.

[32] National Institute of Standards and Technology, Description of SHA-1, Federal Information Processing Standards Publication 180-1, 1995 April 17.

[33] SANS Institute, A Guide to Hash Algorithm by Britt Savage, April 2003.

[34] SHA: A Design for Parallel Architectures Antoon Bosselears, Rene Govaerts and Joos Vandewalle, 25th February 1997.

[35] Fast Hashing on the Pentium, Antoon Bosselears, Rene Govaerts and Joos Vandewalle, Lecture Notes in Computer Science , Vol. 1109, pp 298, 1996.

[36] Michael Roe, Cambridge University Computer Laboratory, Performance of Symmetric Ciphers and One-way hash functions.

[37] RFC 3174, Secure Hash Algorithm 1, September 2001.

[38] ACM SIGCOMM Computer Communication and Review, Performance Analysis of MD5, Joseph D. Touch, Volume 25, Issue 4, October 1995.

[39] Cryptography and Network Security: Principles and Practice(3rd Edition), William Stallings.

[40] Introduction to Public Key Cryptography, http://www.netscape.com.

[41] Alfred V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Scienc*e, chapter 5, pages 256– 300.Elsevier Science Publishers, 1990.

APPENDIX A: OUTPUT SNAPSHOTS

I. Integrity Checker Output

C:\TC\BIN>checker -c E:*.* edir.txt

Scanning E:\MAJORP~1\FILEVE~1.HTM ...
Scanning E:\MAJORP~1\INTEGR~1.TXT ....
Scanning E:\MAJORP~1\REPORT~1.DOC ....
Scanning E:\MAJORP~1\REPORT~2.DOC ....
Scanning E:\MAJORP~1\REPORT~3.DOC ....
Scanning E:\MAJORP~1\REPORT~4.DOC ....
Scanning E:\MAJORP~1\SIGNAT~1\SIGNVIR.DAT .
Scanning E:\MAJORP~1\SIGNAT~1\TESTFILE.C .
Scanning E:\MAJORP~1\SIGNAT~1\WRITEFIL.C .
Scanning E:\MAJORP~1\SIGNAT~1\SCANNER.C .
Scanning E:\MAJORP~1\SIGNAT~1\APR22S~1\SCANNER.EXE ..
Scanning E:\MAJORP~1\SIGNAT~1\APR22S~1\SCANNER.C .
Scanning E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER1.EXE ..
Scanning E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER1.C .
Scanning E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER.C .
Scanning E:\MAJORP~1\SIGNAT~1\24APR~1\SCANGEN.C .
Scanning E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER.EXE ..
Scanning E:\MAJORP~1\SIGNAT~1\24APR~1\SCANGEN.EXE ..
Scanning E:\MAJORP~1\SIGNAT~1\26APRB~1\SCANNER.EXE ..
Scanning E:\MAJORP~1\SIGNAT~1\26APRB~1\SCANNER.C .
Scanning E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANGEN.C .
Scanning E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANNER.C .
Scanning E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANNER.BAK .
Scanning E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANNER.EXE
Scanning E:\MAJORP~1\INTEGR~2\24APR~1\CHEKSHA1.C ..
Scanning E:\MAJORP~1\INTEGR~2\24APR~1\CHECKER.C ..
Scanning E:\MAJORP~1\INTEGR~2\24APR~1\CRC_32\CHECKCRC.EXE ..
Scanning E:\MAJORP~1\INTEGR~2\24APR~1\CRC_32\CHECKCRC.C .
Scanning E:\PROJECT\DATABSE.BAK .
Scanning E:\PROJECT\SIGDETEC.BAK .
Scanning E:\PROJECT\CERTIF~1.DOC ....
Scanning E:\PROJECT\CONT.DOC ..
Scanning E:\PROJECT\FINALR~1.DOC ....
Scanning E:\PROJECT\FRONTP~1.DOC ....
Scanning E:\PROJECT\SOURCE~1.DOC ....

C:\TC\BIN>checker edir.txt

Checking file E:\MAJORP~1\FILEVE~1.HTM ... OK
Checking file E:\MAJORP~1\INTEGR~1.TXT .... OK
Checking file E:\MAJORP~1\REPORT~1.DOC .... OK
Checking file E:\MAJORP~1\REPORT~2.DOC .... OK
Checking file E:\MAJORP~1\REPORT~3.DOC .... OK
Checking file E:\MAJORP~1\REPORT~4.DOC .... OK
Checking file E:\MAJORP~1\SIGNAT~1\SIGNVIR.DAT . OK
Checking file E:\MAJORP~1\SIGNAT~1\TESTFILE.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\WRITEFIL.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\SCANNER.C . OK
Checking file E:\PROJECT\DATABSE.BAK . OK
Checking file E:\PROJECT\SIGDETEC.BAK . OK
Checking file E:\PROJECT\CERTIF~1.DOC .... OK
Checking file E:\PROJECT\CONT.DOC .. OK
Checking file E:\PROJECT\FINALR~1.DOC .... OK
Checking file E:\PROJECT\FRONTP~1.DOC .... OK
Checking file E:\PROJECT\SOURCE~1.DOC .... OK
Checking file E:\MAJORP~1\FILEVE~1.HTM ... OK
Checking file E:\MAJORP~1\INTEGR~1.TXT .... OK
Checking file E:\MAJORP~1\REPORT~1.DOC .... OK
Checking file E:\MAJORP~1\REPORT~2.DOC .... OK
Checking file E:\MAJORP~1\REPORT~3.DOC ....
Date of creation of this file differs
Error: E:\MAJORP~1\REPORT~3.DOC has been modified either by user or by some malicious program
Checking file E:\MAJORP~1\REPORT~4.DOC .... OK
Checking file E:\MAJORP~1\SIGNAT~1\SIGNVIR.DAT . OK
Checking file E:\MAJORP~1\SIGNAT~1\TESTFILE.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\WRITEFIL.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\SCANNER.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\APR22S~1\SCANNER.EXE .. OK
Checking file E:\MAJORP~1\SIGNAT~1\APR22S~1\SCANNER.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER1.EXE .. OK
Checking file E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER1.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\24APR~1\SCANGEN.C . OK
Checking file E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER.EXE ..
Last modified date of this file differs
Error: E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER.EXE has been modified either by user or by some malicious program

Checking file E:\MAJORP~1\SIGNAT~1\24APR~1\SCANGEN.EXE .. OK
Checking file E:\MAJORP~1\SIGNAT~1\26APRB~1\SCANNER.EXE .. OK
Checking file E:\MAJORP~1\SIGNAT~1\26APRB~1\SCANNER.C . OK

Checking file E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANGEN.C .
Checksum of this file differs
Error: E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANGEN.C has been modified either by user or by some malicious program

Checking file E:\MAJORP~1\INTEGR~2\20APR~1\Q.TXT . OK
Checking file E:\MAJORP~1\INTEGR~2\20APR~1\SHA.H . OK
Checking file E:\MAJORP~1\INTEGR~2\20APR~1\W.TXT . OK
Checking file E:\MAJORP~1\INTEGR~2\24APR~1\CHEKSHA1.EXE ... OK
Checking file E:\MAJORP~1\INTEGR~2\24APR~1\CHEKSHA1.C .. OK
Checking file E:\MAJORP~1\INTEGR~2\24APR~1\CHECKER.C .. OK
Checking file E:\MAJORP~1\INTEGR~2\24APR~1\CRC_32\CHECKCRC.EXE .. OK
Checking file E:\MAJORP~1\INTEGR~2\24APR~1\CRC_32\CHECKCRC.C . OK
Checking file E:\PROJECT\DATABSE.BAK . OK
Checking file E:\PROJECT\SIGDETEC.BAK . OK
Checking file E:\PROJECT\CERTIF~1.DOC .... OK
Checking file E:\PROJECT\CONT.DOC .. OK
Checking file E:\PROJECT\FINALR~1.DOC .... OK
Checking file E:\PROJECT\FRONTP~1.DOC .... OK
Checking file E:\PROJECT\SOURCE~1.DOC .... OK

******** Summary ********

Modified Files are:
E:\MAJORP~1\REPORT~3.DOC
E:\MAJORP~1\SIGNAT~1\24APR~1\SCANNER.EXE
E:\MAJORP~1\SIGNAT~1\13MAY~1\SCANGEN.C

II. Database.exe Output

*********** DATABASE ENTRY PROGRAM ***************
Every virus record will have three fields:

1. Signature in HEX
2. Type of the virus
3. Description of virus

Enter the signature of virus in HEXADECIMAL (without comma and blank spaces):
DD7A0BA8

Enter the TYPE of Virus (P: partition table, B: Boot sector, F: File virus):
Type of virus is F

Enter description of the virus:
Code Red Virus

Are the details regarding this virus correct? (Y/n): y
Do you wish to continue? (Y/n): n

III. Scanner Output

(A) If this program is **not run through command prompt** then following error is displayed on screen.

ERROR: Please run the program through command prompt and then
follow the menu given below:

1.Enter -B<drive no> to check the Boot Sector
2.Enter -P<hard disk no> to check the Partition Table
3.Enter -F<file specification> to check the files

(B) Program run through command prompt and an option to check **the boot sector** has been given:

C:\TC\BIN>scanner -b

No of signatures in database : 100

Checking Boot Sector : A (Boot sector of Floppy)...........

*** No viruses present ***

(C) Option to check **the partition table** of hard disk 1 is provided:

C:\TC\BIN>scanner -p

No of signatures in database : 100

Checking partiton table of specified Hard Disk: 1……………

*** No viruses present ***

(D) Output when all the **Executable files in C Directory** has to be checked:

C:\TC\BIN>scanner -f C:/*.EXE

No of signatures in database : 100

Type of files to be scanned is C:/*.EXE

Checking C:/*.EXE files.......
Checking file C:/TC.EXE
Checking file C:/WINDOWS*.EXE
Checking file C:/WINDOWS\HWINFO.EXE
Checking file C:/WINDOWS\CLSPACK.EXE
Checking file C:/WINDOWS\DRWATSON.EXE
Checking file C:/WINDOWS\EXPLORER.EXE
Checking file C:/WINDOWS\EXTRAC32.EXE
Checking file C:/WINDOWS\FONTVIEW.EXE
Checking file C:/WINDOWS\GRPCONV.EXE
Checking file C:/WINDOWS\HH.EXE
Checking file C:/WINDOWS\JVIEW.EXE
Checking file C:/WINDOWS\MSNMGSR1.EXE
Checking file C:/WINDOWS\NETCONN.EXE
Checking file C:/WINDOWS\PIDSET.EXE
Checking file C:/WINDOWS\SETDEBUG.EXE
Checking file C:/WINDOWS\SIGVERIF.EXE
Checking file C:/TC\BIN\CH24_2.EXE
Checking file C:/TC\BIN\CH24_25.EXE
Checking file C:/TC\BIN\CPP.EXE
Checking file C:/TC\BIN\DPMIINST.EXE
Checking file C:/TC\BIN\DPMILOAD.EXE
Checking file C:/TC\BIN\DPMIRES.EXE
Checking file C:/TC\BIN\EX1.EXE
Checking file C:/TC\BIN\GREP2MSG.EXE
Checking file C:/TC\BIN\MAKE.EXE
Checking file C:/TC\BIN\MAKER.EXE

Checking file C:/TC\BIN\PRJ2MAK.EXE
Checking file C:/TC\BIN\PRJCFG.EXE
Checking file C:/TC\BIN\PRJCNVT.EXE
Checking file C:/TC\BIN\TASM2MSG.EXE
Checking file C:/TC\BIN\TC.EXE
Checking file C:/TC\BIN\TCC.EXE
Checking file C:/TC\BIN\TDUMP.EXE
Checking file C:/TC\BIN\TEMC.EXE
Checking file C:/TC\BIN\TIMEIT.EXE
Checking file C:/TC\BIN\TLIB.EXE
Checking file C:/TC\BIN\TLINK.EXE
Checking file C:/TC\BIN\TRANCOPY.EXE
Checking file C:/TC\BIN\TRIGRAPH.EXE
Checking file C:/TC\BIN\TRY1.EXE
Checking file C:/GHOSTGUM\GSVIEW\UNINSTGS.EXE
Checking file C:/GHOSTGUM\GSVIEW\GSPRINT.EXE
Checking file C:/GHOSTGUM\GSVIEW\EPSTOOL.EXE
Checking file C:/GHOSTGUM\PSTOTEXT*.EXE
Checking file C:/GHOSTGUM\PSTOTEXT\PSTOTXT3.EXE
Checking file C:/GHOSTGUM\PSTOEDIT*.EXE
Checking file C:/GHOSTGUM\PSTOEDIT\PSTOEDIT.EXE
****** No Viruses Detected******

(E) Output when option to check **all *.C files in C directory** is given:

C:\TC\BIN>scanner -f C:*.C

No of signatures in database: 100

Type of files to be scanned is C:*.C

Checking C:*.C files.......
Checking file C:\WINDOWS*.C
Checking file C:\WINDOWS\SYSTEM*.C
Checking file C:\WINDOWS\SYSTEM\MUI*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0401*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0403*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0404*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0405*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0406*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0407*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0408*.C
Checking file C:\WINDOWS\SYSTEM\MUI\0409*.C
Checking file C:\WINDOWS\SYSTEM\MUI\040B*.C

Checking file C:\WINDOWS\SYSTEM\MUI\040C*.C
Checking file C:\WINDOWS\SYSTEM\MUI\040D*.C
Checking file C:\WINDOWS\SYSTEM\MUI\040E*.C
Checking file C:\MYDOCU~1\SUNITA~1\SIGDETEC.C
Checking file C:\MYDOCU~1\SUNITA~1\DATABSE.C
Checking file C:\MYDOCU~1\PROJECT*.C
Checking file C:\TC*.C
Checking file C:\TC\BGI*.C
Checking file C:\TC\BGI\BGIDEMO.C
Checking file C:\TC\BIN*.C
Checking file C:\TC\BIN\CH24_25.C
Checking file C:\TC\BIN\SIGDETEC.C
Checking file C:\TC\BIN\TESTFILE.C

******** Virus Detected ********
Signature (dd7a0ba8) found at offset 76 in C:\TC\BIN\TESTFILE.C file.
Name of Virus: Code Red

Checking file C:\TC\BIN\A.C
Checking file C:\TC\BIN\TEST1.C
Checking file C:\TC\BIN\TEST2.C
Checking file C:\TC\BIN\TEST3.C
Checking file C:\TC\BIN\WRITEFIL.C

******** Virus Detected ********
Signature (dd7a0ba8) found at offset 105 in C:\TC\BIN\WRITEFIL.C file.
Name of Virus: Code Red

Checking file C:\TC\BIN\DATABSE.C
Checking file C:\TC\BIN\CPROGR~1\TEST.C
Checking file C:\TC\BIN\CPROGR~1\ABS.C
Checking file C:\TC\BIN\CPROGR~1\TEST1.C
Checking file C:\TC\BIN\CPLUSP~1*.C
Checking file C:\TC\BIN\ANTIVI~1*.C
Checking file C:\TC\BIN\ANTIVI~1\SIGDETEC.C
Checking file C:\TC\BIN\ANTIVI~1\DATABSE.C
Checking file C:\TC\BIN\ANTIVI~1\PROJEC~1*.C
Checking file C:\TC\BIN\ANTIVI~1\PROJEC~1\BOOTVACI.C
Checking file C:\TC\BIN\ANTIVI~1\PROJEC~1\MEMCHE.C
Checking file C:\TC\BIN\ANTIVI~1\PROJEC~1\PARTIVAC.C
Checking file C:\TC\BIN\ANTIVI~1\PROJEC~1\SIGDETEC.C
Checking file C:\TC\BIN\ANTIVI~1\PROJEC~1\DATABSE.C
Checking file C:\TC\BIN\ANTIVI~1\WORKING*.C
Checking file C:\TC\BIN\ANTIVI~1\WORKING\SIGDETEC.C
Checking file C:\TC\BIN\ANTIVI~1\WORKING\DATABSE.C
Checking file C:\TC\BIN\DEBUG*.C

Checking file C:\TC\CLASSLIB*.C
Checking file C:\TC\CLASSLIB\EXAMPLES*.C
Checking file C:\TC\CLASSLIB\INCLUDE*.C
Checking file C:\TC\CLASSLIB\LIB*.C
Checking file C:\TC\CLASSLIB\OBJS*.C
Checking file C:\TC\CLASSLIB\OBJS\DL*.C
Checking file C:\TC\CLASSLIB\OBJS\DS*.C
Checking file C:\TC\EXAMPLES\BARCHART.C
Checking file C:\TC\EXAMPLES\CPASDEMO.C
Checking file C:\TC\EXAMPLES\GETOPT.C
Checking file C:\TC\EXAMPLES\GREP2MSG.C
Checking file C:\TC\EXAMPLES\HELLO.C
Checking file C:\TC\EXAMPLES\MATHERR.C
Checking file C:\TC\EXAMPLES\PLOTEMP.C
Checking file C:\TC\EXAMPLES\PLOTEMP1.C
Checking file C:\TC\EXAMPLES\PLOTEMP2.C
Checking file C:\TC\EXAMPLES\PLOTEMP3.C
Checking file C:\TC\EXAMPLES\PLOTEMP4.C
Checking file C:\TC\EXAMPLES\PLOTEMP5.C
Checking file C:\TC\EXAMPLES\PLOTEMP6.C
Checking file C:\TC\EXAMPLES\SALESTAG.C
Checking file C:\TC\EXAMPLES\TASM2MSG.C
Checking file C:\TC\EXAMPLES\TCALC*.C
Checking file C:\TC\EXAMPLES\TCALC\TCALC.C
Checking file C:\TC\EXAMPLES\TCALC\TCDISPLY.C
Checking file C:\TC\EXAMPLES\TCALC\TCINPUT.C
Checking file C:\TC\EXAMPLES\TCALC\TCOMMAND.C
Checking file C:\TC\EXAMPLES\TCALC\TCPARSER.C
Checking file C:\TC\EXAMPLES\TCALC\TCUTIL.C

APPENDIX B: SOURCE CODE

(I) INTEGRITY CHECKER

CHECKER.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>
#include<conio.h>
#include <stdarg.h>
#include <dos.h>

#include "sha.h"
#include "endian.h"


/* the SHS block size and message digest sizes, in bytes */

#define SHS_DATASIZE      64
#define SHS_DIGESTSIZE   20

/* SHS f() - functions */

/* #define f1(x,y,z) ( (x & y) | (~x & z) ) // Rounds 0-19 */

#define f1(x,y,z)  ( z^ ( x & ( y ^ z)) )           /* Rounds 0 -19 */
#define f2(x,y,z)  ( x ^ y ^ z)                              /* Rounds 20-39 */

/* #define f3(x,y,z) ( (x & y) |(x & z) | (y & z) ) //Rounds 40-59 */

#define f3(x,y,z) ( (x & y) | ( z & ( x | y ) ) )  /* Rounds 40-59 */
#define f4(x,y,z)        ( x ^ y ^ z )

/* The SHS Mysterious Constants */

#definek1       0x5A827999L
#define k2      0x6ED9EBA1L
#define k3      0x8F1BBCDCL
#define k4      0xCA62C1D6L

/* SHS initial values */
```

```
#defineh0Init  0x67452301L
#define h1Init 0xEFCDAB89L
#defineh2Init  0x98BADCFEL
#defineh3Init  0x10325476L
#define h4Init 0xC3D2E1F0L

/* 32-bit rotate left */

#define ROTL(n,X)   ( ( (X) << n ) | ( (X) >> ( 32 - n ) ) )

#define expand(W,i) ( W[i&15] = ROTL(1, ( W[i&15]  ^ W[(i-14) & 15] ^ \
                                          W[(i-8)&15] ^ W[(i-3)&15] )))
#define subRound(a,b,c,d,e,f,k,data) \
        ( e+= ROTL(5,a) + f(b,c,d) + k + data, b = ROTL(30,b))

void SHAInit( SHA_CTX * shsInfo);
static void SHSTransform( UINT4 * digest, UINT4 * data );
static void longReverse( UINT4 * buffer, int byteCount, int Endianness);
void SHAUpdate( SHA_CTX * shsInfo, BYTE * buffer, int count);
void SHAFinal( BYTE * output, SHA_CTX * shsInfo);
static void SHAtoByte( BYTE * output, UINT4 * input, unsigned int len);
void endianTest( int * endian_ness);


#define SEPARATOR "\\"
#define FILENAME_SIZE FILENAME_MAX
//#define MAXBYTES 10000   //upto 1MB file size

#ifdef __TURBOC__

#include <dir.h>
#define FILE_INFO               struct ffblk
#define FIND_FIRST( n, i )        findfirst( ( n ), ( i ), FA_DIREC )
#define FIND_NEXT( info )         findnext( ( info ) )
#define FILE_NAME( info )         ( ( info ).ff_name )
#define NAME FILE_NAME( fileinfo )

#else

#define FILE_INFO               struct find_t
#define FIND_FIRST( n, i )        _dos_findfirst( (n), _A_SUBDIR, (i) )
#define FIND_NEXT( info )         _dos_findnext( ( info ) )
#define FILE_NAME( info )         ( ( info ).name )
#define NAME FILE_NAME( fileinfo )

#endif
```

```
#define FILE_MAX 100

void ErrorHandler( char *fmt, ... );
unsigned char* CalculateFileChecksum( FILE *file );
void ProcessFile(char* fullname,FILE *checksum_file);
void BuildChecksumFile( char *input_dir_name, char *checksum_file_name );
void CheckFiles( char *checksum_file_name );
void BuildCRCTable( void );

struct ftime ft;

/*
  The main program checks for valid occurrences of the two different types
  of command lines, and executes them if found.
  Otherwise, it prints out a simple usage statement and exits.
 */

int main( int argc, char *argv[] )
 {
   setbuf( stdout, NULL );
   if ( argc == 2 )
        CheckFiles( argv[ 1 ] );
   else if ( argc == 4 && strcmp( argv[ 1 ], "-c" ) == 0 )
        BuildChecksumFile( argv[ 2 ], argv[ 3 ] );
   else {
        printf( "Usage: CHECKER [-c DIR] checksum-file \n" );
        printf( "\n" );
        printf( "Using the -c option CHECKER checks all files under the input DIR\n" );
        printf( "and appends their data to the checksum-file.  Otherwise, the\n" );
        printf( "program checks the Checksum data of all of the files in the\n" );
        printf( "checksum-file and prints the results\n" );
        return( 1 );
   }
   return( 0 );
}

/*
  The routine to check the CHECKSUM values for a list of files just reads in
  a line at a time from the CHECKSUM file. Each line contains a file name and
  a CHECKSUM value.
  The program then just has to calculate the actual CHECKSUM for that
  file, and compare it with the current calculated value.  Any
  mismatch triggers an error message. */
void CheckFiles(char *checksum_file_name)
{
```

```
FILE * _file;
FILE *test_file;
unsigned char* log_checksum;
unsigned char* checksum;
char log_name[ FILENAME_SIZE ];
char modif[FILE_MAX][FILENAME_SIZE];

int result,i,j,k,flag;
static int c=0;
long int filesize;
long int fsize;
int hour,min,sec,month,day,year;
struct ffblk fileinfo;
checksum_file = fopen( checksum_file_name, "rb" );
if ( checksum_file == NULL )
      ErrorHandler( "Couldn't open the log file: %s\n", checksum_file_name );
c=0;
for ( ; ; )
 {
  for(j=0;j<20;j++)
  {
   result = fscanf( Checksum_file,"%02x",&log_checksum[j]);
  }
  if(result<1)
  {
  goto loop_end;
  }
  result = fscanf(checksum_file,"%s %d %d %d",log_name,&hour,&min,&sec);
  result = fscanf(checksum_file,"%d %d %d",&month,&day,&year);
  result = fscanf(checksum_file,"%ld",&fsize);
  if(result<1)
          {
          goto loop_end;
          }
  test_file = fopen( log_name, "rb" );
  if ( test_file != NULL )
          {
           printf("Checking file %s ",log_name);
           checksum= CalculateFileChecksum( test_file );
           filesize=fileinfo.ff_fsize;
           getftime(fileno(test_file), &ft);
           flag=0;
           for(j=0;j<20;j++)
           {
                if(log_checksum[j]!=checksum[j])
                {
```

```
                        flag=1;
                        printf("\nChecksum of this file differs\n");
                    }
                }
            if((ft.ft_hour!=hour)||(ft.ft_min!=min)||(sec!=ft.ft_tsec*2))
                {
                        printf("\nLast modified time differs\n");
                          flag++;
                }
            if((ft.ft_month!=month)||(ft.ft_day!=day)||(year!=(ft.ft_year+1980)))
                    {
                     printf("\nDate of creation of this file differs\n");
                     flag++;
                    }
            if (flag>0)
                {
                    strcpy( modif[c],log_name);
                    printf( "Error: %s has been modified either by user or by some malicious
program\n\n\n",modif[c]);
                    c++;
                    getch();
                }
              else
                    printf( "OK\n" );
            }
     else
             printf( "\nCould not open file %s\n", log_name );
     fclose( test_file );
     }
   loop_end:
   {
     printf(" \n\n\n            ****** Summary ******          ");
    if(c!=0)
          {
                    printf("\n\n       Modified Files are:  \n");
                    for(i=0;i<c;i++)
                    printf("           %s\n",modif[i]);
          }
    else
     printf("\n    ***    All files are intact  ***   \n");
     getch();
   }
getch();
exit(1);
}
```

```
/*
  This routine defers the hard part of directory scanning to a routine
  called ProcessAllFiles(), which takes care of scanning through the
  directory.  That means, here after openning the output Checksum file,just
  start the processing.  This routine also makes sure that the directory
  name passed on the command line is stripped of any trailing '/' or '\'
  character, since people tend to include those when specifying directory
  names.
 */

void BuildChecksumFile(char *input_dir_name,char *checksum_file_name)
{
   char path[ FILENAME_SIZE ];
   FILE *checksum_file;

   strcpy( path, input_dir_name );
   if ( path[ strlen( path ) - 1 ] == SEPARATOR[ 0 ] )
          path[ strlen( path ) - 1 ] = '\0';
   checksum_file = fopen( checksum_file_name, "a" );
   if ( checksum_file == NULL )
          ErrorHandler( "Can't open checksum log file: %s\n", checksum_file_name );
   filerecus( path, checksum_file );
}

/*
  This routine is responsible for actually performing the
  calculation of the 160 bit Checksum for the entire file.
  The actual calculation consists of reading in blocks of 512 bytes at a
  time from the file, then updating the Checksum with the value for that
  block. The checksum of a file is calculated by using SHA-1 algo,that
  generates 160 bit(20 Bytes) checksum.
 */

unsigned char*CalculateFileChecksum(FILE *file)
{
   int count;
   unsigned char buffer[512];
   unsigned char*c;
   int i;
   SHA_CTX sha;
   unsigned char *checksum;
   i = 0;

   SHAInit(&sha);
   for ( ; ; )
     {
```

```
            count = fread( buffer, 1, 512, file );
            if ( ( i++ % 32 ) == 0 )
                putc( '.', stdout );
            if ( count == 0 )
                break;
            SHAUpdate(&sha,buffer,count);
            SHAFinal(checksum,&sha);
      }
    putc( ' ', stdout );
    return( checksum);
}

/*
 This is the routine that is responsible for calculating all of
 the CHECKSUM values for the files in a given directory.  The CHECKSUM
 values and the file names are written out to the checksum_file.
 This routine sits in a loop for each directory, opening each file
 and processing it.  Before a file is opened, a check is made to
 see if the file is actually a directory.  If it turns out that
 the file is a directory, a new path name is constructed, and this
 routine calls itself recursively so that all the files in the
 subdirectory are also processed.
 */

 /***** function to access all the files in specified directory *****/
filerecus(char*filemask,FILE *checksum_file)
{
struct ffblk fileinfo; // ffblk is DOS file control block structure
char path[256],drive[5],dir[256],name[14],ext[5],tempdir[256];
/*first files present in directory*/
if(findfirst(filemask,&fileinfo,39)!=-1) //findfirst search a disk directory for files
{
 fnsplit(filemask,drive,dir,name,ext);
 /*fnsplit takes a file's full path name as a string, split
the name into its four components, then store those components.*/
 strcpy(path,drive);
 strcat(path,dir);
 strcat(path,fileinfo.ff_name);
 ProcessFile(path,checksum_file);
 while(findnext(&fileinfo)!=-1)//findnext continue the search in a disk directory for files
 {
   fnsplit(filemask,drive,dir,name,ext);
   strcpy(path,drive);
   strcat(path,dir);
   strcat(path,fileinfo.ff_name);
   ProcessFile(path,checksum_file);
```

```
 }
}

/*now search for subdirectories*/
fnsplit(filemask,drive,dir,name,ext);
//split a given path with fnsplit, then merge the resultant components with
//fnmerge,end up with path
fnmerge(path,drive,dir,"*",".");  // " '*' and '.' " means wildcard directory entry
if(findfirst(path,&fileinfo,FA_DIREC)==0)//;
{
if(strcmp(fileinfo.ff_name,".")&&strcmp(fileinfo.ff_name,"..")&&fileinfo.ff_attrib==F
A_DIREC)
 {
  strcpy(tempdir,dir);
  strcat(tempdir,fileinfo.ff_name);
  strcat(tempdir,"\\");
  fnmerge(path,drive,tempdir,name,ext);
  //path created in present directory
  filerecus(path,checksum_file);
 }
 while(findnext(&fileinfo)!=-1)
 {

if(strcmp(fileinfo.ff_name,".")&&strcmp(fileinfo.ff_name,"..")&&fileinfo.ff_attrib==F
A_DIREC)
  {
   strcpy(tempdir,dir);
  strcat(tempdir,fileinfo.ff_name);
  strcat(tempdir,"\\");
  fnmerge(path,drive,tempdir,name,ext);
  //path created in present directory
  filerecus(path,checksum_file);
 }
}
}
return;
}//end function filerecus

//void ProcessAllFiles( char *path,FILE *checksum_file)
void ProcessFile(char* fullname,FILE *checksum_file)
{

    FILE_INFO fileinfo;
    int done;
    long int fsize;
    int hour,min,sec,month,day,year;
```

```
    unsigned char *checksum;
    FILE *file;
    int i;

    file = fopen( fullname, "rb" );
    if ( file != NULL )
          {
            printf( "Scanning %s ", fullname );
            getch();
            checksum=CalculateFileChecksum( file );
            getftime(fileno(file), &ft);
            fsize=(long int)fileinfo.ff_fsize;
            putc( '\n', stdout );
            for(i=0;i<20;i++)
              {
                    fprintf( checksum_file, "%02x", checksum[i] );
              }

            hour=(int)ft.ft_hour;
            min=(int)ft.ft_min;
            sec=(int)(ft.ft_tsec * 2);
            month=(int)ft.ft_month;
            day=(int)ft.ft_day;
            year=(int)(ft.ft_year+1980);
            fprintf( checksum_file, "%s", fullname);
            fprintf(checksum_file," %d %d %d",hour,min,sec);
            fprintf(checksum_file," %d %d %d",month,day,year);
            fprintf(checksum_file," %ld\n" ,fsize);
            fclose( file );
          }
    else
      printf( "Could not open %s!\n", fullname );
}

/*
 * The fatal error handler just has to print out a formatted error
 * message and then exit.*/

void ErrorHandler( char *fmt, ... )
{
    va_list argptr;
    va_start( argptr, fmt );
    printf( "Error: " );
    vprintf( fmt, argptr );
    va_end( argptr );
    exit( -1 );
```

```
}

/* Initialize the SHS values */

void SHAInit( SHA_CTX * shsInfo)
{
        endianTest(&shsInfo->Endianness);
        /* set the h-vars to their initial values */
        shsInfo->digest[0] = h0Init;
        shsInfo->digest[1] = h1Init;
        shsInfo->digest[2] = h2Init;
        shsInfo->digest[3] = h3Init;
        shsInfo->digest[4] = h4Init;

        /* initialize the bit count */

        shsInfo->countLo=shsInfo->countHi = 0;
}

/* Perform the SHS Transformation */

static void SHSTransform( UINT4 * digest, UINT4 * data )
{
        UINT4 A,B,C,D,E; /* Local Vars */
        UINT4 eData[16]; /* Expanded Data */

        /* set up the first buffer and local data buffer */
        A = digest[0];
        B = digest[1];
        C = digest[2];
        D = digest[3];
        E = digest[4];

        memcpy( (POINTER)eData, (POINTER)data, SHS_DATASIZE);

        /* heavy mangling in 4 sub rounds of 20 iterations each */

        subRound(A,B,C,D,E,f1,k1,eData[0]);
        subRound(E,A,B,C,D,f1,k1,eData[1]);
        subRound(D,E,A,B,C,f1,k1,eData[2]);
        subRound(C,D,E,A,B,f1,k1,eData[3]);
        subRound(B,C,D,E,A,f1,k1,eData[4]);
        subRound(A,B,C,D,E,f1,k1,eData[5]);
        subRound(E,A,B,C,D,f1,k1,eData[6]);
        subRound(D,E,A,B,C,f1,k1,eData[7]);
        subRound(C,D,E,A,B,f1,k1,eData[8]);
```

```
subRound(B,C,D,E,A,f1,k1,eData[9]);
subRound(A,B,C,D,E,f1,k1,eData[10]);
subRound(E,A,B,C,D,f1,k1,eData[11]);
subRound(D,E,A,B,C,f1,k1,eData[12]);
subRound(C,D,E,A,B,f1,k1,eData[13]);
subRound(B,C,D,E,A,f1,k1,eData[14]);
subRound(A,B,C,D,E,f1,k1,eData[15]);
subRound(E,A,B,C,D,f1,k1,expand(eData,16));
subRound(D,E,A,B,C,f1,k1,expand(eData,17));
subRound(C,D,E,A,B,f1,k1,expand(eData,18));
subRound(B,C,D,E,A,f1,k1,expand(eData,19));


subRound(A,B,C,D,E,f2,k2,expand(eData,20));
subRound(E,A,B,C,D,f2,k2,expand(eData,21));
subRound(D,E,A,B,C,f2,k2,expand(eData,22));
subRound(C,D,E,A,B,f2,k2,expand(eData,23));
subRound(B,C,D,E,A,f2,k2,expand(eData,24));
subRound(A,B,C,D,E,f2,k2,expand(eData,25));
subRound(E,A,B,C,D,f2,k2,expand(eData,26));
subRound(D,E,A,B,C,f2,k2,expand(eData,27));
subRound(C,D,E,A,B,f2,k2,expand(eData,28));
subRound(B,C,D,E,A,f2,k2,expand(eData,29));
subRound(A,B,C,D,E,f2,k2,expand(eData,30));
subRound(E,A,B,C,D,f2,k2,expand(eData,31));
subRound(D,E,A,B,C,f2,k2,expand(eData,32));
subRound(C,D,E,A,B,f2,k2,expand(eData,33));
subRound(B,C,D,E,A,f2,k2,expand(eData,34));
subRound(A,B,C,D,E,f2,k2,expand(eData,35));
subRound(E,A,B,C,D,f2,k2,expand(eData,36));
subRound(D,E,A,B,C,f2,k2,expand(eData,37));
subRound(C,D,E,A,B,f2,k2,expand(eData,38));
subRound(B,C,D,E,A,f2,k2,expand(eData,39));


subRound(A,B,C,D,E,f3,k3,expand(eData,40));
subRound(E,A,B,C,D,f3,k3,expand(eData,41));
subRound(D,E,A,B,C,f3,k3,expand(eData,42));
subRound(C,D,E,A,B,f3,k3,expand(eData,43));
subRound(B,C,D,E,A,f3,k3,expand(eData,44));
subRound(A,B,C,D,E,f3,k3,expand(eData,45));
subRound(E,A,B,C,D,f3,k3,expand(eData,46));
subRound(D,E,A,B,C,f3,k3,expand(eData,47));
subRound(C,D,E,A,B,f3,k3,expand(eData,48));
subRound(B,C,D,E,A,f3,k3,expand(eData,49));
subRound(A,B,C,D,E,f3,k3,expand(eData,50));
```

```
        subRound(E,A,B,C,D,f3,k3,expand(eData,51));
        subRound(D,E,A,B,C,f3,k3,expand(eData,52));
        subRound(C,D,E,A,B,f3,k3,expand(eData,53));
        subRound(B,C,D,E,A,f3,k3,expand(eData,54));
        subRound(A,B,C,D,E,f3,k3,expand(eData,55));
        subRound(E,A,B,C,D,f3,k3,expand(eData,56));
        subRound(D,E,A,B,C,f3,k3,expand(eData,57));
        subRound(C,D,E,A,B,f3,k3,expand(eData,58));
        subRound(B,C,D,E,A,f3,k3,expand(eData,59));

        subRound(A,B,C,D,E,f4,k4,expand(eData,60));
        subRound(E,A,B,C,D,f4,k4,expand(eData,61));
        subRound(D,E,A,B,C,f4,k4,expand(eData,62));
        subRound(C,D,E,A,B,f4,k4,expand(eData,63));
        subRound(B,C,D,E,A,f4,k4,expand(eData,64));
        subRound(A,B,C,D,E,f4,k4,expand(eData,65));
        subRound(E,A,B,C,D,f4,k4,expand(eData,66));
        subRound(D,E,A,B,C,f4,k4,expand(eData,67));
        subRound(C,D,E,A,B,f4,k4,expand(eData,68));
        subRound(B,C,D,E,A,f4,k4,expand(eData,69));
        subRound(A,B,C,D,E,f4,k4,expand(eData,70));
        subRound(E,A,B,C,D,f4,k4,expand(eData,71));
        subRound(D,E,A,B,C,f4,k4,expand(eData,72));
        subRound(C,D,E,A,B,f4,k4,expand(eData,73));
        subRound(B,C,D,E,A,f4,k4,expand(eData,74));
        subRound(A,B,C,D,E,f4,k4,expand(eData,75));
        subRound(E,A,B,C,D,f4,k4,expand(eData,76));
        subRound(D,E,A,B,C,f4,k4,expand(eData,77));
        subRound(C,D,E,A,B,f4,k4,expand(eData,78));
        subRound(B,C,D,E,A,f4,k4,expand(eData,79));

        /* Build Message Digest */

        digest[0]+=A;
        digest[1]+=B;
        digest[2]+=C;
        digest[3]+=D;
        digest[4]+=E;
}

/* when run on a little endian CPU we need to perform byte reversal on an
        array of long word */

static void longReverse( UINT4 * buffer, int byteCount, int Endianness)
{
        UINT4 value;
```

```
        if (Endianness == 1)
                return;
        byteCount /= sizeof(UINT4);

        while( byteCount-- )
        {
                value = *buffer;
                value = ( (value & 0xFF00FF00L) >> 8 ) | \
                        ( (value & 0x00FF00FFL) << 8 );
                *buffer++ = (value << 16) | (value >> 16);
        }
}

/*Update SHS for a block of data */

void SHAUpdate( SHA_CTX * shsInfo, BYTE * buffer, int count)
{
 UINT4 tmp;
 int dataCount;
 /* Update bitcount */
 tmp = shsInfo->countLo;
 if ( (shsInfo->countLo = tmp + ((UINT4) count << 3)) < tmp)
         shsInfo->countHi++;
 shsInfo->countHi += count >> 29;

 /* Get count of bytes already in data */
 dataCount = (int) (tmp >> 3) & 0x3F;
 /* Handle any leading odd-sized chunks*/
 if(dataCount)
 {
  BYTE * p = (BYTE *) shsInfo->data + dataCount;
  dataCount = SHS_DATASIZE - dataCount;

  if(count < dataCount)
  {
   memcpy(p,buffer,count);
   return;
  }
  memcpy(p,buffer,dataCount);
  longReverse(shsInfo->data, SHS_DATASIZE, shsInfo->Endianness);
  SHSTransform(shsInfo->digest,shsInfo->data);
  buffer += dataCount;
  count -= dataCount;
 }
 while( count >= SHS_DATASIZE)
 {
```

```
  memcpy( (POINTER)shsInfo->data,(POINTER)buffer,SHS_DATASIZE);
  longReverse(shsInfo->data,SHS_DATASIZE,shsInfo->Endianness);
  SHSTransform(shsInfo->digest,shsInfo->data);
  buffer+=SHS_DATASIZE;
  count-=SHS_DATASIZE;
 }
 /* Handle any remaining bytes of data */
 memcpy( (POINTER)shsInfo->data,(POINTER)buffer,count);
}

/* Final wrapup - pad to SHS_DATASIZE-byte boundary with the bit pattern */
/* 1 0*  (64-bit count of the bits processed, MSB_FIRST) */

void SHAFinal( BYTE * output, SHA_CTX * shsInfo)
{
        int count;
        BYTE * dataPtr;

        /* compute the number of bytes mod 64 */
        count = (int) shsInfo->countLo;
        count = (count >> 3) & 0x3F;

        /* set the first char of padding to 0x80, this is safe since there is
           at least one byte free */

        dataPtr = (BYTE *) shsInfo->data + count;
        *dataPtr++=0x80;

        /* Bytes of padding needed to make 64 bytes */

        count = SHS_DATASIZE - 1 - count;

   /* Pad out to 56 mod 64 */

        if ( count < 8 )
        {
                /* Two lots of padding : pad the first block to 64 bytes */
                memset(dataPtr,0,count);
                longReverse(shsInfo->data,SHS_DATASIZE,shsInfo->Endianness);
                SHSTransform(shsInfo->digest,shsInfo->data);
                /* now fill the next block with 56 bytes */
                memset( (POINTER) shsInfo->data,0,SHS_DATASIZE-8);
        }
        else
                /* pad block to 56 bytes */
                memset(dataPtr,0,count-8);
```

```
        /* append length in bits and transform */
        shsInfo->data[14] = shsInfo->countHi;
        shsInfo->data[15] = shsInfo->countLo;

        longReverse(shsInfo->data, SHS_DATASIZE - 8, shsInfo->Endianness);
        SHSTransform(shsInfo->digest, shsInfo->data);

        /* output to an array of bytes */

        SHAtoByte(output, shsInfo -> digest, SHS_DIGESTSIZE);

        /* Zeroise sensitive stuff */

        memset( (POINTER)shsInfo, 0,sizeof(shsInfo));
}

static void SHAtoByte( BYTE * output, UINT4 * input, unsigned int len)
{
        /* output SHA digest in byte array */
        unsigned int i,j;

        for (i=0, j =0; j < len; i++, j+= 4)
        {
         output[j+3] = (BYTE) (input[i] & 0xff);
         output[j+2] = (BYTE) ( (input[i] >> 8)  & 0xff);
         output[j+1] = (BYTE) ( (input[i] >> 16) & 0xff);
         output[j] = (BYTE) ( (input[i] >> 24) & 0xff);
        }
}
void endianTest( int * endian_ness)
{
 if ( (*(unsigned short *)("#S") >> 8) == '#' )
 {
  *endian_ness  = !(0);
 }
 else
 {
  *endian_ness  = 0;
 }
}
/************************* End of CHECKER.C
*************************/
```

I. SCANNER

(a) DATABASE.C

This program is used to enter the virus details in file SIGNVAR.DAT.

/******** DATABASE ENTRY PROGRAM **************/**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>

#define MAXSTRING 50
#define MAXTXT 60

main()
{
 char sign[MAXSTRING],sign1[MAXSTRING];
 char temp[MAXTXT],temp1[MAXTXT];
 char descrip[MAXTXT];
 unsigned chr,chr1;
 FILE *fp;
 int siglen,count=0,type;
 int j,len;
 int g,z, c,ch,p,m,blk;
 int n;

 printf("**********    DATABASE ENTRY PROGRAM **************\n");
 printf("Every virus record will have three fields:\n   1. Signature in HEX\n   2. Type of the virus \n   3. Description of virus");
```

```
/* open signature log file to append*/

if((fp=fopen("signvir.dat","a+b"))==NULL)
{
 printf("ERROR:unable to open signvir.dat file\n");
 getch();
 exit(1);
}

while(1)
{
 count=0;
 blk=0;

 while(1)
 {
  printf("\nEnter the signature of virus in HEXADECIMAL (without comma and blank
spaces):\n");
  scanf("%s",sign1);
  blk=0;
  for(m=0;m<MAXSTRING;m++)
  {
   if(sign1[m]>32)
   {
    blk++;
   }
  }
  if(blk==0)
  {
   printf("\nERROR:Signature should be in HEX without comma and blank
spaces\n");
```

```
    getch();
   }
   else
   {
    break;
   }
  }

  /* error check for valid signatures*/
  strcpy(sign,sign1);
  for(j=0;j<strlen(sign);j++)
  {
   if(!((sign[j]>='0'&&sign[j]<='9')||
          (sign[j]>='A'&&sign[j]<='F')||
          (sign[j]>='a'&&sign[j]<='f')||
          sign[j]==' '))
           {
            printf("\nERROR:Signature should be in HEX without comma and blank
spaces\n");
            getch();
            exit(1);
           }
   }

  /*Record 2:Type of virus*/

  while(1)
  {
  printf("Enter the TYPE of Virus (P:partition table,B:Boot sector,F:File virus):");
  type=toupper(getch());
  if(type=='P'||type=='B'||type=='F')
```

```
  {
   printf("\n Type of virus is:%c\n",type);
   break;
  }
 }

 //Record 3:Description of the virus
printf("Enter 1 line description of the virus\n");
c=getchar();
z=0;
temp[z]=c;
while((c=getchar())!='\n')
{
z++;
temp[z]=c;
}
printf("\nAre the details regarding this virus correct?(y/n):");
while(1)
 {
  ch=toupper(getch());
  if(ch=='N'||ch=='Y')
  {
   break;
  }
 }
 if(ch=='Y')
 {
  for(p=0;p<MAXTXT;p++)
    {
     descrip[p]=temp[p];
    }
```

```
    count=strlen(sign);
   fwrite(sign,MAXSTRING,1,fp);
   fputc(count,fp);
   fputc(type,fp);
   fwrite(descrip,MAXTXT,1,fp);
    }
  printf("\nDo you wish to continue?(y/n):");
  while(1)
  {
   ch=toupper(getch());
   if(ch=='N'||ch=='Y')
   {
    break;
   }
  }
  if(ch=='N')
  {
   break;
  }
}
return 0;
}
```

(b) SCANNER.C

```
/*************** VIRUS SCANNER *********************/

/****** It uses the Signature Scanning Tecnique to detect the presence of viruses in
executables and fast string matching Boyer-Moore-Horspool (BMH) algorithm
**********/

#include<stdio.h>
#include<mem.h>
#include<io.h>
#include<dos.h>
#include<bios.h>
#include<alloc.h>
#include<dir.h>
#include<fcntl.h>
#include<time.h>

#define MAXBYTES 10000
#define MAXSTRING  50
#define MAXTXT     60
#define MAXSIG     100
#define SECTSIZE  512
#define MAXCHAR 256

struct VIRUSINFO {
                char signature[MAXSTRING];//signature in HEX
                char xcount;          //No of characters in signature
                char type;            //Type of the virus
               };


struct VIRUSINFO *virus;
int sigcount;
char firstchar[MAXSIG];
FILE *fps;

void main(int argc,char *argv[])
{
 int count,drive=0,harddisk=1;
 int s;
 char filemask[MAXPATH]; //predefined MAXPATH: Complete file name with path

 strcpy(filemask,"*.EXE"); /*If the type of file has not been specified in option then by
default will search executable files.*/
```

```
 if (argc<2)
 {
  printf("  ERROR:Please run the program through command prompt and then\n  follow
the menu given below:\n\n");
  printf("       1.Enter -B<drive no> to check the Boot Sector\n");
  printf("       2.Enter -P<hard disk no> to check the Partition Table\n");
  printf("       3.Enter -F<file specification> to check the files\n");
  getch();
  exit(1);
 }

 if(( virus =malloc(sizeof(struct VIRUSINFO)*MAXSIG))==NULL)
 {
  printf("ERROR: Unable to reserve space for the signatures\n");
  getch();
  exit();
 }

 if((fps=fopen("signvir.dat","rb"))==NULL)
 {
  printf("ERROR: Unable to open SIGNVIR.DAT-Signature Log File\n");
  getch();
  exit();
 }

 sigcount=0;


 /*       &virus[sigcount] : Points to a block into which data is read
    sizeof(virus[sigcount]) : Length of each item read, in bytes
                          1 : Number of items read
                        fps: Points to input stream */
//read the signatures
 while(fread(&virus[sigcount],sizeof(virus[sigcount]),1,fps))
 {
  fseek(fps,60l,SEEK_CUR); //skip description of current signature(60 bytes)to read the
next signature
  firstchar[sigcount]=virus[sigcount].signature[0];
  sigcount++;
 }
 printf("\n  No of signatures in database : %d\n",sigcount);

//parse the command line
 for(count=1;count<argc;count++)
 {
 //check for correct arguments
```

```
 //memchr searches the first 6 bytes of the block "bpfBPF" for argv[count][1]
 if(argv[count][0]!='-'||!memchr("bpfBPF",argv[count][1],6))
 {
  printf("ERROR: Options(-B,-P,-F<filename>) expected.\n");
  getch();
  exit();
 }

 //boot sector checking
 if(argv[count][1]=='B'||argv[count][1]=='b')
 {
  if(argv[count+1][0]!='-'&&(count+1)<argc)
  {
   count++;
   drive=toupper(argv[count][0])-65;
  }
  scanboot(drive);
 }

 //partition table checking
 if(argv[count][1]=='P'||argv[count][1]=='p')
 {
  if(argv[count+1][0]!='-'&&(count+1)<argc)
  {
   count++;
   harddisk=atoi(argv[count]);
  }
  scanpart(harddisk);
 }

 //File checking
 if(argv[count][1]=='F'||argv[count][1]=='f')
 {
  if(argv[count+1][0]!='-'&&(count+1)<argc)
  {
   count++;
   strcpy(filemask,argv[count]);
   printf("\nType of files to be scanned is %s\n",filemask);
  }
   scanfile(filemask);
 }

} //end for

fclose(fps);
}
```

```
//******** End Main() ********

/*********function definitions called in main functions***********

/***********  Check the Boot sector *************/
/*this function reads the boot sector of given drive and
then calls scansector function to do actual checking*/
scanboot(int drive)
{
 char buffer[SECTSIZE];

 printf("\n\n Checking Boot Sector : %c (Boot sector of Floppy)...........\n",drive+65);
 if(absread(drive,1,0,buffer)==-1)
 {
  printf("ERROR: Unable to read boot sector of drive %c\nPress any key to
exit...\n",drive+65);
  getch();
  exit(1);
 }
 scansector('B',buffer);
 return;
}

/********** Check the Partition Table ************/
/*this function reads the partition table of given hard disk and
then calls scansector function to do actual checking*/
scanpart(int harddisk)
{
 char buffer[SECTSIZE];

 printf("\n\n Checking partiton table of specified Hard Disk: %d\n",harddisk);
 if(biosdisk(2,harddisk+0x7F,0,0,1,1,buffer)!=0)
 {
  printf("ERROR: Unable to read the partition table of Hard Disk:%d\nPress any key to
exit...\n",harddisk);
  getch();
  exit(1);
 }
 scansector('P',buffer);
 return;
}

/************** Check the files *****************/
/*this function calls function filerecus for a given file type*/
scanfile(char *filemask)
{
```

```
printf("\n\nChecking %s files.......\n",filemask);
 start= clock();
 filerecus(filemask);
  getch();
 return;
}


/*********** Functon called by scanboot and scanpart functions,
                 it scans contents of a particular sector  ************/
scansector(char type,char *buffer)
{
 int j,k,flag;
 unsigned i;
 char string[MAXSTRING];

 for(i=0;i<SECTSIZE;i++)
 {
  if(memchr(firstchar,buffer[i],sigcount))
  {
   for(k=0;k<sigcount;k++)
   {
    if(virus[k].type==type)
    {
     memcpy(string,virus[k].signature,MAXSTRING);
     for(j=0;j<virus[k].xcount&&(string[j]==buffer[i+j]);j++);
     if(j==virus[k].xcount)
     {
      printf("\n ****** VIRUS DETECTED ******\n Signature (%s) of found at offset
%u.\n ",string,i);
      showdescrip(k);
      return;
     }
    }
   }
  }
 }
 printf("\n*** No viruses present ***\n");
 return;
}

/***** Function to show the description of the detected virus *****/
showdescrip(int virnum)
{
 char desc[MAXTXT];
 int i;
```

```
 fseek(fps,(long)(virnum*(MAXSTRING+2+MAXTXT)),SEEK_SET);
 fseek(fps,(long)(MAXSTRING+2),SEEK_CUR);
 fread(desc,MAXTXT,1,fps);
 printf("\n Name of Virus:%s",desc);
 printf("\n");
 return;
}

/***** function to access all the files in specified directory *****/
filerecus(char*filemask)
{
struct ffblk fileinfo; // ffblk is DOS file control block structure
char path[256],drive[5],dir[256],name[14],ext[5],tempdir[256];
/*first files present in directory*/
if(findfirst(filemask,&fileinfo,39)!=-1) //findfirst search a disk directory for files
{
 fnsplit(filemask,drive,dir,name,ext);
 strcpy(path,drive);
 strcat(path,dir);
 strcat(path,fileinfo.ff_name);
 printf("\nChecking file %s",path);
 filechk(path);
 while(findnext(&fileinfo)!=-1)//findnext continue the search in a disk directory for files
 {
   fnsplit(filemask,drive,dir,name,ext);
   strcpy(path,drive);
   strcat(path,dir);
   strcat(path,fileinfo.ff_name);
   printf("\nChecking file %s",path);
   filechk(path);
 }
}

/*now search for subdirectories*/
fnsplit(filemask,drive,dir,name,ext);
fnmerge(path,drive,dir,"*",".");  // " '*' and '.' " means wildcard directory entry
if(findfirst(path,&fileinfo,FA_DIREC)==0)//;
{

if(strcmp(fileinfo.ff_name,".")&&strcmp(fileinfo.ff_name,"..")&&fileinfo.ff_attrib==F
A_DIREC)
 {
  strcpy(tempdir,dir);
  strcat(tempdir,fileinfo.ff_name);
  strcat(tempdir,"\\");
  fnmerge(path,drive,tempdir,name,ext);
```

```
 filerecus(path);
 }
 while(findnext(&fileinfo)!=-1)
 {

if(strcmp(fileinfo.ff_name,".")&&strcmp(fileinfo.ff_name,"..")&&fileinfo.ff_attrib==F
A_DIREC)
  {
   strcpy(tempdir,dir);
  strcat(tempdir,fileinfo.ff_name);
  strcat(tempdir,"\\");
  fnmerge(path,drive,tempdir,name,ext);
  filerecus(path);
 }
}
}
return;
}//end function filerecus

/***** searches the presence of a particular virus signature string in a file *****/
filechk(char*filename)
{
 FILE*fp;
 int k,flag=0,count,i,j;
 char string[MAXSTRING];
 unsigned char buffer[1024];
 fp=fopen(filename,"rb");
 if(fp!=NULL)
 {
   j = 0;
   for ( ; ; )
     {
      count = fread( buffer, 1, 1024, fp);
      if ( ( j++ % 100 ) == 0 )
              putc( '.', stdout );
      if ( count == 0 )
              break;
      for(k=0;k<sigcount;k++)
              {
               if(virus[k].type=='F')
                     {
                       memcpy(string,virus[k].signature,MAXSTRING);
                       flag =bmh(buffer,count,string);
                       if(flag!=0)
                         {
```

```
                                printf("\n****** Virus Detected ******\n Signature (%s) found
at offset %d in %s file.",string,flag,filename);
                                showdescrip(k);
                                fclose (fp);
                                return;
                           }
                      }
               }
  }
  putc( ' ', stdout );
  fclose(fp);
}
else
{
 printf("ERROR:unable to open file %s\n",filename);
 getch();
}
return;
}
```

/* This function implements **Boyer-Moore-Horsepool** a fast string matching algorithm*/

```
int bmh(char* buffer,int n,char string[30])
{
  int j=0,i;
  int d[MAXCHAR],m,k;
  m=strlen(string);
 for(i=0;i<MAXCHAR;i++)
 d[i]=m;
 for( i=0;i<m-1;i++)
  {
     d[string[i]]=m-i-1;
  }
  i=m-1;
for(i=m-1;i<n;i+=d[buffer[i]&(MAXCHAR-1)])
{
 for(k=m-1,j=i;k>=0&&buffer[j]==string[k];k--)
   j--;
   if(k==(-1))
    return(j+1);
}
return 0;
}
```

In this dissertation performance of BMH algorithm has been compared with sequential, Boyer Moore and Turbo Boyer Moore algorithms, whose implementation has been given here.

(c) Sequential algorithm

```
filechk(char*filename)
{
 FILE*fp;
 int k,count,i,j,m;
 char string[MAXSTRING];
 unsigned char buffer[ 1024 ];
  char desc[MAXTXT];
   FILE*sum;

 fp=fopen(filename,"rb");
 if(fp==NULL)
 {
 printf("ERROR: unable to open file %s", filename);
 getch();
 }
 else
 {
   m= 0;
    for ( ; ; )
      {
       count = fread( buffer, 1, 1024, fp);
       m++;
       if ( count == 0 )
               break;
       for(i=0;i<count;i++)
            {
             if(memchr(firstchar,buffer[i],sigcount))
             {
               for(k=0;k<sigcount;k++)
                 {
                  if(virus[k].type=='F')
                        {
                          memcpy(string,virus[k].signature,MAXSTRING);
                          for(j=0;j<virus[k].xcount&&(string[j]==buffer[i+j]);j++);
                          if(j==virus[k].xcount)
                           {
                            printf("\n ****** Virus Detected ****** \n Signature (%s) found in
%s \n at offset %u",
                            string,filename, (1024*(m-1)+i));
                            showdescrip(k);
```

```
                    }
                }
            }
        }
    }
}
putc( ' ', stdout );
fclose(fp);
}
return;
}
```

(d) Boyer More Algorithm

```
void preBmBc(char *x, int m, int bmBc[])
{
  int i;

  for (i = 0; i < ASIZE; ++i)
    bmBc[i] = m;
  for (i = 0; i < m - 1; ++i)
    bmBc[x[i]] = m - i - 1;
}

void suffixes(char *x, int m, int *suff)
{
  int f, g, i;

  suff[m - 1] = m;
  g = m - 1;
  for (i = m - 2; i >= 0; --i)
  {
    if (i > g && suff[i + m - 1 - f] < i - g)
         suff[i] = suff[i + m - 1 - f];
    else
    {
         if (i < g)
            g = i;
         f = i;
         while (g >= 0 && x[g] == x[g + m - 1 - f])
            --g;
         suff[i] = f - g;
    }
  }
}
```

```
void preBmGs(char *x, int m, int bmGs[])
{
  int i, j, suff[XSIZE];

  suffixes(x, m, suff);

  for (i = 0; i < m; ++i)
    bmGs[i] = m;
  j = 0;
  for (i = m - 1; i >= -1; --i)
    if (i == -1 || suff[i] == i + 1)
        for (; j < m - 1 - i; ++j)
          if (bmGs[j] == m)
            bmGs[j] = m - 1 - i;
  for (i = 0; i <= m - 2; ++i)
    bmGs[m - 1 - suff[i]] = m - 1 - i;
}

int Boyer_Moore(char *x, int m, char *y, int n)
{
  int i, j, bmGs[XSIZE], bmBc[ASIZE];

  /* Preprocessing */

  preBmGs(x, m, bmGs);
  preBmBc(x, m, bmBc);

  /* Searching */

  j = 0;
  while (j <= n - m)
   {
    for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
    if (i < 0)
    {
        return(j);
    }
    else
        j += max(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
  }
  return (0);
}
```

(e) Turbo Boyer Moore Algorithm

```
void preBmBc(char *x, int m, int bmBc[])
{
  int i;

  for (i = 0; i < ASIZE; ++i)
    bmBc[i] = m;
  for (i = 0; i < m - 1; ++i)
    bmBc[x[i]] = m - i - 1;
}


void suffixes(char *x, int m, int *suff)
{
  int f, g, i;

  suff[m - 1] = m;
  g = m - 1;
  for (i = m - 2; i >= 0; --i)
  {
    if (i > g && suff[i + m - 1 - f] < i - g)
         suff[i] = suff[i + m - 1 - f];
    else
    {
         if (i < g)
           g = i;
         f = i;
         while (g >= 0 && x[g] == x[g + m - 1 - f])
           --g;
         suff[i] = f - g;
    }
  }
}

void preBmGs(char *x, int m, int bmGs[])
{
  int i, j, suff[XSIZE];

  suffixes(x, m, suff);

  for (i = 0; i < m; ++i)
    bmGs[i] = m;
  j = 0;
  for (i = m - 1; i >= -1; --i)
    if (i == -1 || suff[i] == i + 1)
```

```
        for (; j < m - 1 - i; ++j)
          if (bmGs[j] == m)
            bmGs[j] = m - 1 - i;
  for (i = 0; i <= m - 2; ++i)
    bmGs[m - 1 - suff[i]] = m - 1 - i;
}


int Turbo_Boyer_Moore(char *x, int m, char *y, int n)
{
  int bcShift, i, j, shift, u, v, turboShift,
      bmGs[XSIZE], bmBc[ASIZE];

  /* Preprocessing */
  preBmGs(x, m, bmGs);
  preBmBc(x, m, bmBc);

  /* Searching */
  j = u = 0;
  shift = m;
  while (j <= n - m)
  {
    i = m - 1;
    while (i >= 0 && x[i] == y[i + j])
    {
        --i;
        if (u != 0 && i == m - 1 - shift)
          i -= u;
    }
    if (i < 0)
    {
        shift = bmGs[0];
        u = m - shift;
        return(j);
    }
    else
    {
        v = m - 1 - i;
        turboShift = u - v;
        bcShift = bmBc[y[i + j]] - m + 1 + i;
        shift = max(turboShift, bcShift);
        shift = max(shift, bmGs[i]);
        if (shift == bmGs[i])
          u = min(m - shift, v);
        else
        {
```

```
          if (turboShift < bcShift)
            shift = max(shift, u + 1);
          u = 0;
         }
    }
    j += shift;
  }
  return(0);
}
```