The .2880 and .2886 variants, after displaying 'HKs Vtech', also say 'using VTME v1.11' and 'Please ask S-S Chan to help you!'. These are polymorphic, and it seems likely that VTME is the name of the mutation engine developed for this family.

MING

Ming.1017 was first found in January 1994 and contains the text message: 'Nice To Meet You! Copyright(c) 1-11-1993 By Ming. From Tuen Mun, Hong Kong Version 3.10'.

It is a simple, overwriting file virus and is only notable because a minor variant reached Lapland two weeks later on video driver diskettes which had originated in Hong Kong. Even something as obvious as an overwriting virus can spread internationally with a lucky break.

In April, messages appeared on the local virus echo claiming to be from the Ming who had written this virus. He said that he was changing his name to Crazy_Lord and he that had written CLME, the Crazy Lord Mutation Engine. Different messages mentioned versions 0.5 and 0.6. He also thanked 'HKs Vtech' for assistance and challenged 'S-S Chan' to kill it.

It was July before a sample of a CLME virus was obtained. The decryptor used a simple XOR loop with a variable amount of junk code.

Ming.CLME.1952, Ming.1017 and Ming.CLME.1528 increased in samples uploaded to my BBS during the following months and are still occasionally uploaded now. A total of five Ming viruses are currently known.

The author appears to be a novice programmer, improving his skills with practice. One can only guess at the collaboration between this person and the author of the Jerusalem.HK family.

SHUTDOWN

Shutdown.644 appeared in January 1994. It contains the text message:

> Computers must be shut down to dedicate my sister

Reports of it and a second variant, .698, have appeared sporadically in the BBSs until recently.

CHINESE LANGUAGE VIRUSES

Two other viruses known in Hong Kong BBSs are notable because they display, or attempt to display, a message in Chinese characters.

Jerusalem.J, which first appeared in October 1994, uses DOS calls to redefine four characters as two Chinese characters 'Death God'. The display routine fails on some video cards. Interestingly, in November 1994, a message complaining that it had been named incorrectly, and was actually called 'Jerry virus' was posted in the local virus echo. The writer of the message claimed it was not related to Jerusalem because it 'only has 30 bytes like Jerusalem'.

Shatin, which first appeared in Febraury 1995, takes a different approach to displaying Chinese characters. It uses block graphics to make the Chinese characters for 'Forget me not' as a full screen display (fig. 2). All these viruses have been widely reported in the local virus echo, but there have been no local reports from commercial sites.

I had hoped to give a detailed profile of Hong Kong virus writers, but firm evidence has not surfaced. The impression from numerous BBS messages is of a small number of actual writers, probably adolescent males, and a larger number of 'wannabe' virus-writers and hangers-on. One correspondent at different times claimed to be a virus writer and to know a virus-writing group, but not be a writer himself. He claimed to be concerned about what the group would do to him if he revealed what he knew. I could not work out if he was trying to fool me, or if he was really worried, or even if the messages originated from one or more people.

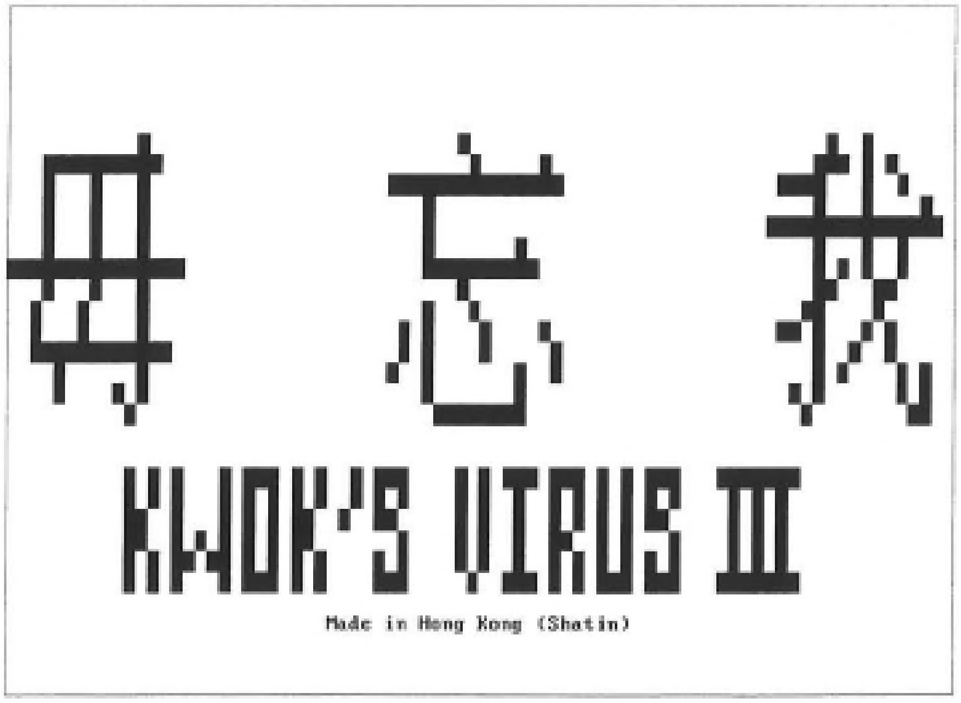


Fig. 2: Screen shot of Shatin

THE ROUTE FROM CHINA

The commercial companies have more international interchange.

Literally thousands of Hong Kong companies have offices or factories in China. It is easy to imagine that this provides a channel for viruses written in China to reach the international scene. Two specific incidents suggest, but do not prove this:

ANTICMOS

In March 94, within a couple of weeksof each other, I received two samples of AntiCMOS.A. One was from an airline, the other a manufacturing company. Both believed that the virus had come from their China branches. These were not the first samples of AntiCMOS found internationally, but they were close to that date.

The next stage in the chain is a company without an anti-virus policy. In April 94, I received a sample from a large local bank. In May, a salesperson for an on-line banking service from that bank was found to have an infected demonstration diskette. The diskette in question was not write-protected, and the salesperson's standard procedure was to use a customer's machine to do a directory of the diskette, and then to boot the machine from it. This procedure seems designed to maximise the chances of picking up an infection and distributing it to the maximum number of customers. Since then, reports of AntiCMOS have risen.

I have had only one sample of AntiCMOS.B, in early May 94. This was collected from a computer company in Shenzen. This variant contains the text: 'I am Li Xibin'. Li Xibin is a plausible Chinese name, that and the dates of first appearance suggest that AntiCMOS is a Chinese virus.

MANGE_TOUT.1099

This was first found in January 1994 in a manufacturing company with a branch in China (as labour costs rose in Hong Kong, many companies moved their factories to China, and kept their head office in Hong Kong). Again, the possibility that this virus had travelled from the Chinese branch was mentioned. It spread in Hong Kong in the subsequent months and a trip to Beijing at the beginning of August 94 showed it to be well known there. Several samples were received, and local anti-virus software detected it as DA01. Near the end of August, it leaped to Norway via video device driver diskettes which had been duplicated in Hong Kong.

CHINA

China itself requires a lot more research, but of the ten viruses found, six have not been seen elsewhere. Of the others: AntiCMOS.B was found almost simultaneously in China and elsewhere; Mange_tout we have already mentioned; Hidenowt was found in Hong Kong soon after its original discovery (Chinese anti-virus software recognises it with the name of 'Dong'); and Changsha contains a name and address in China.

A perception noted in China was that foreign anti-virus software is unable to detect Chinese viruses, this sample shows how that idea could arise. The Chinese anti-virus software mentioned is called Kill and is produced by the Ministry of Public Security. It used to be distributed free, but it is now being sold. The apparent conflict of interest between the same organisation having responsibility for issuing permits to sell security software, and selling its own security product, does not seem to be a concern.

THREE ENVIRONMENTS

In Hong Kong, the corporate and hobbyist appear to form two separate environments (fig. 3). The corporate environment has more frequent Chinese and international connections but, for the most part, exchange only data diskettes. Thus, boot sector viruses are dominant and their prevalence assisted by the failure of many corporations to take the simplest of precautions.

Fig. 2: Major and Minor Routes of Virus Spread in Hong Kong and China

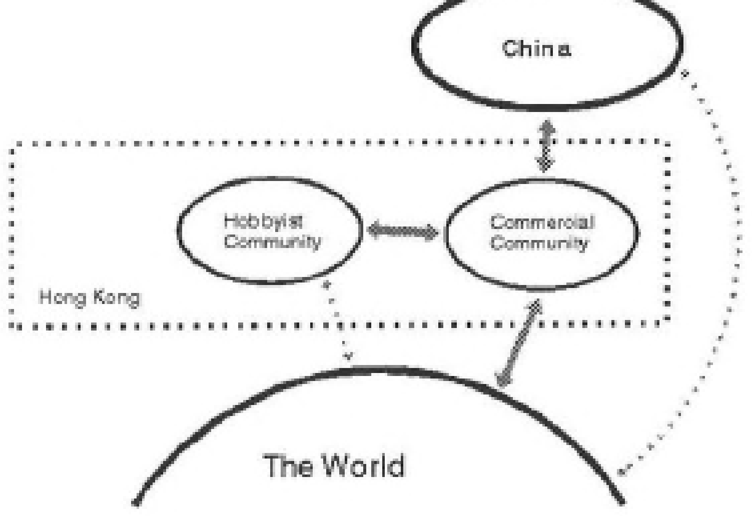


Fig. 3: Major and minor routes of virus spread in Hong Kong and China

The hobbyist environment frequently exchange files on BBSs, but these will generally be entirely within Hong Kong. So, file viruses are common, and locally written viruses are well represented, but some viruses, well-known internationally, are missing. This may change as the Internet boom allows exchange between previously isolated collectors.

China is largely isolated, but has its own virus writers. Who they are, and why they write, is currently unknown. However, it seems certain that we will encounter more from China as it develops and increases its links.

China and Hong Kong
Beijing (Peking)
Shanghai
Guangzhou (Canton)
Shenzen
Hong Kong

REFERENCES

[1] Dr Matthew K O Lee, 'Information Systems Security: legal aspects', *Hong Kong Computer Journal*, vol. 9 no. 11 pp. 19-22

[2] Title: 18th Feb., 1994 PRC NC no. 147 issue, available at Law-on-Line, URL:http://lawhk.hku.hk

CASE STUDY OF VIRUS CONTROL IN A LARGE ORGANISATION

THE UNITED NATIONS PEACE FORCES APPROACH TO VIRUS CONTROL

Lucijan Caric

Information Technology Services Section, United Nations Peace Forces, PO Box 870, Zagreb, Croatia
Tel +385 1 180394 · Fax +385 1 176288 · Email Lucijan_Caric_at_DPKO-UNPF@un.org

Phillip D. Kruss

Information Technology Services Section, United Nations Peace Forces, PO Box 870, Zagreb, Croatia
Tel +385 1 180327 · Fax +385 1 176288 · Email Phillip_Kruss_at_DPKO-UNPF@un.org

ABSTRACT

The United Nations Peace Forces (UNPF) operation in the former Yugoslavia includes a significant information technology (IT) component, with an IT Services Section (ITSS) staff of 80 personnel, several networks and 4,000 PCs. Despite initial similarities, the environment under which ITSS must control viruses is not a typical corporate one. There are fresh injections of viruses on a regular basis because each of the many troop contingents rotates every six months. Also, a high percentage of the PCs are standalone and located in war-torn locations that are difficult to access on a regular basis in order to update virus scanners.

Prior to the implementation of a comprehensive anti-virus program, the number of virus incidents had reached a critical level. Hence, all measures which could be brought to bear were utilized, including BIOS-in-built features, scanning, and the use of cryptographic checksums. It was also clearly necessary to develop a system which would defend against attack from users (e.g. changes to the DOS environment), which could cause as much damage and downtime as viruses, and also lead to the disabling of the virus protection. The constraints to the approach were software cost, maintenance cost, and the need to reduce the annoyance to the user to a minimum (the greater the impact on the user – such as long boot process – the more likely the user was to attempt to disable the virus protection).

An analysis of the virus population at large in UNPF at that time showed that 90% of infections were due to boot sector viruses, and therefore an initial emphasis was placed on the elimination of these viruses. Many of UNPF's computers possess so-called built-in 'Boot Sector Virus Protection', but, for a number of reasons, this was found to be totally ineffective. However a BIOS-configuration solution was found and implemented which drastically reduced the incidence of boot sector viruses and as an added bonus reduced the 'annoyance factor' by speeding up the boot process. This solution has often been overlooked.

In addition to the above, both virus scanning and verification of cryptographic checksums were introduced. This implementation covered memory and the most critical files, including those files which do not come under virus threat, but are subject to user attack; the checking of files subject to user attack was implemented to prevent deterioration of the DOS/Windows environments and disabling of the virus protection itself. In the execution of these measures, it was found that displaying a message to the user regarding the detection of a virus was simply not effective. Hence, on detection of a virus or a user-instigated change to vital system files, locking of the system was introduced so that an infected computer could no longer be used.

The end result of this exercise was that, at very low cost, the computers were very effectively protected against known and unknown viruses, and also against degradation of the DOS and Windows environments due to inadvertent or intentional attack by users.

Despite a continuing significant increase in PCs, and regular injections of virus-infected disks by incoming troops, the number of virus infections of PCs at UNPF has been dramatically reduced.

INTRODUCTION

One major task before the Information Technology Services Section (ITSS) of the *United Nations Peace Forces* (*UNPF*) in the former Yugoslavia at the beginning of the 1994 was the necessity to develop a standard and reliable user environment for the standalone PCs used within the organization. A primary criterion in this regard was to design an operating system environment which would prevent degradation at both the DOS and Windows levels. At the same time, the number of virus infections had reached a critical level, counting more than 15 separate incidents per month. A decision was taken to address both of these problems with one solution.

In the early stages of any *United Nations (UN)* mission, virus infections always present a difficult problem because many computers are moved rapidly into areas out of reach of any centralised information technology (IT) section. Further, there is a communications problem, which results in widespread use of diskettes for the dissemination of information. In addition, the rapid arrival of thousands of civilians and military personnel and the rotation of troops every six months results in a very regular inflow of viruses into the mission area.

There are currently over 4,500 civilians and 40,000 troops in the former Yugoslavia. The total *UN* PC count provided by the *UN* is over 4,000, although many nations, as well as other *UN* and non-governmental agencies, also possess their own PCs. A high percentage of PCs are standalone and are located in war-torn areas that are difficult to access on a regular basis to update virus scanners.

In designing the user environment, we were aware of the problems caused on machines by the users. Installing unauthorised software packages, either intentionally or unintentionally, changing the configuration of the DOS/Windows environments and introducing virus infected files and disks onto the PC were common actions. It was decided to design an environment which would be resistant to change and which would prevent operation of a PC whose configuration was significantly modified. At the same time, that environment needed to ensure trouble-free operation of the PCs, while providing the user with all necessary application software. It should also cause no virus false alarms or irritate users with long lasting and frequent system checks.

THE DESIGN ENVIRONMENT

When anti-virus measures were planned, ITSS was responsible for over 1000 PCs, with more than 3000 PCs projected to be in place within one additional year. More than half of these PCs were standalones and the rest were connected in several multiple-server networks. Most application software was located on the PCs, with only major applications (e.g. procurement) running as server-based. Since more than half of the PCs were standalones, and this percentage was expected to remain static (there is only so much network penetration that can be accomplished in areas such as Sarajevo), it was clear that it would be difficult to upgrade anti-virus software on any regular basis.

ANTI-VIRUS MEASURES

BIOS SETUP FEATURES

Before any anti-virus software is introduced, it is very important to identify and utilize all possible means of virus prevention present as built-in features of the PC. Most BIOS versions for the PCs used by *UNPF* possess several features which can be used to help prevent a PC from becoming infected. There are three main features which were used:

1 Boot Sequence

The ability to alter the boot sequence is a feature available on essentially all desktop PCs available on the market today. By altering the boot sequence, so that the computer boots from drive C: first, the time required for the system to boot up is shortened and the risk of the computer being infected with the boot sector viruses is greatly reduced since it is impossible for a virus to utilize accidental booting from drive A: in order to infect the PC.

2 Boot Sector Protection

Boot sector protection is another BIOS feature available for anti-virus prevention on today's PCs. The idea behind boot sector protection is to prevent a virus writing itself to the master boot sector (MBS) of the hard disk. If a virus tries to write to the MBS, a message is displayed on the screen warning the user and asking for confirmation before writing is allowed. Regrettably this virus protection feature is of very limited use because of the several reasons. One major loophole of this type of boot sector protection is that it typically protects only the MBS of the hard disk, leaving the DOS boot sector unprotected. For example, if infection is attempted by the Jack The Ripper virus it will be detected, but if infection is attempted by the Form virus, the user will not be warned and the virus will successfully infect the PC.

A second disadvantage of such boot sector protection is that it relies on the user's judgement and action. The user is typically confronted with a message like: 'Boot sector write – possible virus. Continue (Y/N)?' Most users as a rule choose the easier possible escape and press YES which leads to infection of the PC. It is our opinion that it would be much better if BIOS producers allowed for the possibility of writing to the master boot sector to be authorized from the BIOS setup in the ON/OFF form. In this case, if a write attempt is made to MBS, it would just cause a message to be displayed stating that writing to the MBS is not allowed. In order to write to the MBS, the user would need to change the BIOS setup first. Since writing to the MBS is a rare operation under most operating systems, imposing a more complicated procedure in order to allow somebody to perform it would not unduly complicate anyone's life since the advantages are obvious.

3 BIOS Setup Password

The availability of a BIOS setup password is a standard feature designed to prevent unauthorized access to the BIOS setup. This feature is of very limited use in professional data protection because passwords are easily disclosed by the many BIOS password crackers available today and many boards lose their password protection if the CMOS battery is removed. However, despite these limitations, this feature has been found to be very useful at *UNPF* through the use of a standard password to prevent a user from altering the BIOS setup, especially the boot sequence.

SCANNING

An anti-virus scanner was loaded onto each PC, with at-boot scanning of memory, boot sectors and selected files at various levels depending on boot action (see next section). However, in view of access difficulties as outlined above, and hence reduced update opportunities, the dependence on scanning was minimised. At this point we are not using a resident scanner because it appears that most resident scanners suffer from a lack of detection capabilities and as well cause significant overheads during operation of the PC.

Since in almost all cases the scanner started during booting was successfully detecting infections on the PC (and because the booting sequence was set in such a way as to prevent accidental infections with boot sector viruses and cryptographic checksumming was used to prevent installation of the unauthorised software to the PC), we did not consider the use of a proactive anti-virus component at this time. It was decided that the server based type of proactive anti-virus software would be considered for use and tested later. This process is now completed and all networked PCs will be protected with a server based anti-virus system.

CRYPTOGRAPHIC CHECKSUMS AND LOCKING

In order to strengthen the protection against unknown viruses, cryptographic checksumming was added to the virus protection offered by the scanner. The use of cryptographic checksums is the only possible way to discover all viruses, known and unknown and is particularly critical at *UNPF* where the scanners on many PCs cannot be updated regularly. Cryptographic checksumming was applied to the master and DOS boot sectors as well as to all executables for the purpose of virus detection. However, this concept was further extended to test the integrity of the main parts of the operating environment, including DOS and Windows configuration files. In other words, cryptographic checksumming was used to detect modifications not only to executable files but to other file types as well, which were not subject to virus attack but which were subject to user attack.

For the purpose of virus detection, the master and DOS boot sectors, all files invoked during the booting sequence, and all executables with the following extensions COM, EXE, SYS, DLL, OV? were checked using cryptographic checksums. For the purpose of detecting user intervention WIN.INI, SYSTEM.INI, PROGMAN.INI, AUTOEXEC.BAT, CONFIG.SYS and all .BAT files were checked.

A customized installation floppy-disk was designed and batch files provided for automated use. The creation and encryption of checksums is carried out through batch files on floppy-disk provided only for use by authorized ITSS staff. Users have no means of computing checksums on their computers. In addition, six DOS and Windows configuration files were automatically copied as part of the installation process for backup and recovery purposes.

Checking is performed automatically each time the computer boots, Windows is started or Norton Commander, Login or Logout are executed. If an integrity violation is discovered, the user is notified. If system configuration files have been modified, then, depending on which files have been modified, they will either be automatically restored from backup and control returned to the user, or the system will be halted and user will be advised to call the Help Desk. If executables are found to be modified, the system is halted.

CHECKING SEQUENCES

On first power up

Memory and hard disks are scanned for the presence of known viruses. Cryptographic checksums of all executable files are checked. If the integrity of any file has been violated, the computer is halted. This full check is performed only once a day on first power up.

Every time computer boots

The computer's memory is scanned for the presence of known computer viruses. Integrity of WIN.INI, SYSTEM.INI and PROGMAN.INI files are checked. If the integrity check fails then the original of each file is restored from backup, with modified files being stored for analysis. The integrity of NC.MNU (Norton Commander Menu) file is checked. If the integrity check fails, the original file is restored from backup. The modified file is stored for analysis. The integrity of all files invoked during booting is checked and if the integrity of any file used during booting has been violated, the computer is halted.

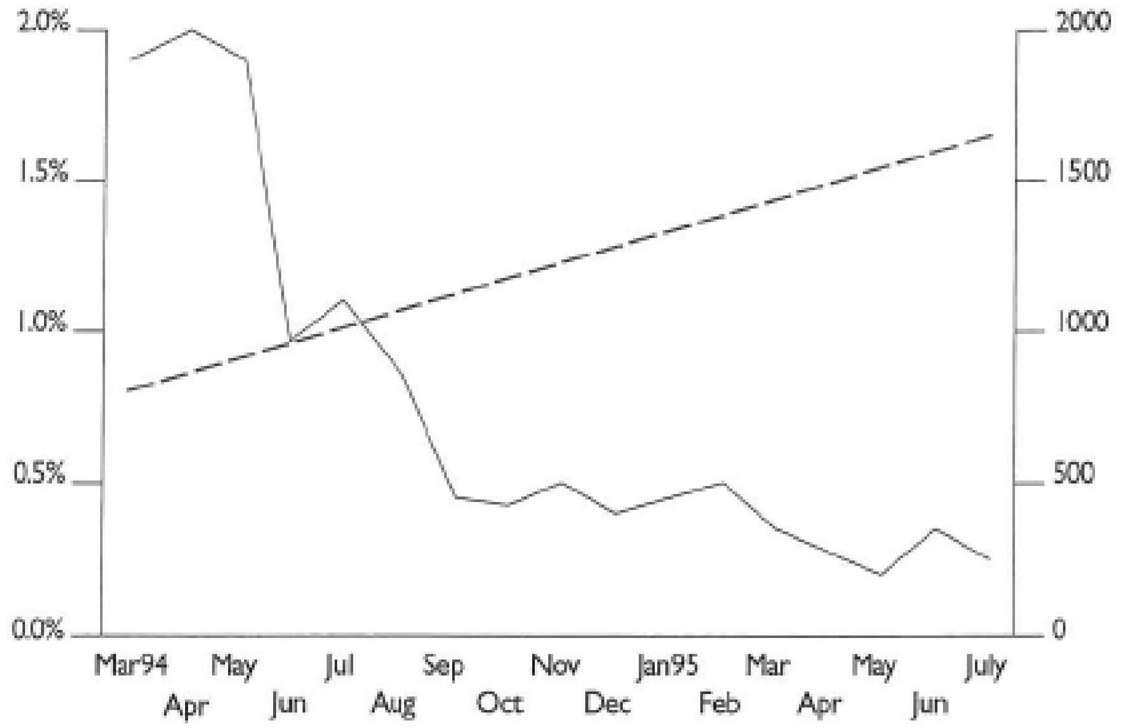
Every time Windows is started or Norton Commander executed

The integrity of AUTOEXEC.BAT and CONFIG.SYS is checked and if the integrity of either of these files has been violated they are restored from the backup. Modified files are stored for analysis.

ADVANTAGES

During the implementation and operation of the above measures, several objectives were reached. With the basic features implemented to prevent accidental booting from infected floppy-disks, over 90% of viruses present in the environment were covered. Implementation of cryptographic checksumming covered the problem of unknown viruses was proved worthwhile by detecting two previously 'unknown' viruses at that time (OneHalf-3544 and SillyCOM-290). At the same time, the boot process was speeded up and checking procedures were causing little or no delay. Finally, operating system setup was protected against tampering and degradation.

RESULTS



The graph shows the very rapid decrease in infections soon after the above measures were introduced (graph covers Zagreb sites only). The graph shows the percentage of PCs infected during a given month (solid line, left axis) as well as the number of PCs in use in Zagreb (dashed line, right axis). Over time, the number of PCs used within the organization grew, the number of virus infections showed no increase. Maintenance was straightforward and effective since the computer setup was standardized and protection procedures were able to identify problems. The number of false positives was very low and detection accurate and effective.

CONCLUSIONS

The best anti-virus protection on a standalone PC is useless if disabled by the user, something which users tend to do in an environment where control is difficult due to lack of access.

Messages to users regarding possible detection of a virus are essentially useless – some disabling action is necessary, within our case, any infected machine being locked to prevent use.

Changing the boot sequence to C: then A: and subsequently password-protecting the BIOS setup is a very effective way of reducing the spread of boot-sector viruses in an environment where floppy-disks are in significant use.

Cryptographic checksum techniques designed for anti-virus purposes can be effectively used to limit users ability to modify Windows and DOS configuration files, thus enhancing environment stability and reducing maintenance costs.

In environments where policy enforcement is difficult, anti-virus techniques cannot be effective unless they are married with techniques for the preservation of the operating system environment, including anti-virus measures themselves – users tend to disable anti-virus measures.

The annoyance factor of any suite of anti-virus measures must be reduced as far as possible to reduce the likelihood of user attack against those measures.

COMPUTER VIRUSES: A GLOBAL PERSPECTIVE

Steve R. White, Jeffrey O. Kephart and David M. Chess

High Integrity Computing Laboratory, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

Tel +1 914 784 7368 · Fax +1 914 784 6054 · Email srwhite@watson.ibm.com

1 INTRODUCTION

Technical accounts of computer viruses usually focus on the microscopic details of individual viruses: their structure, their function, the type of host programs they infect, etc. The media tends to focus on the social implications of isolated scares. Such views of the virus problem are useful, but limited in scope.

One of the missions of *IBM's* High Integrity Computing Laboratory is to understand the virus problem from a global perspective, and to apply that knowledge to the development of anti-virus technology and measures. We have employed two complementary approaches: observational and theoretical virus epidemiology [1, 2, 3, 4, 5, 6]. Observation of a large sample population for six years has given us a good understanding of many aspects of virus prevalence and virus trends, while our theoretical work has bolstered this understanding by suggesting some of the mechanisms which govern the behavior that we have observed.

In this paper, we review some of the main findings of our previous work. In brief, we show that, while thousands of DOS viruses exist today, less than 10% of these have actually been seen in real virus incidents. Viruses do not tend to spread wildly. Rather, it takes months or years for a virus to become widespread, and even the most common affect only a small percentage of all computers. Theoretical models, based on biological epidemiology, can explain these major features of computer virus spread.

Then, we demonstrate some interesting trends that have become apparent recently. We examine several curious features of viral prevalence over the past few years, including remarkable peaks in virus reports, the rise of boot-sector-infecting viruses to account for almost all incidents today, and the near extinction of file-infecting viruses. We show that anti-virus software can be remarkably effective within a given organization, but that it is not responsible for the major changes in viral prevalence worldwide. Instead, our study suggests that changes in the computing environment, including changes in machine types and operating systems, are the most important effects influencing what kinds of viruses become prevalent and how their prevalence changes.

Finally, we look at current trends in operating systems and networking, and attempt to predict their effect on the nature and extent of the virus problem in the coming years.

2 THE STATUS OF THE VIRUS PROBLEM TODAY

Over the past decade, computer viruses have gone from being an academic curiosity to a persistent, worldwide problem. Viruses can be written for, and spread on, virtually any computing platform. While there have been a few large-scale network-based incidents to date [7, 8, 9, 10] the more significant problem has been on microcomputers. Viruses are an ongoing, persistent, worldwide problem on every popular microcomputing platform.

In this section, we shall first review briefly our methods for monitoring several aspects of computer virus prevalence in the world. Then, we shall present a number of the most interesting observations. We will attempt to explain these observations in later sections of the paper.

2.1 MEASURING COMPUTER VIRUS PREVALENCE

We have learned much about the extent of the PC-DOS virus problem by collecting virus incident statistics from a fixed, well-monitored sample population of several hundred thousand PCs for six years. The sample population is international, but biased towards the United States. It is believed to be typical of Fortune 500 companies, except for the fact that central incident management is used to monitor and control virus incidents.

Briefly, the location and date of each virus incident is recorded, along with the number of infected PCs and diskettes and the identity of the virus. From these statistics, we obtain more than just an understanding of the virus problem within our sample population: we also can infer several aspects of the virus problem worldwide. Figure 1 illustrates how this is possible[1].

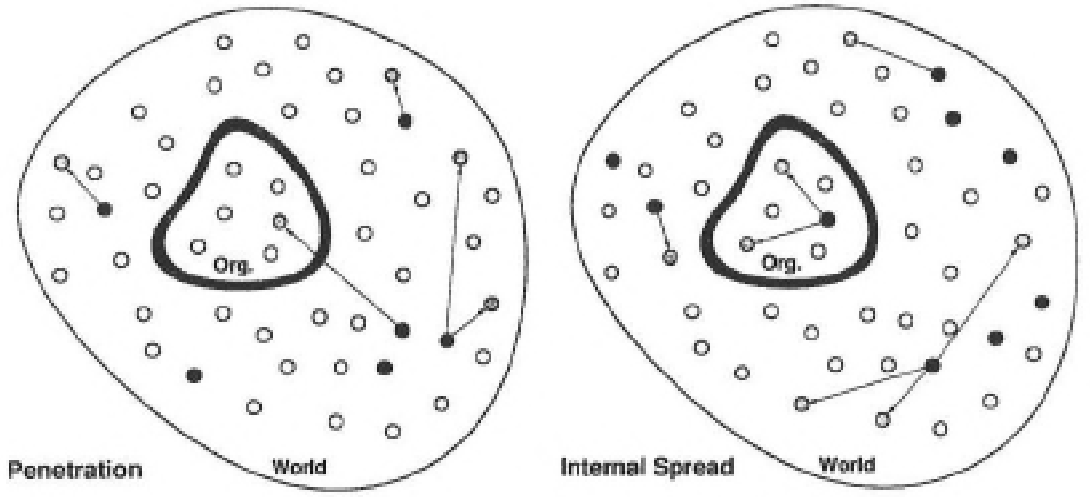


Figure 1: Computer virus spread from an organization's perspective. White circles represent uninfected machines, black circles represent infected machines, and gray circles represent machines in the process of being infected. Throughout the world, computer viruses spread among PCs, many of them being detected and eradicated eventually. Left: Occasionally, a virus penetrates the boundary separating the organization from the rest of the world, initiating a virus incident. Right: The infection has spread to other PCs within the organization. The number of PCs which will be infected by the time the incident is discovered and cleaned up is referred to as the site of the incident.

[1]Further details about our methods for collecting and interpreting statistics can be found in several references [2, 4, 5, 6].

From the perspective of one of the organizations which comprises our sample population, the world is full of computer viruses that are continually trying to penetrate the semi-permeable boundary which segregates that organization from the external world. At a rate depending on the number of computer virus infections in the world, the number of machines in the organization, and the permeability of the boundary, a computer virus will sooner or later make its way into the organization. This marks the beginning of a *virus incident*. Assuming that the permeability of the boundary remains constant, the number of virus incidents per unit time per machine within the set of organizations that makes up our sample population should be proportional to the number of computer virus infections in the world during that time period (in fact, our measure will lag the actual figure somewhat, since incidents are not always discovered immediately).

2.2 OBSERVATIONS OF COMPUTER VIRUS PREVALENCE

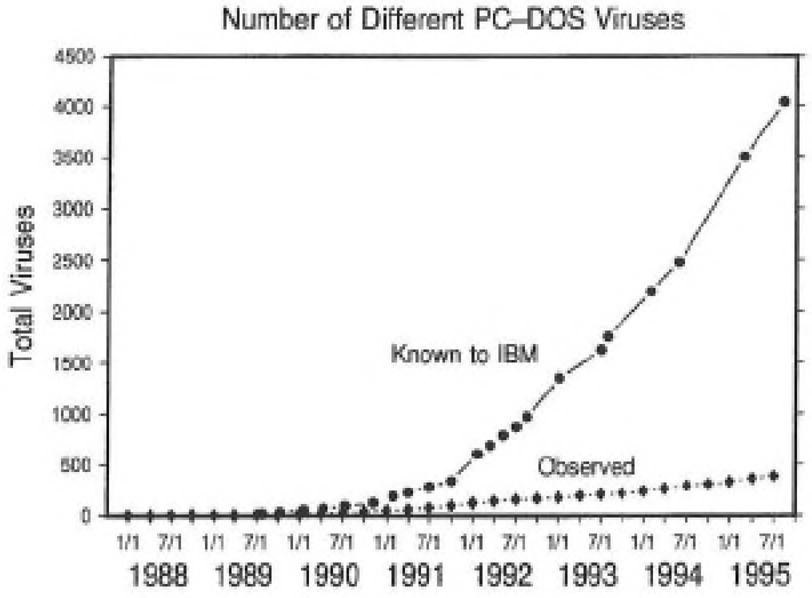


Figure 2: Cumulative number of viruses for which signatures have been obtained by IBM's High Integrity Computing Laboratory vs. time. There are thousands of viruses, but only a few have been seen in real incidents.

As shown in Figure 2, there are thousands of DOS viruses today. During the past several years, the rate at which they have appeared worldwide has crept upwards to its present value of 3-4 new viruses a day, on average (see Fig. 3).

Note that the number of new viruses is not 'increasing exponentially', as is often claimed [11, 3]. The rate of appearance of new viruses in the collections of anti-virus workers has been increasing gradually for several years. The number of new viruses is increasing at no more than a quadratic rate. In fact, almost nothing at all about viruses is 'increasing exponentially'. The problem is significant, and it is growing somewhat worse, but prophets of doom in this field do not have good trade records.

While there are thousands of DOS viruses, less than 10% of them have been seen in actual virus incidents within the population that we monitor. These are the viruses which actually constitute a problem for the general population of PC users. It is very important that anti-virus software detect viruses which have been observed 'in the wild'. The remainder are rarely seen outside the collections of anti-virus groups like ours. Although many of them might never spread significantly, viruses which are not prevalent remain of interest to the anti-virus community. We must always be prepared for the possibility that a low-profile virus will start to become prevalent. This requires us to be familiar with all viruses, prevalent or not, and to incorporate a knowledge of as many of them as possible into anti-virus software. We continue to monitor the prevalence of *all* viruses, regardless of how prevalent they are at present.

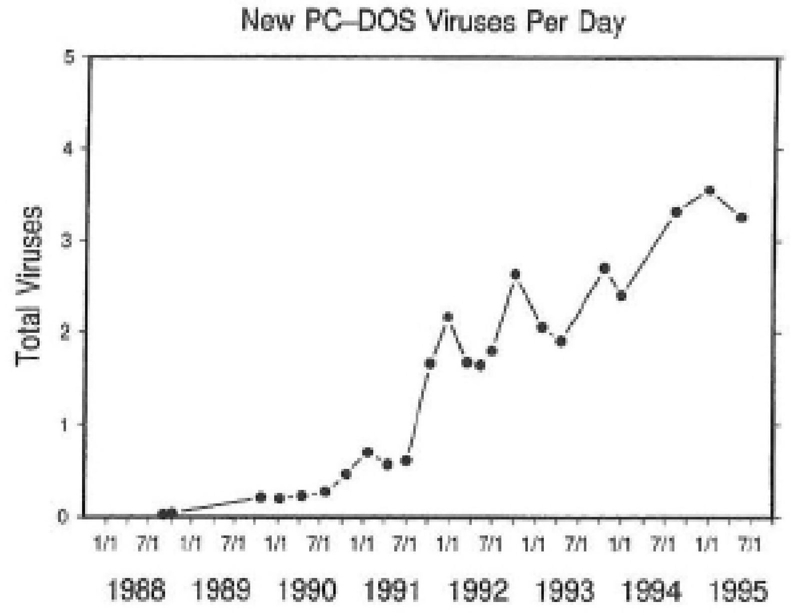


Figure 3: The number of new viruses appearing worldwide per day has been increasing steadily.

Out of the several hundred viruses which have ever been observed in actual incidents, a mere handful account for most of the problem. Figure 4 shows the relative fraction of incidents caused by the ten most prevalent viruses in the world in the past year. These ten account for over two-thirds of all incidents. The one hundred other viruses which have been seen in incidents in the past year account for less than third of the incidents. Most of these were seen in just a single incident.

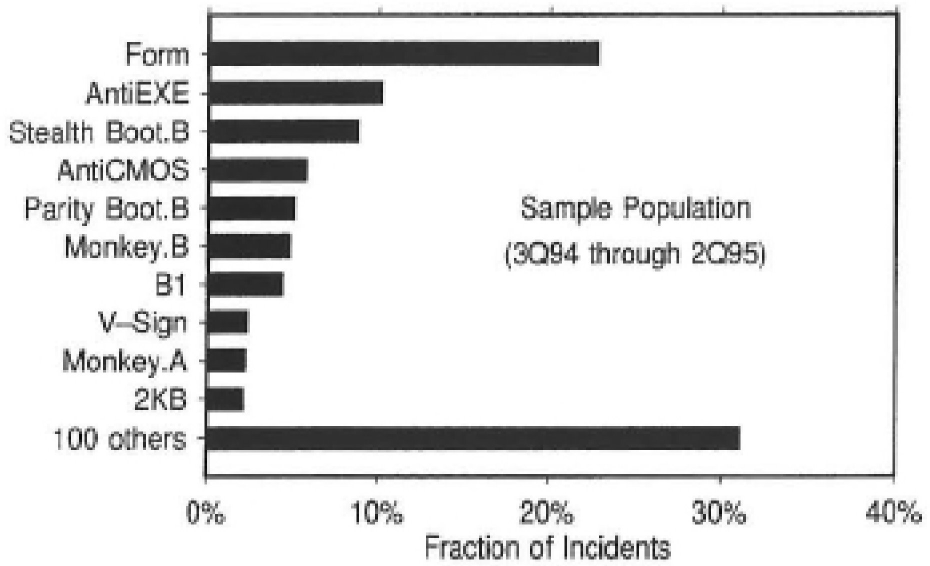


Figure 4: The top ten viruses account for two-thirds of all incidents, and are all boot infectors.

Curiously, the ten most prevalent viruses are all boot viruses. Boot viruses infect boot sectors of diskettes and hard disks. When a system is booted from an infected diskette, its hard disk becomes infected. Typically, any non-write-protected diskette which is used in the system thereafter also becomes infected, spreading the virus. The dominance of boot viruses is especially striking when one takes into account the fact that, of the thousands of known DOS viruses, only about 10% are boot sector infectors.

Boot viruses have not always been dominant. Three years ago, the second and third most prevalent viruses were file infectors, as were 4 of the top 10. The total incident rates for boot infectors and file infectors were roughly equal. Figure 5 provides another view of what has happened to the relative prevalence of these two types of viruses over time. Beginning in 1992, the incident rate for boot sector infectors continued to rise, while the incident rate for file infectors began to fall dramatically. We will attempt to explain this phenomenon in a subsequent section.

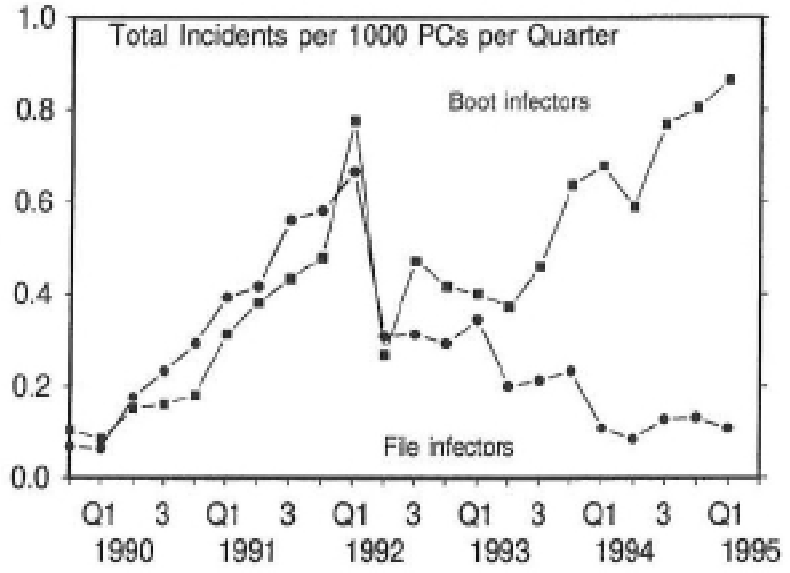


Figure 5: Boot viruses have continued to rise in prevalence, while file viruses have declined.

It is interesting to break up our incident statistics even further into trends for individual viruses. Figure 6 shows the incident rate for selected viruses. Note that some viruses have increased in prevalence, while others have declined.

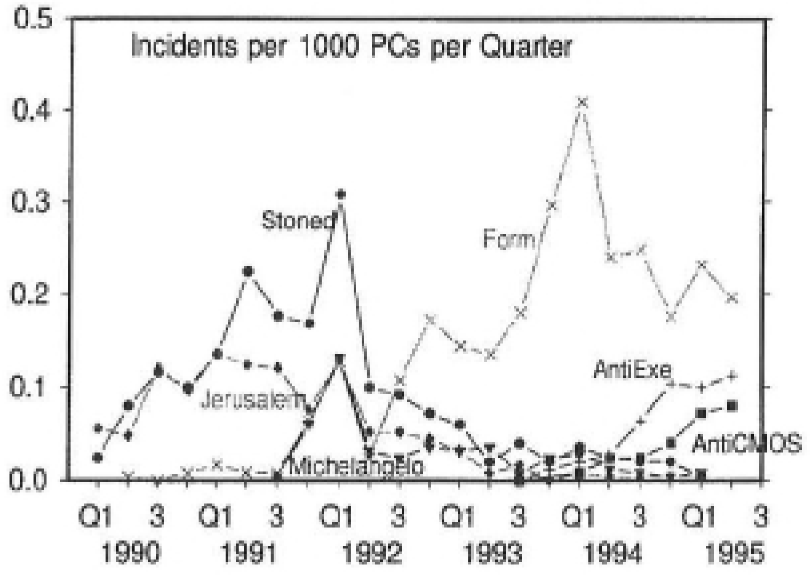


Figure 6: Some viruses have increased in prevalence, while others have declined.

Figures 2-6 raise several important questions:

1. Why are some viruses more prevalent than others?
2. Why do some viruses continue to increase in prevalence, while others plateau or decline?
3. Why are boot viruses so prevalent relative to file infectors, and why has their dominance increased over time?
4. Finally, can we predict what viruses are likely to become more prevalent in the future?

To begin to address these questions, we now review some of our previous theoretical work on virus epidemiology.

3 HOW VIRUSES SPREAD

Over the past several years, we have constructed theoretical models of how computer viruses spread in a population, and compared them against the results of an ongoing study of actual virus incidents [1, 2, 3, 4, 5, 6].

Our models are purposefully simple, in an attempt to understand the most important aspects of global virus spread. In these models, a system is either infected or it is not. If it is infected, there is some probability each day that it will have an infectious contact with some other system in the world, typically via exchange of floppy diskettes, or software exchange over a network. This is called the *birth rate* of the virus. Similarly, there is some probability each day that an infected system will be discovered to be infected. When that happens, it is cleaned up, and it returns to the pool of uninfected systems. This is called the *death rate* of the virus.

The birth and death rates are influenced by a number of factors. A virus' birth rate is governed by its intrinsic properties, such as the particular way in which it infects and spreads. Just as for biological diseases, its birth rate is also highly dependent on social factors, such as the rate of software or diskette exchange among systems. The death rate is determined by how quickly the virus is found and eliminated, which in turn depends on the extent to which people notice the virus, due to its behavior or through the use of anti-virus software. As we shall see, the birth and death rates also depend critically on the nature of the world's computing environment.

All our models show the same basic characteristics of virus spread. One fundamental insight is that there is an *epidemic threshold* above which a virus may spread, and below which it cannot. If the birth rate of a virus is greater than its death rate, the virus has a chance to spread successfully, although it may die out before it spreads much. If the virus does manage to get a foothold, it will start to rise slowly in prevalence. The rate at which it does so is governed by a number of factors, such as intrinsic characteristics of the virus and the overall rate at which software is exchanged. A second fundamental insight which has emerged from our research is that the growth rate can be much slower than the exponential rate that was predicted by one theory [11]. Our theory shows that, when software sharing is localized, the global rate of spread can be very slow, even roughly linear [1, 2]. At some point, the virus levels off in prevalence, reaching an equilibrium between spreading and being eliminated. Figure 7 illustrates the typical behavior of a system above the epidemic threshold.

If the birth rate is less than the death rate – if the virus is found and eliminated more quickly than it spreads – then the virus cannot spread widely. It may spread to a few machines for a little while, but it will eventually be found and eliminated from the population, becoming 'extinct'. Figure 8 illustrates this behavior.

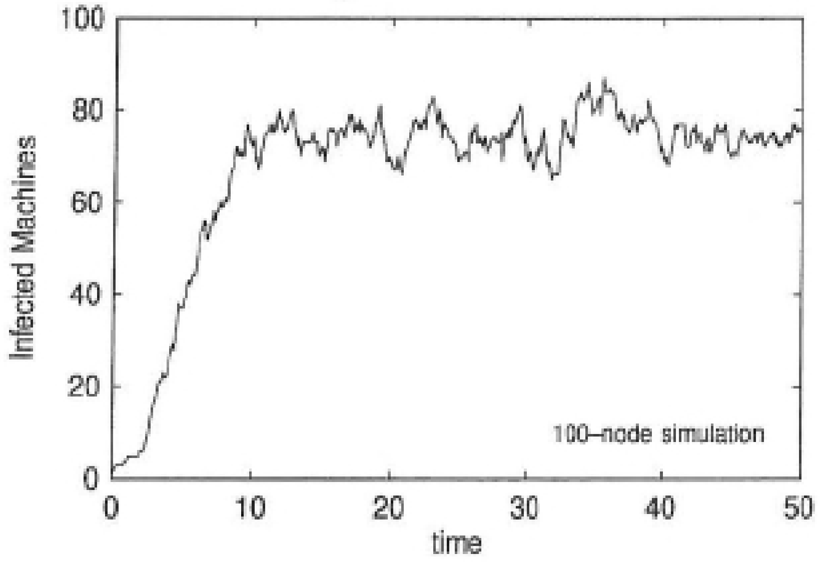


Figure 7: Above the epidemic threshold, a virus rises in prevalence at a rate which depends on a variety of factors, then plateaus at an equilibrium. In this simulation, the birth rate exceeded the death rate by a factor of 5.

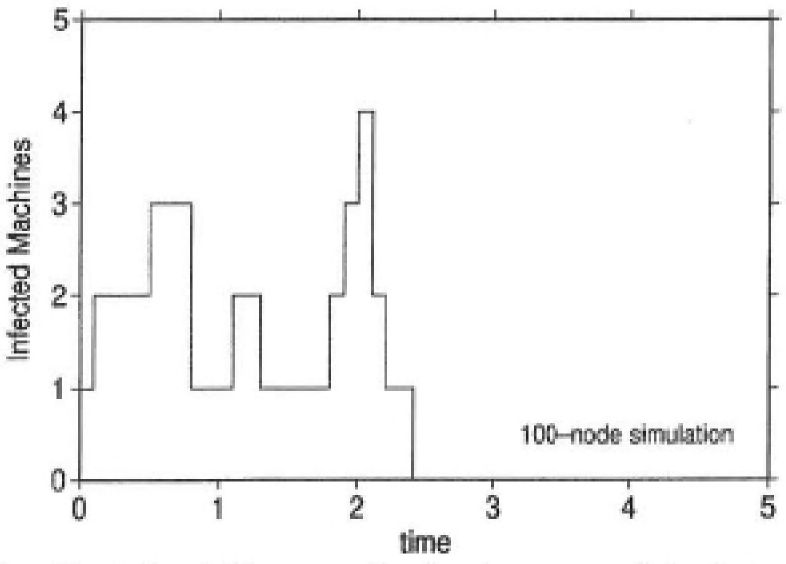


Figure 8: Below the epidemic threshold, very small outbreaks can occur, but extinction of the infection is inevitable. In this simulation, the birth rate was 10% less than the death rate. Note that the vertical and horizontal scales are very different from those of Fig. 7.

4 VIRUS CASE STUDIES

In this section, we illustrate the interaction between viruses and their environment by narrowing our focus to the behavior of selected, individual viruses. We relate changes and shifts in virus prevalence to theoretical findings and to our knowledge of relevant shifts in the computing environment.

4.1 MICHELANGELO MADNESS

The Michelangelo virus was first found in early 1991 in New Zealand. It is a typical infector of diskette boot records and the Master Boot Record of hard disks, with one exception. If an infected system is booted on March 6 of any year, the Michelangelo virus will overwrite parts of the hard disk with random data. This renders the hard disk of the system, and all its information, inaccessible.

The virus is named Michelangelo not because of any messages in the virus itself, but because one of the first people to analyze it noticed that March 6 is the birthday of the famous artist. The name stuck.

Finding a new virus is not unusual in itself; several dozen new viruses are found each week. Michelangelo was unusual in that it was found in an actual incident, rather than as one of the thousands of viruses gathered by anti-virus workers but as yet unseen in an incident. It was also unusual because it could cause such substantial damage to the information on peoples' PCs, and because that damage would all happen on a single day.

In the weeks which preceded March 6, 1992, something even more unusual happened. In a fascinating interplay between the media and some parts of the anti-virus industry, the Michelangelo virus became a major news event. News stories warning about Michelangelo's destructive potential were broadcast on major television networks. Articles about the virus appeared prominently in major newspapers.

As March 6 drew nearer, the stories grew ever more hysterical. The predictions of the number of systems which would be wiped out grew to hundreds of thousands, then millions [12, 13].

When the fateful date came, the predictions of doom turned out to have been a bit inflated. The Michelangelo virus was found on some systems, and probably did destroy data on a few of them. But the worldwide disaster did not occur. Indeed, it was difficult to find any verified incident of destruction of data by Michelangelo in most places [14].

This should not have come as a surprise. Our own research at the time showed that the Michelangelo virus was not very prevalent, and certainly not one of the most common viruses. We estimated that about the same number of systems would have their hard disks crash due to random hardware failures on March 6, as would have their data destroyed by the Michelangelo virus. It is important to keep the risks in perspective.

'Michelangelo Madness', as we came to call it, did have a dramatic effect, though not the anticipated one. Concerned about the predictions of widespread damage, people bought and installed anti-virus software in droves. In some locations, lines of people waiting to buy anti-virus software stretched around the block. In other places, stores sold out of their entire supply of anti-virus software during the week leading up to March 6. Around the world, a very large number of people checked their systems for viruses in those few days.

Figure 9 illustrates the effect of this activity. In the two weeks before March 6, 1992, reports of virus incidents shot up to unprecedented levels. Naturally, this was not because viruses were spreading out of control during those two weeks. Rather, infections which had been latent for days or weeks were found, simply because people were looking for them. In environments like that of our sample population, where anti-virus software is widely installed and used, it is likely that these same infections would have been caught anyway in subsequent weeks. But, since so many people checked their systems prior to March 6, the infections were discovered then rather than later.

People did find the Michelangelo virus, but they found far more viruses of other kinds. The Stoned virus, for instance, the most prevalent virus at the time, was found about three times more frequently than was the Michelangelo virus.

In the first few months after Michelangelo Madness, fewer virus incidents were reported than in the few month before it. This is easy to understand. First, virus incidents were caught earlier than they might have been because everyone was looking. Viruses found in the beginning of March might have been found in the beginning of April instead. So, you would expect fewer virus incidents to be reported shortly after March 6 of that year. Second, viruses were probably found and eliminated even in systems which might not have found them for a very long time. In just a few days, the worldwide population of viruses was decreased. We would expect that the virus population, and hence virus incident reports, would increase again in subsequent months.

Virus incidents did increase after that, but in a way which is rather complicated. We will examine this in more detail in a subsequent section.

Despite the beneficial effects of eliminating some viruses temporarily, the hysteria caused by this event was clearly out of proportion to the risk. Individuals and businesses spent vast sums of money and time warding off a threat which was much smaller than they were led to believe. We hope that those involved learned from the experience – that our friends in the anti-virus industry will be more careful in saying that they understand viral prevalence when they do not, and that the media will examine predictions of impending doom with a somewhat more critical eye.

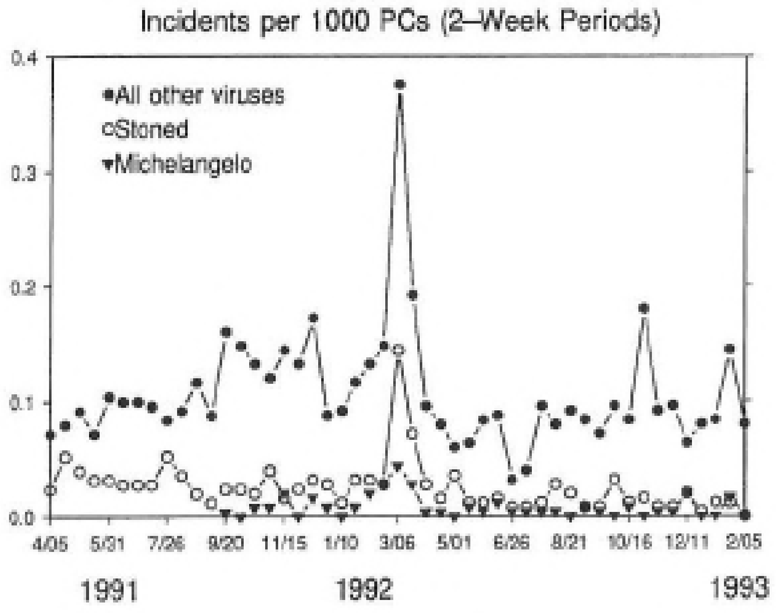


Figure 9: Michelangelo Madness resulted in many people finding viruses of all kinds.

4.2 THE MISSING BRAIN

The Brain virus was first observed in October 1987, making it one of the first DOS viruses seen in the world [15]. It infects diskette boot sectors, and becomes active in a system when that system is booted from an infected diskette. Unlike most boot viruses today, Brain does not infect boot sectors of hard disks.

In the early days of PCs, most PCs were booted from diskettes and did not have hard disks. This provided a perfect medium for Brain to spread. Diskettes used in an infected system became infected themselves, and could carry that infection to other systems. Brain spread around the world in just this way.

Beginning with the introduction of the IBM PC-XT in 1982, the PC industry made a transition to systems which have hard disks. Unlike their predecessors, these systems were not booted from diskettes as frequently. When they were booted from diskettes, it was typically for some special activity, such as system maintenance. Once that activity was concluded, the system was rebooted from the hard disk. It became very uncommon for a system to be booted from a diskette and then used for an extended period of time, with more diskettes being inserted into the system. This denied the Brain virus the opportunity to spread in most cases. The world became a much more difficult place for the Brain virus to spread, and its prevalence declined.

This decline in prevalence occurred before we started gathering accurate statistics about virus incidents, so we cannot illustrate it quantitatively. Anecdotal evidence and our own informal statistics from the late 1980s, however, suggest that the Brain virus was substantially more common than it is today. While Brain is still seen on rare occasions, it does not spread well today. We sighted the Brain virus several times from mid-1988 until mid-1990, but since 1990 it has only appeared in our sample population once, in early 1992.

4.3 NOT STONED AGAIN

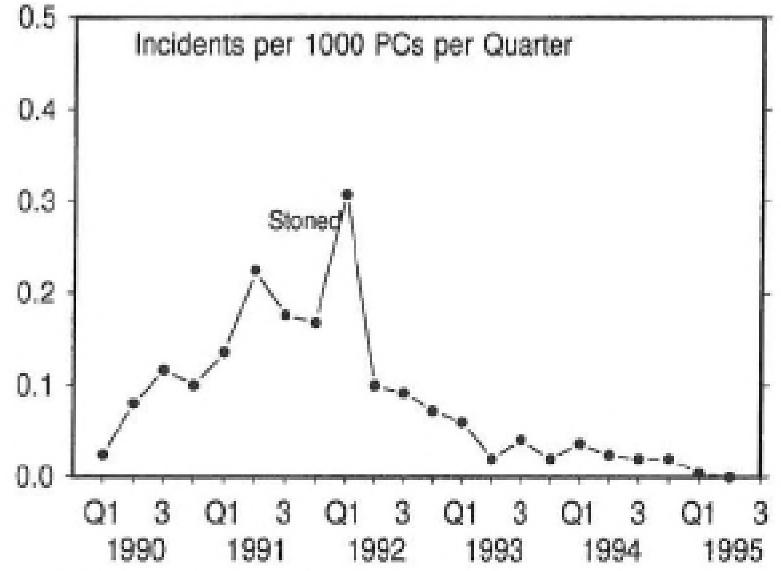


Figure 10: The Stoned virus, a boot infector, rose in prevalence and then declined.

The Stoned virus was first observed in an incident in 1989. It is a typical boot virus, infecting diskette boot records and Master Boot Records of hard disks. One time out of eight that a system is booted from an infected diskette, the message 'Your PC is now Stoned!' will appear on the display. The virus has no other effects.

This virus followed the expected pattern of rising in prevalence through 1991, at which time it had reached a rough equilibrium. After a large peak during Michelangelo Madness, it slowly declined in prevalence over the next several years. Once the most prevalent virus in the world, the Stoned virus is seen much less frequently today.

Its rise in prevalence and subsequent equilibration is what we expect of a virus. Its decline is a bit puzzling at first, until we notice that a system infected with the Stoned virus only spreads that infection to the diskette in the A: drive, not to any other diskette drive. The system became infected in the first place by booting from an infected diskette in the A: drive. The Stoned virus started its life on 5.25-inch diskettes. In spreading from diskette to system to diskette, it could only spread to other 5.25-inch diskettes.

Early in Stoned's life, most systems used 5.25-inch drives, so there was a fertile medium around the world which Stoned could use to spread. In the late 1980s, however, a trend began towards systems that used 3.5-inch drives as their A: drive. The fraction of systems which had 5.25-inch A: drives declined, and has been declining steadily ever since. With fewer and fewer systems that Stoned could infect and spread between, the virus too declined in prevalence.

4.4 JERUSALEM'S RISE AND FALL

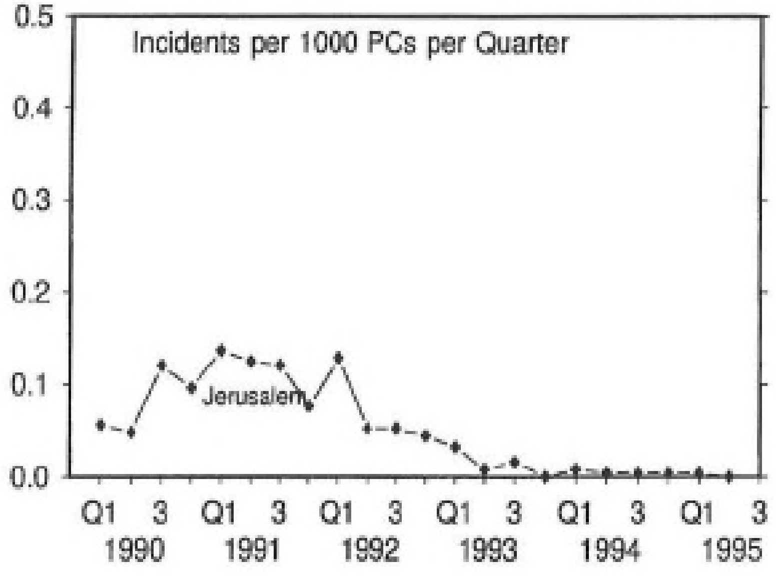


Figure 11: The Jerusalem virus, once quite prevalent, is seen much less often today.

The Jerusalem virus was first observed in December 1987, in the city of Jerusalem, Israel [15]. In many ways, it is an archetypical file virus. When an infected program is run, the Jerusalem virus installs a resident extension in DOS. Subsequently, when any other program is executed, the virus' resident extension will infect the program file.

Prior to 1992, the Jerusalem virus followed the expected pattern of a virus which is spreading around the world. It rose gradually in prevalence through 1990. At the end of 1990, it had reached an equilibrium level in most of the world. Through 1991, it maintained this same level of prevalence, neither increasing or decreasing.

After 1991, however, an odd thing happened. Fewer and fewer incidents of the Jerusalem virus occurred. What was one of the most prevalent viruses in 1990 declined to one of the least prevalent viruses in 1995. Indeed, we saw only five incidents of the Jerusalem virus in our sample population in 1994, and just a single incident so far in 1995.

What caused this decrease? It was not a change in diskette drive type, or the move from floppy diskettes to hard disks. File viruses like the Jerusalem virus spread to files on any kind of diskette, and persist in systems that boot from hard disks. We will return to the cause of this mysterious decrease in a subsequent section of this paper.

4.5 FORM FOLLOWS FUNCTION

The Form virus was first observed in an incident in the second quarter of 1990. It infects diskette boot sectors and system boot sectors of hard disks. When the system is booted from an infected diskette or hard disk, the virus becomes active in memory and infects essentially any diskette used in the system thereafter.

Unlike the Brain virus, the Form virus remains on the hard disk and can spread if the system is booted from the hard disk subsequently. Unlike the Stoned virus, the Form virus is capable of infecting diskettes of any kind in any diskette drive, so it did not remain limited to one kind of diskette. On the 18th of any month, the Form virus will cause a slight clicking when keys are depressed on an infected system. This is often subtle enough to go unnoticed.

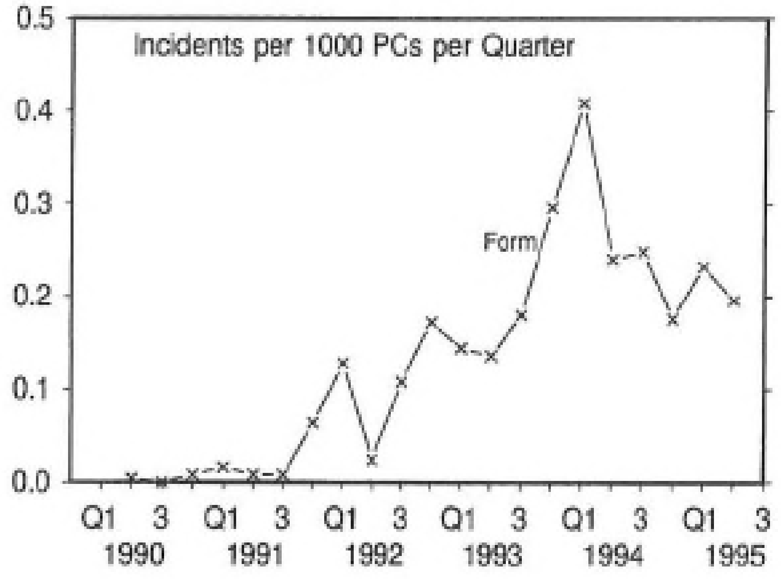


Figure 12: The Form virus, another boot infector, rose steadily in prevalence before reaching equilibrium

The Form virus does not possess the limiting features that caused the Brain and Stoned viruses to have difficulty spreading in the early and middle 1990s. It has exhibited what we expect to be typical behavior for a virus which has found its way into the world. It took over a year before it started rising significantly in prevalence. It rose steadily during 1992 and 1993, becoming the most prevalent virus worldwide. By the end of 1994, it had reached a rough equilibrium at about the same level as other mature viruses such as Jerusalem or Stoned. In the absence of environmental change, we might expect the Form virus to remain about as prevalent as it is today.

5 WHY ARE BOOT VIRUSES SO COMMON?

Boot viruses are by far the most common viruses today, accounting for nearly 90% of all incidents in the second quarter of 1995. File viruses, on the other hand, have decreased in prevalence. This is a remarkable change. Several years ago, file viruses accounted for around 50% of all incidents. What could be responsible for this dramatic change?

Was it Michelangelo Madness? No. That caused only a temporary depletion of viruses of all kinds. Michelangelo Madness explains the large peak in reported incidents, and the subsequent temporary decrease in incidents. It does not account for the difference in prevalence between boot infectors and file infectors.

Is it due to the increased use of anti-virus software? As anti-virus researchers and producers of anti-virus software, we would certainly like to think so. It is tempting to conclude that anti-virus software has made a difference in the world, given our experience with the sample population, in which we have found that widespread usage of anti-virus software and central incident management substantially reduces the size of incidents within an organization [4, 5, 2, 6]. Unfortunately, a closer look at our own data show that, while anti-virus software and policies can make a real difference within organizations, anti-virus software does *not* seem to have made as much of a difference to the world in general. All of the common viruses have been known for quite some time. All of these are detected, even by older anti-virus programs. If anti-virus software was responsible, we would have expected to see a decline in all viruses. The use of anti-virus software does not account for the difference in prevalence between boot infectors and file infectors.

To find the solution to this mystery, we look once again at changes in the computing environment, rather than events associated with the anti-virus industry. The biggest change in the PC computing environment over the past several years has been the change from the use of native DOS to the use of Windows 3.0 and 3.1. Windows 3.0 was released in 1990, and started to become a popular enhancement to the DOS operating system. Windows 3.1, released in 1992, accelerated this trend. Today, many PCs run Windows 3.1.

How does Windows affect the spread of viruses? Experiments carried out at IBM's High Integrity Computing Laboratory demonstrated that Windows is a fragile environment in the presence of typical file viruses. In many cases, if a file virus is resident in the memory of a DOS system, Windows cannot even start. On the other hand, Windows behaves very differently on a system infected with a typical boot virus. For many boot viruses, an infected DOS system can not only start Windows, but can spread the virus to diskettes from within Windows.

If Windows users get a file virus, Windows will typically be inoperable. This will cause the users to eliminate the virus one way or another, whether or not they realise that the system is infected. They might use anti-virus software. They might send their system out for repair. They might re-install everything from backups. Whatever they do, they will eliminate the virus because they cannot get back to work until they do.

If Windows users get a boot virus, however, they might not notice it at all. Windows will usually start and function as expected. Unfortunately, the virus will typically spread to non-write-protected diskettes that are accessed from within Windows. In this sense, most boot viruses are not affected by Windows, and spread in just the same way whether the user is running DOS or Windows. Unless users have good anti-virus software, they will not usually have any reason to suspect a problem, and hence will have no reason to get rid of the virus.

This environmental analysis led us to predict, in 1994, that boot viruses would continue to increase in prevalence, oblivious to the use of Windows. Similarly, we predicted that file infectors would continue to decrease in prevalence. Furthermore, we predicted that boot viruses that were not then very prevalent would become more prevalent, while few file viruses would [16].

This is exactly what has happened. Figure 5 illustrates the dramatic rise of boot virus incidents over the past several years, and the corresponding dramatic decrease in file virus incidents.

Several boot viruses which do spread from within Windows, including AntiEXE and AntiCMOS, were low in prevalence in 1994 but are now substantially more prevalent. As shown in Figure 6, they are approaching the prevalence of more common boot viruses like Form. Once they increase to this level of prevalence, we would expect them to reach equilibrium and not increase further in prevalence.

6 PREDICTING THE FUTURE

We have come to the surprising conclusion that the world's computing environment has been the primary factor in determining the change in prevalence of computer viruses. It is reasonable to assume that this will continue to be the case for some time.

If this is so, we can get some insight into future problems by examining current trends and the expected changes in the computing environment over the next several years. Some of these changes will tend to decrease viral prevalence, while others will tend to increase it.

If there were no changes in the world's computing environment, we might expect to see current trends continue. File viruses would continue to remain very low in prevalence. Boot viruses which have already reached equilibrium, such as the Form virus, would remain at about the same level of prevalence they have today. Other boot viruses would be expected to start becoming more prevalent, perhaps rising in prevalence until they too reach equilibrium. Since there are several hundred boot viruses, having all of them rise in prevalence to the level reached by Form would result in a huge rise in virus incidents worldwide.

There are, however, some environmental changes which we might expect over the next few years: 32-bit operating systems and networking. These changes could have a significant effect on the virus problem.

6.1 32-BIT OPERATING SYSTEMS

One of the significant environmental changes will be the transition from DOS to 32-bit operating systems for PCs, such as OS/2 and Windows 95. In the next few years, we expect that more and more systems will run 32-bit operating systems in order to better use the increasing power of newer PCs.

IBM's OS/2 is a 32-bit operating system which lets users run DOS, Windows and OS/2 simultaneously. The effects of computer viruses on OS/2 systems is described elsewhere [17]. Boot viruses do not generally spread from within OS/2 itself, though they can spread from systems which have DOS as well as OS/2 installed in separate partitions.

File viruses can often spread to other files when infected programs are run in Virtual DOS Machines (VDM) within OS/2. However, they remain active in the system only as long as the infected VDM is active, which is often only as long as the infected program is running. Some file viruses are likely to not spread in VDMs, simply because of differences between VDMs and DOS. This decreases the rate at which file viruses spread in collections of OS/2 systems [17]. In environments in which OS/2 predominates over DOS, we would expect this to lead to a decline in prevalence of all current DOS viruses.

Microsoft's Windows 95 is a 32-bit operating systems that supports DOS, Windows 3.1 and Windows 95 programs. Recent experiments with a pre-release version of Windows 95 suggest that DOS boot viruses will not in general spread well from Windows 95 systems [18]. File viruses were not tested in these experiments.

Preliminary experiments carried out at the High Integrity Computing Laboratory with a pre-release version of Windows 95 suggest that some DOS file viruses will spread as usual, some might not, and some might cause system problems. In environments in which Windows 95 predominates over DOS, we would also expect this to lead to a decline in prevalence of all current DOS viruses.

Not all of the news is good, however. Viruses can be written for 32-bit operating systems, and the first few such crude viruses have already appeared [17]. These operating systems offer new facilities which viruses can use both to hide and spread. The transition to these newer operating systems will change the virus problem, perhaps significantly, but it will not eliminate it.

6.2 NETWORKING

As more and more systems are connected to local and wide area networks, networks may become a more common medium for viral spread.

Of particular interest is the inclusion of networking capabilities in newer 32-bit operating systems. If people typically configure their systems to take advantage of these capabilities, and if that leads to more program sharing on local area networks, it could also increase viral spread in these environments. Currently, these capabilities are used primarily for workgroup computing rather than wide area networking, so the increased spread will result primarily in larger incidents (affecting an entire workgroup instead of a single PC), rather than a large increase in worldwide prevalence.

The final trend which bears watching is the rise of the Internet and global computing. This has the ability to increase the virus problem substantially over time.

There have been incidents of DOS viruses being transmitted on the Internet. Sometimes, they are posted to Internet newsgroups, which function much like bulletin board systems for anyone on the Internet. When the infected programs are downloaded and run, they can infect your PC just like any other infected program. So far, vigilance and rapid action have spread the word about infected programs in newsgroups quickly, and eliminated the problems as they have occurred.

The Internet can be used to support wide-area file servers. These are much like file servers on a LAN, but they can be accessed globally. A virus can spread to files on a LAN-based file server, and from there to the other client systems attached to the server. Similarly, systems which run programs from wide-area file servers can become infected if the programs on the server are susceptible to infection.

While boot viruses could be transmitted on the Internet as diskette images, which would be downloaded and installed onto diskettes, this seems unlikely to become a common means of transporting information. As more information is exchanged over the Internet instead of on diskettes, and the use of diskettes decreases, we would expect a decrease in the prevalence of DOS boot viruses. We would also expect that the increased use of the Internet to interchange and access programs would promote an increase in the prevalence of DOS file viruses.

There have been a few incidents of viruses and worms which are specifically designed to use world-wide networks to spread [7, 8, 9, 10]. These provide dramatic examples of how quickly and how widely viruses can spread on such networks. Fortunately, while these incidents have been rapid and large, they did not usually recur. After a matter of hours or days, when the virus was eliminated from the network and increased defenses put into place, the virus did not continue to spread. Unlike DOS viruses, which have continued to spread around the world for years, Internet viruses have (so far!) been episodic – they come, and then they go. But this need not always be the case.

7 CONCLUSION

The problem of DOS viruses continues to get slowly worse around the world. There are many more viruses than there were a few years ago, and they are appearing at a slightly higher rate. Virus incidents have also increased slightly, but we have to analyze the changes in prevalence of each individual virus in order to understand this trend.

Fortunately, we have made significant progress in this regard. We have achieved a good basic understanding of the spread of computer viruses. We know that a virus can either spread widely or almost not at all, depending upon how fast the virus spreads and how quickly an infection can be found and eliminated. If a virus does spread worldwide, it will rise slowly in prevalence, until it reaches an equilibrium level in the population.

For DOS viruses, this rise is very slow, often taking months or years. The equilibrium level is also quite low. Well-prepared organizations experience about one virus incident per quarter for every one thousand PCs they have, and this incident rate has not changed substantially for a number of years.

Our ongoing study of actual virus incidents had also demonstrated the remarkable effectiveness of good anti-virus software coupled with central incident management in controlling the virus problem within an organization.

This paper has focused on the causes of the major changes in viral prevalence worldwide. We conclude, perhaps surprisingly, that the use of anti-virus software does not play a major role in these changes. Rather, they are determined by the way in which specific viruses, and classes of viruses, interact with the world's computing environment.

We examine the history of several specific viruses to understand this interaction between a virus and its changing environment. The Michelangelo virus was never very prevalent, but media attention to it resulted in increased reports of viruses of all kinds, followed by a temporary decrease in reports. The Brain virus, which spread primarily among systems without hard disks, effectively died out as systems with hard disks became the norm. Virtually all file viruses, including the once-prevalent Jerusalem virus, have decreased dramatically in prevalence because of the increased usage of Windows, and because Windows is fragile in the presence of file viruses. The Form virus, along with other boot viruses, have increased substantially in prevalence, to the point where boot viruses account for around 90% of all virus incidents today. Their spread is not unusual. It is the expected behavior of viruses in a population. They have not died off as have file viruses because their spread is not limited by Windows.

If the computing environment did not change, we would expect that file viruses would remain very low in prevalence, while other boot viruses would increase substantially. If dozens of boot viruses became as prevalent as the Form virus is today, the total number of virus incidents would increase substantially.

By examining trends in the computing environment, however, we can analyze how these might affect computer virus prevalence in the next few years.

Increased use of 32-bit operating systems, such as OS/2 and Windows, is likely to cause a decrease in the prevalence of all current DOS viruses. This is not because they were designed to resist viruses. Quite the contrary: viruses can be written for, and spread by, these operating systems. Rather, the predicted decrease in DOS virus prevalence is simply because features which current DOS viruses use to spread changed in these newer operating systems.

Increased networking, and global networking in particular, will tend to increase the spread of file viruses and decrease the spread of boot viruses. Viruses written to take advantage of features of 32-bit operating systems, especially local and global networking, could become increasing problems. This is a worrisome prospect, as viruses can spread with remarkable speed on world-wide networks.

The technology required to deal with a world of rapidly spreading viruses will be much more challenging than current anti-virus technology. It will be required to respond very quickly, and globally, to new viruses – probably more quickly than humans can respond. While elements of this technology are working in the lab today [19, 20] the task of creating an immune system for cyberspace will occupy us for some time to come [21].

ACKNOWLEDGMENTS

The authors thank Alan Fedeli, Yann Stanczewski and many others for diligently gathering accurate information on worldwide virus incidents for many years. We also thank Joe Wells for his suggestion, later verified experimentally, that *most* boot viruses can spread from within Windows, while most file viruses cannot.

REFERENCES

[1] J.O. Kephart and S.R. White, 'Directed-Graph Epidemiological Models of Computer Viruses', *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, May 20-22, 1991, pp. 343-359.

[2] Jeffrey O. Kephart and Steve R. White. 'Measuring and Modeling Computer Virus Prevalence', *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, May 24-26, 1993, pp. 2-15.

[3] J.O. Kephart and S.R. White, 'Commentary on Tippett's ''Kinetics of Computer Virus Replication''', *Safe Computing: Proceedings of the Fourth Annnal Computer Virus and Security Conference*, New York, New York, March 14-15, 1991, pp. 88-93.

[4] J.O. Kephart and S.R. White, 'How Prevalent Are Computer Viruses', *Proceedings of the Fifth International Computer Virus and Security Conference*, March 12-13, 1992, New York, pp. 267-284.

[5] J.O. Kephart and S.R. White, 'Measuring Computer Virus Prevalence', *Proceedings of the Second International Virus Bulletin Conference*, Edinburgh, Scotland, September 2-3, 1992, pp. 9-28.

[6] Jeffrey O. Kephart, Steve R. White, and David M. Chess. 'Computers and Epidemiology', *IEEE Spectrum*, May 1993, pp. 20-26.

[7] Spafford, E. H. 1989. 'The Internet Worm Program: An Analysis'. *Computer Comm. Review, 19*, p.1.

[8] Cliff Stoll, 'An Epidemiology of Viruses and Network Worms', *12th National Computer Security Conference*, 1989, pp. 369-377.

[9] M.W. Eichin and J.A. Rochlis, 'With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988', *Proc. 1989 IEEE Symp. on Security and Privacy*, Oakland, California, May 1-3, 1989, pp. 326-343.

[10] D. Seeley, 'A Tour of the Worm', Proc. *Usenix Winter 1989 Conference*, San Diego, California, 1989, p. 287.

[11] P.S. Tippett, 'The Kinetics of Computer Virus Replication: A Theory and Preliminary Survey', *Safe Computing: Proceedings of the Fourth Annual Computer Virus and Security Conference*, New York, New York, March 14-15, 1991, pp. 66-87.

[12] J. McAfee, quoting expert sources on The MacNeil/Lehrer News Hour, March 5, 1992.

[13] Joshua Quittner, 'Michelangelo Virus: No Brush With Disaster', *New York Newsday*, April 5, 1992, pp. 68.

[14] Michael W. Miller, ''Michelangelo' Scare Ends In an Anticlimax', *The Wall Street Journal*, March 9, 1992, pp. B5.

[15] Harold J. Highland, 'Computer Virus Handbook', *Elsevier Advanced Technology*, Oxford, England, 1990, pp. 32.

[16] Steve R. White, Jeffrey O. Kephart, David M. Chess, 'An Introduction to Computer Viruses', *Proceedings of the Fourth International Virus Bulletin Conference*, St. Helier, Jersey, UK, September 8-9, 1994.

[17] John F. Morar and David M. Chess. 'The Effect of Computer Viruses on OS/2 and Warp', *Proceedings of the Fifth International Virus Bulletin Conference*, Boston, Massachusetts, Sept. 20-22, 1995.

[18] 'Viruses on Windows 95', *Virus Bulletin*, June 1995, pp. 15-17.

[19] Jeffrey O. Kephart, 'A Biologically Inspired Immune System for Computers', in R. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 130-139, MIT Press, 1994.

[20] Jeffrey O. Kephart, Gregory B. Sorkin, William C. Arnold, David M. Chess, Gerald J. Tesauro, and Steve R. White, 'Biologically Inspired Defenses against Computer Viruses', *Proceedings of IJCAI '95*, Montreal, August 19-25, 1995.

[21] *IBM*'s Massively Distributed Systems home page on the World Wide Web, http: //www. research. ibm. com/massdist.

VIRUS PROTECTION AS PART OF THE OVERALL SOFTWARE DEVELOPMENT PROCESS

Robin J. Kinney

Varian Oncology Systems, Mailstop C-080, 911 Hansen Way, Palo Alto, CA 94304, USA
Tel +1 415 424 6127 · Fax +1 415 424 4511

INTRODUCTION

In developing written processes for dealing with computer viruses in a commercial software development environment, there are two aspects to be considered: the successful elimination of the virus after an infection occurs, and the prevention of virus infection in the first place. The first is relatively straightforward and mechanical. It relies primarily on thoroughness for success. The second aspect can be more difficult, as it is more motivational and relies heavily on the culture of the organization. Developing a formal process for dealing with computer viruses provides a uniform way of training new employees and a mechanism of improving the organization's performance of both prevention and eradication of viruses.

An alternative to a written process is an *ad hoc* process. When a virus is discovered, why not just choose someone for the task, arm them with the best tools, and tell them to go forth? If the same person is chosen each time, they will likely develop their own process, whether it is captured on paper or not. In addition, they will probably get better at it each time. However, without the benefit of a written process, the upper boundary of process improvement is limited, management intervention is required, and the same person must be involved each time for the task to be efficiently performed.

Committing the process to paper does not provide the complete answer. People must be trained to know when and how to use the process. Using the example of system backups, an alternate system administrator who is shown the location of the development system, how to mount a tape, and what commands to type, can successfully perform that task on his own when using the written procedure for guidance. Effective training of people in how to use the process is key and is discussed in greater detail later.

The purpose of a software process is to support the mission of the organization. This mission is to make money through the development of software products, and to do this in a sustainable way. It is not enough simply to produce innovative products which meet customer needs. You must be fast in time to market and deliver a product of high quality. This allows the software developers to concentrate on new products which generate revenue, as opposed to working on removing bugs found by the customer. High quality and short times to market are expected by customers today: if you can't provide them, your competition will. Effective processes which deal with computer viruses support the mission of the organization, and should be part of the overall software development process.

When examining another organizations processes, that organization must be understood to a certain extent to best determine how those processes may apply to your organization. I will describe our software development organization at Varian Oncology Systems to provide that understanding. I will then move on to discuss the specific processes dealing with virus eradication and prevention; then finish with a case study.

Oncology Systems of Varian Associates produce medical devices used to treat cancer. Our major products are linear accelerators used to deliver radiation therapy to patients and data systems which manage information concerning patients' treatments, and also the business of radiation therapy, such as billing and patient scheduling. The business generates $350 million in annual sales, and employs a total of 1250 people with 40 people involved in software development. Currently we have 3000 accelerators and data systems in use at 2700 locations world wide.

The business is highly complex. Part of this is due to the highly technical nature of the products, and part to the safety aspects, since a coding defect could result in the death of a patient. Our business is regulated by the Federal Drug Administration, and an investigator could turn up on our door step without notice. We are also ISO-9001 certified as of 1992.

The data systems we produce use DOS workstations and fileservers in a Novell network. The linear accelerator also uses a DOS PC to provide the user interface and as a storage of configuration data. Several options available for the linear accelerator require additional DOS PC workstations for user interface and control.

The software development environment also consists of DOS workstations and fileservers in a Novell network. Programming languages are C, C++, assembly, and SQL. Software is maintained in a configuration management system and a defect tracking system is used. Processes are used to promote uniformity in our documentation to coordinate reviews of documentation, to organize software testing, and to conduct software releases, to name just a few.

VIRUS PREVENTION

Even in today's environment of networks, fileservers and workgroups, the floppy diskette is heavily relied upon. The floppy diskette is also the primary vector of computer virus infections. The challenge of virus prevention in commercial software development is to control the flow of software in and out of an organization without reducing productivity or stifling creativity. 'Into the organization' pertains to prevention of infection in the first place and 'out of the organization' pertains to increasing the probability that the virus is detected before distribution to the customer - a highly undesirable event.

There are two ways to prevent infections - the methods used by the Department of Defense (DoD), and everything else. In organizations under DoD contracts, it may be a federal offense to bring unauthorized media into a secure area. This may be a bit excessive for a commercial organization, but any time floppies move freely in and out of an organization, viruses are going to propagate. This means that a process to prevent infection in a commercial software organization is only going to be marginally effective. Still, marginal success is the best we have, short of tight security.

The goal, then, is to develop processes which provide the best possible results. At Varian, our prevention process is based on the statement, 'Do not place a floppy diskette in any computer unless you know that it is free of computer virus'. The purpose of this statement is to raise awareness, as it has obvious flaws if taken literally. For example, how do you know when a diskette is completely free of viruses? Virus detection software offers the best solution, and should be part of the prevention process.

Virus detection software available today is good, but there are no guarantees. Perhaps the most difficult problem is how to motivate people to use such software and use it correctly. The virus detection programs which execute as TSRs perhaps offer the best solution, but again, there are no guarantees. People must be motivated to use them properly, keep them at the latest revision (assuming the signature detection type), and continue to use them even when main memory conflicts arise. Awareness and motivation are key.

Having said that, I must admit that I know of no sure-fire way to motivate people regarding processes. So much of the motivational issue is governed by the organization's system of rewards. Often, software development teams receive their rewards by the timeliness and quality of their products, and not by their effectiveness at virus prevention. Perhaps the best which can be done is motivation on a personal level. Make people aware of the problem of computer viruses and what is expected of them regarding prevention. The written process is a good tool for providing awareness and can also describe how virus detection software is to be used, as well as good practices which decrease the likelihood of infection.

Some of the topics to consider in a process dealing with virus prevention are listed below:

- the vectors by which viruses enter an organization
- the types of viruses and how they infect systems
- the proper use of virus detection programs
- the simple rules of virus prevention:
  - always buy from reputable dealers.
  - never install shareware.
  - 'do you know where that floppy has been?'
- how might a virus manifest itself
- what to do if you suspect an infection.

If we acknowledge the fact that viruses are occasionally going to find their way into the development environment, then two issues are of primary importance: virus eradication and prevention of distribution to external customers.

VIRUS ERADICATION

The process of virus eradication goes hand in hand with the process of virus prevention. The lessons learned during the elimination of a virus can feed back to make the virus prevention process more effective. In our organization, neither process is so complicated that separate process documents are warranted, thus both topics are covered in the same document. This allows more cohesion between the two processes in both execution and education.

Becoming suspicious of a virus when computers misbehave is the crucial first step in the eradication process. If the virus is detected by a virus scanner, this first step is simplified. Equally important to this first step is the notification of a designated person on suspicion or detection. Even if the person finding the virus is qualified, and they usually are, to remove the virus from their system effectively, they must not do so. There are two reasons for this. The first is that it is difficult to say where else in the organization the virus exists. The second is, even though this one computer may be the only one infected (and it usually isn't), merely removing it deprives the organization of the learning experience which leads to process improvement.

At Varian Oncology Systems, we have a Systems Administrator who maintains the development environment. This is the designated person whom we notify, and the person empowered to form an eradication team. We have found the team approach to be the most effective means of removing the virus from the organization. Although the process of eradication is not particularly difficult, thoroughness is demanded, and speed is necessary to prevent further virus spread. A team is best suited for the task. Whoever is the focus of virus eradication must be empowered to assemble a team.

Who are the people who make good members of the eradication team? One logical person is the person who discovered the virus or first became suspicious of a virus infection. He/she is likely to take a personal interest in the eradication process, and is motivated to see it through to the end. Other people who make good team members are the leaders in the software development organization; often the more senior developers. Although you're using the most valuable resource for a task which may not warrant it from a technical standpoint, you're sending a very important message to the organization. You're saying that this is important, and that it requires a thorough and professional job. Using a highly-respected person decreases the likelihood that developers say, 'Gee, can you come back when I'm done debugging this module?' when the eradication team comes around. Just think of the message sent if only the most junior developers are assigned! Clearly, you don't need every team member to be your senior developers - one or two are adequate.

How many people should be on the team? This depends on the size of the organization, and on the type of virus. In our organization, we have three separate product teams simultaneously developing software, and a total of about 40 people involved in this effort. One or two people from each development team up to a maximum of perhaps 10% of the entire software organization, is a good rule of thumb. Be careful to not let your team become too large as it may cease to function effectively.

Now that an eradication team is assembled, what are they to do? Before they can do anything, the virus must be identified and an effective means of removal must be devised. Identification includes understanding the means of infection, of course. Virus scanners are your best resource here if the virus is detectable by this means. Virus Bulletin is also a source of information. Some developers of virus scanner software offer advice over the telephone as part of their product's licensing agreement. They see a lot more viruses in a month than most of us will (hopefully) see in our entire career. I frequently rely on this service. The virus eradication process is useless, unless a means of detecting and removing the virus is devised. Thus, these steps are essential before the team can proceed.

The team is now armed with a means of virus detection and removal, and is ready to go. If the virus can propagate by means other than floppy diskette, the development systems or fileservers should be taken off-line and not placed back on-line until all systems or workstations which have connectivity are cleared of the virus. A good tactic is to divide and conquer. Separate the organization by workgroups, departments, labs, etc., then proceed through that part of the organization checking for and eliminating the virus. Every floppy which could have been used in the recent past must be checked. When searching in people's offices, out of respect for their space, it may be best to engage them in scanning the hard disk and floppies. If you purchase pre-formatted floppy diskettes, and cannot determine the entry point of the virus, it would be prudent to check a diskette from each box. Be especially careful in shared work areas such as labs and testing areas. Don't forget those floppies kept in briefcases, and the systems people have at home. Thoroughness is the watchword during the phase of virus eradication.

As the eradication team moves through the organization eliminating the virus, good record-keeping is necessary. Every floppy diskette, every workstation hard disk, every fileserver checked for viruses should be recorded. The purpose of the recording is to answer three questions. First, how did the virus enter the organization? Second, how much did the infection cost the organization? Third, could the virus have escaped into the customer's environment and, if so, by what means?

A sample worksheet and checklist are included at the end of this paper. The worksheet helps organize information before the eradication effort begins. The checklist simplifies the record-keeping during the eradication process.

The process of eradication should specify checking all areas which could be infected, but must allow some flexibility. For example, if the team has checked a large quantity of floppies and workstations directly associated with the original infection and found no new infections, the search could end at this point. I would recommend that this decision be made on the conservative side, because a missed infected floppy is likely to reinfect the organization at a later time.

The team should attempt to follow the trail of infection by asking questions such as, 'Who has used this floppy?' and 'Who has most recently used this laptop?'. Remember, we want to determine the source as well as removing all instances of the virus.

Even when the team has completed the eradication to its satisfaction, the task is still not complete. For this process, and every software development process, areas of improvement must be explored. This exploration is performed through the development of a post-mortem report. Many of the written records kept by the team during the eradication process are used to generate this report, which is written by the team leader or anyone on the team, and archived for future reference. The post-mortem report should cover the following topics:

1. the chain of events from suspicion through eradication
2. the source of infection, or most probable sources if the specific source cannot be determined
3. the total cost to the organization
4. evaluation of the effectiveness of the prevention and eradication process
5. suggestions for process improvement.

The action necessary to affect the improvement must be assigned to someone for implementation. Imagine the effect on the team if their recommendations are ignored. A member of the eradication team is a logical candidate for the process improvement task. It is important that individuals are rewarded for process improvement efforts. At Varian Oncology Systems, process improvement is expected, and is one of the items considered during annual performance evaluations.

PREVENTION OF VIRUS DISTRIBUTION

There remains one other area in which a process is helpful in dealing with computer viruses: preventing the distribution of virus to customers. At Varian Oncology Systems and in most commercial organizations, this is a highly undesirable situation, to say the least. The nature of our business is such that we would make a service call to each infected customer to install 'clean' software. Each service call costs our business approximately $1000 domestically; even more if the customer is in another country. In addition, it is damaging to our reputation. From our customers' point of view, they are trusting the quality of cancer therapy to the quality of our software. A virus distributed with that software is inconsistent with that trust. Lastly, we are regulated by the Food and Drug Administration, and the distribution of a virus with our software may require explanations to that regulatory agency.

Having identified this as a threat, what processes are effective at mitigating the release of a virus to a customer? A well-defined process for how a software product is released is a good start. I will describe the highlights of the processes which work effectively for us at Varian Oncology Systems.

Our Software Quality Engineering organization, in preparation for software validation, builds the software using 'gets' of source code explicitly called by file name from the software configuration management system. It is unlikely that source code would become infected, except by malicious intent. Libraries in object form could be infected, but this probability is reduced by purchasing only from reputable vendors and never using shareware or objects downloaded from uncontrolled bulletin boards.

Software Quality Engineering, after validation testing is completed, creates distribution masters. These are never created on preformatted diskettes. The distribution masters are released to our manufacturing organization, along with byte counts for each diskette. The manufacturing organization will accept software only if it is properly released, and only from Software Quality Engineering. This provides a narrow controllable channel through which all released software must pass.

The first thing the manufacturing organization does with the distribution masters is to verify the byte counts and scan the floppy diskettes for viruses. The diskettes are kept in a secure location, and media is copied for distribution on an isolated system. This system is kept secure because only authorized people can gain access: these people are knowledgeable about the threat of virus infection, and are educated in the process. From each batch of distribution diskettes made, a floppy is checked to verify the correct byte count.

TRAINING

Education of the processes associated with virus prevention, eradication, and all software development processes for that matter, is essential if the process is to be effective. Integration of the processes associated with viruses into the processes directly associated with software development provides a uniform way of training new employees, and demonstrates an equal level of importance. Employees new to the organization should be provided with training which familiarizes them with the complete set of processes. This education should stress the employees' roles and responsibilities regarding these processes and to their continual improvement.

Clearly, the set of processes for an organization defines the entire collection of activities. Often, for small projects or those which are less critical, not all processes apply. Flexibility for sizing the process to the task at hand should be built in to the processes. This helps eliminate confusion, and empowers the team to determine what activities apply. On the other hand, if certain employees, projects, or teams are held to a different set of standards than those defined by the processes, then a larger organizational problem exists.

CASE STUDY

SUSPICION AND IDENTIFICATION

At 10:05 a.m. on Tuesday, August 9, 1994, Jeff, a software engineer on our data systems product, asked that I come to his office. We had only one systems administrator at the time, and she was off-site at a training course for the week. Jeff was aware of my interest in computer viruses and my involvement in software process improvement. Jeff told me that he had discovered a virus identified as 'Newbug' on a floppy he used to transport files between work and home. The virus was discovered on the floppy at his home, by Central Point's scanner, PCTools Anti Virus version 2.0 executing as a TSR.

DETERMINING THE METHOD OF REMOVAL

I began a preliminary investigation of the virus. I determined that the current version of Vi-Spy, the virus tool generally used by our organization, successfully detected and removed the virus. I was unable to locate our back issues of the Virus Bulletin so I gave a quick call to Ray Glath of RG Software. Ray confirmed that it was a simple boot sector virus which corrupts DOS executables, and that it has aliases of 'Generic', 'D3', and 'Newbug', but is more generally known as 'Anti-Exe'.

ASSEMBLING THE ERADICATION TEAM

With the Systems Administrator unavailable, I was hoping that the Software Operations Manager would assign someone to lead the eradication team. The data systems team was in the middle of a software release, so I volunteered to lead the team. This decision was made at 1:00 p.m. on the same day Jeff first called me.

As this was a simple boot sector virus, I decided to deviate from the process slightly, and begin with a preliminary search through this one project team. I engaged the help of Jeff for eradication during this preliminary phase. Also, due to the nature of the virus, I elected to not remove fileservers and workstations from the network.

ERADICATION OF THE VIRUS

Throughout the afternoon of August 9, and into the next day, Jeff and I scanned for AntiEXE. Below are the statistics for this preliminary phase.

total number of workstations checked	26
total number of workstations infected	1
total number of floppies checked	25
total number of floppies infected	7

Five of the seven floppies infected were the distribution masters of an official build. This was rather curious, given the mode of propagation of AntiEXE, because the build engine on which these masters were built was not infected. This led to two possibilities. Either someone discovered the infected build engine and removed the virus without proper notification, or the floppies were infected before files were copied to them.

Due to the significant number of infections discovered during the preliminary search and particularly, the inconsistencies in the information, the search was expanded with more rigor. I added one more person to the eradication team. During the first three days of the week of August 15th, a total of 19 additional floppies in the data system testing lab were found to be infected. At that time we used preformatted floppies for the distribution masters, so one floppy from each box from our in-house store was checked. These floppies were free of viruses.

Then, on Friday, August 19th, 36 infected floppies and one infected workstation were discovered in the office of one of the data systems technical writers. Many of these floppies were received from an outside vendor under contract to create customer training material for the data system product. This vendor was contacted, but they were very quick to stress that the virus could not have originated with them. It was impossible to tell if their floppies infected the technical writer's workstation or vice versa.

Also on Friday, August 19th, due to the large number of infections discovered thus far, I decided to check the building where the linear accelerator software is developed, and engaged five people to form the eradication team there. We moved quickly through 23 offices, 32 workstations, and 88 floppies, finding no instance of the virus. This effort required approximately three hours.

I had learned during the eradication process that, approximately two weeks before this discovery of the virus, AntiEXE was discovered on the workstation of the same technical writer who, this time, had the 36 infected floppies. At that time, the virus was removed from the workstation by a person who meant well but was outside the software organization, and was unaware of our process.

It was impossible to determine with certainty the vector of the infection. The two prime candidates were the vendor creating the customer training material, and the preformatted floppies.

The statistics for this infection of AntiEXE are as follows:

Elapsed time from virus suspicion to eradication	9 days
Total time to eradicate the virus	9 days
Total number of computers checked	52

Total number of floppies checked	316
Total number of workstations infected	2
Total number of floppies infected	61
Total number of fileservers infected	0
Total man-hours lost due to infection	116
Total cost to the organization	$3500

In examining the statistics for this infection, it seems unlikely that only two workstations would have propagated 61 infected floppies. Unfortunately, there is insufficient data to reach a conclusion.

LESSONS LEARNED

- It would serve us well to have a more comprehensive way of dealing with viruses for the entire enterprise. Had this been the case, we would have likely been spared the reinfection of AntiEXE.
- We have a process by which we evaluate outside software houses. This includes evaluation of their ability to deal with computer viruses. The development of the customer training material was managed by a department which was unaware of this process, and thus did not arrange for a vendor evaluation. Had this process been followed, the probability that the infection would have been prevented increases somewhat. This assumes that this vendor is the source of the virus, which cannot be positively determined. A better way of general process training may be beneficial.
- We should not use preformatted floppies for software distribution masters.
- A group of people working as a team can move quickly and effectively through an organization to search and remove a virus detectable by a virus scanner.

CONCLUSION

Written processes dealing with computer viruses can be of great value to a software organization. Although for a commercial organization the process offers only marginal benefit for prevention, it offers significant benefit for removal of the virus. An effective process can also help in preventing viruses from being distributed to customers with released software. Incorporation of these processes into the set of processes which define how business is conducted provides a uniform way of educating new employees in the area of computer viruses. This incorporation also provides a standard method for process improvement. The processes associated with computer viruses not only complement each other, they mesh with those processes associated with software releases, producing distribution media, development system backup, and development environment security, to name just a few.

Instituting written processes in an organization, and the continuous process improvement associated with this, can be a difficult undertaking. It is extremely hard to build that initial momentum, and there are always those in the organization who fear and resist change. Don't be discouraged. Begin with a small set which the organization is already using informally and build from there.

As for the processes dealing with viruses, each time you are required to respond to an infection, examine what works well and what doesn't. Concentrate your process improvement not only on the mechanics of virus eradication, but also on the infrastructure and communications needed to be effective in the process. It is the infrastructure and communication, as opposed to the mechanics, which will be most important when faced with one of the viruses not detectable by virus-scanning software.

VIRUS ERADICATION TEAM WORKSHEET

Virus Name ______________________________

Aliases ______________________________

Virus Type ______________________________

First Suspicion

Date ____________ Time ______________

Person ______________________

Date Systems Administrator Notified ______________

Time ______________________

Positive I.D.

Date ____________ Time ______________

Person ______________________

Scanner Name ______________________ Version Number ___________________

Virus Expression Characteristics ______________________________

____

____

Virus Elimination Method ______________________________

____

____

Is this the first infection by this virus? Y / N

Previous Date if 'No' ____________

Eradication Team Members

VIRUS ERADICATION TEAM CHECKLIST

Name_______________________________________ Date ____________

page_____ of_____

FOR EVERY ITEM CHECKED:

Type	Name	Location	Time Spent	Infected
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N
f h f/s				Y N

f - floppy **h** - hard disk **f/s** - fileserver

Name - any specific identification of the media or the fileserver (not necessary for uninfected floppies)

Location - name of the lab, office, server room, etc

Time Spent - total time to and from location and checking and/or removing virus and restoration of files

HARMLESS AND USEFUL VIRUSES CAN HARDLY EXIST

Pavel Lamacka

HT Computers Ltd, (Computer Accidents Research Center), Sliacska 1, 831 02 Bratislava, Slovak Republic

Tel +42 7251 426 · Fax +42 7252 742

INTRODUCTION

Some virus authors and even some antiviral experts claim that not all viruses and programming techniques used in them are harmful and therefore bad [Cohen-85] [Cohen-92]. They argue that viruses which have the ability to execute no action, neither harmful nor useful, are harmless and therefore neutral, and that viruses which are able to execute beneficial actions are useful and therefore good. Discussions about harmless and useful viruses are still not finished as can be seen, for example, from [Kaspersky-93] or [Timson-93]. Neither are they academic, because our basic attitude towards viruses; the techniques of their implementation; and their originators and propagators depends on the results of these discussions. If viruses are really neutral in nature, it is necessary to discipline only those responsible for their unsuitable purposes and usage. But if we find out that viruses are bad in principle, we obtain the right to take a consequently defensive attitude towards their originators and propagators. The goal of this paper is the presentation of reasoning leading to a standpoint which is usable in practice regarding the existence and feasibility of harmless and useful computer viruses. The presented reasoning is based on a combination of known, lesser known and so far probably undiscussed facts and conclusions. Those of which are considered contributions of this paper *are indicated.*

HARMFUL VIRUSES

Before we start a discussion about the possible existence of harmless and useful viruses, we will take a look at harmful viruses. Viruses which are able to execute a harmful action, like destroying data or disabling the usage of a computer, are considered harmful. Generally a harmful virus V_H (Fig. 1) consists of at least the two following modules: a module of self-replication M_{SR} and a module of harmful action M_{HA}. The harmful action is usually executed by the virus on a certain condition. Other modules and functions of the viruses, for example, stealth, encryption or polymorphism, will not be considered here, because they are irrelevant to this paper.

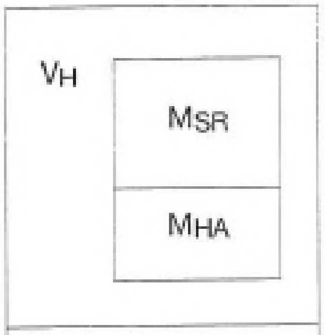


Fig. 1 Harmful virus

Many people claim that viruses have gained their bad reputation only due to the harmful actions which many of them execute. Let us therefore look at whether the harmfulness of a virus would disappear after removing the harmful action code from it, and then at whether it is possible to obtain a useful virus after it is given the ability to execute a useful action.

HARMLESS VIRUSES

Virus V_N (Fig.2), which can do nothing but self-replicate, consists of a self-replication module M_{SR} only. Several such viruses are known in practice. It is known that although this type of virus does not contain any harmful action module, it is able to damage the code of a program on which it is a parasite. This is often caused by the untidy implementation of the virus. It may happen that the virus is implemented in a competent way, but then it meets with a new program structure which it cannot infect properly and therefore it damages the structure. Users have no means of defending against such side-effects because until now it has been unusual to accept complaints about viruses, for example, on hot-lines.

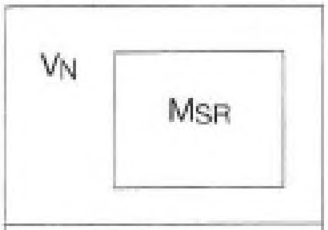


Fig. 2 No action virus

Moreover, it follows from the principle of the function of viruses that they always interfere with the integrity of infected programs by their activity. This results from the fact that all *viruses obtain control flow by theft*, that viruses steal control from the programs which they have infected. Usually, but not always, they steal control by modifying the code of the infected programs. An example of viruses which steal control without program modification, are companion viruses. *By the theft of control the viruses act as parasites* on the programs infected by them. This ability is given to each virus at the time of its origin. It is the inherently parasitic nature of the self-replication of computer viruses which interferes with the integrity of the programs affected by them.

Besides, by self-replication, *viruses waste computer resources*, particularly memory and processor time, which is also a form of doing harm. Although this form is often tolerated, it is *unpredictable* and in time-critical applications it can be substantial and is therefore intolerable.

It depends whether some of the given influences are demonstrated to be harmful ones. In all cases, by their ability to self-replicate and their parasitic nature, viruses violate the conditions of function of the programs affected by them, therefore the authors of the programs cannot guarantee the functionality of the programs, which restricts their author's rights.

From the above mentioned arguments, it is sufficient for everyday practice that the best which can be said about the simplest viruses, containing no code for harmful actions, is the following:

(1) The harmlessness of computer viruses is not guaranteed.

In other words, the usage of any virus is risky, because of the danger of violating computer activity. This riskiness of computer viruses results from their ability to parasitically self-replicate. Because without this ability the virus is not a virus, it follows that this riskiness is peculiar to all computer viruses, also in cases when they have no ability to execute harmful action, and even when they have the ability to execute useful action. It also follows from that, that the danger of harmful viruses does not rest only in their ability to execute harmful actions.

USEFUL VIRUSES

Generally, a useful virus V_U (Fig.3) consists of at least the following two modules: module of self-replication M_{SR} and a module of useful action M_{UA}. The useful action is usually executed by the virus on a certain condition.

A useful action which virus can execute is, for example, the compression of the code of an infected program, as is done by the virus 'Cruncher' [Kaspersky-93] [Timson-93]. Another example is the virus 'AVV' [Kaspersky-94], which detects the presence of other viruses.

It is problematic to evaluate the virus V_U as unambiguously useful, because it is unknown whether the usefulness of its action outweighs the riskiness of its self-replication. Moreover, it is problematic to compare the usefulness of the action to the riskiness of the self-replication. Even if the usefulness of the action was much greater, the riskiness of the self-replication, however low, might be intolerable, and therefore, the virus as a whole could not be evaluated as unambiguously useful.

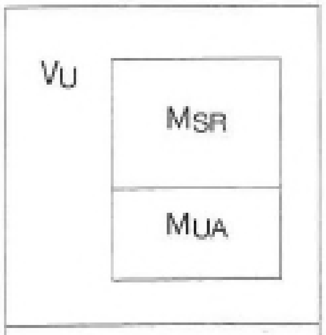


Fig. 3 Useful virus

Let us consider a virus V_C, whose useful action is a compression of the code of programs. Fig. 4 shows the situation when N programs P_1 - P_N have been infected by the virus. Each of the programs P_i has been compressed at the time of its infection by the virus. Along with it a part of the virus V_C acting as a parasite on it has been compressed. The rest of the virus, which is a decompression module D, has not been compressed and receives control at the time of activation of the program P_i. Module D decompresses the program P_i and the compressed part of the virus V_C to their original state, control is passed to the decompressed remainder of the virus V_C, and the rest of the process goes on as usual for viruses. This means that a program P_i infected by the virus V_C behaves like a self-expanding program.

From the user's point of view, besides the above mentioned problems, there are the following interesting matters. The compression module is present in each instance of the virus V_C, in our case it is N-times, which is not the case in common compression programs. Next, it is interesting that the *virus activity is uncontrollable*, because the virus itself finds the programs to be infected, fully *autonomously*, according to the rules built into it. Due to this reason, the user is unable to decide on which programs the compression should be applied and on which ones it should not. Finally, it is interesting that *viruses behave in an indeterminate way*, because their activity often depends on software configuration, sequence of executed operations and other parameters of random nature. Therefore it is generally *unpredictable* whether at a given moment the compression has been applied to any given program or whether it has been applied to all considered programs.

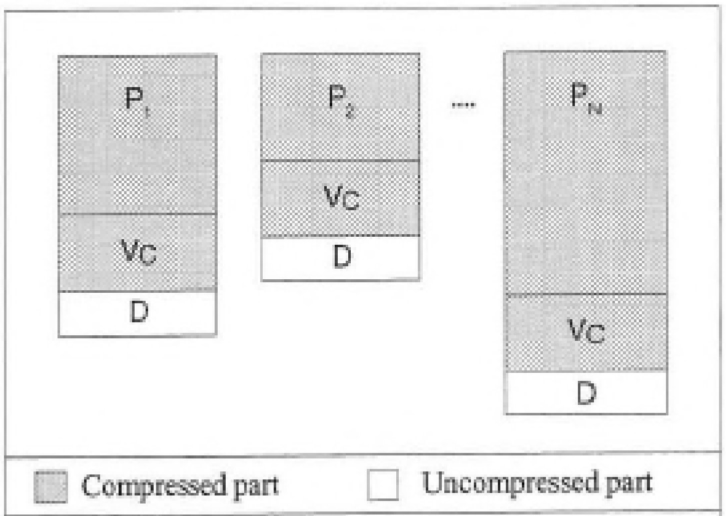


Fig.4 Programs P_i infected by compression virus V_C

The above facts disqualify the useful and harmless viruses to such a degree, that the least negative statement we can say about them, is the following:

(2) *Harmless and useful viruses can hardly exist*

In the next section we will look at whether it is necessary to regret that useful viruses have such weak prospects.

USEFUL VIRUSES VS USEFUL NON-VIRAL PROGRAMS

If a common, correctly implemented compression program is used, we avoid the problems inherent in the compression virus, V_C. First of all, we avoid problems with uncontrollability and indeterminacy, because it is possible to state on which programs to apply the compression and, after the compression is finished, it is apparent that compression has been applied only to the stated programs and not to others.

The differences in the demands on memory and time are not negligible as well. The compression code together with the self-replicating one occur in each instance of the virus V_C, that is, in each infected program (Fig. 4). On the other hand, if we use a compression program P_{CD}, which may or may not be memory resident, the compression code is necessary only in one instance and the self-replicating code is unnecessary (Fig. 5). It can be seen that the programs P_1 - P_N, on which the compression program P_{CD} was applied, contain no extraneous code.

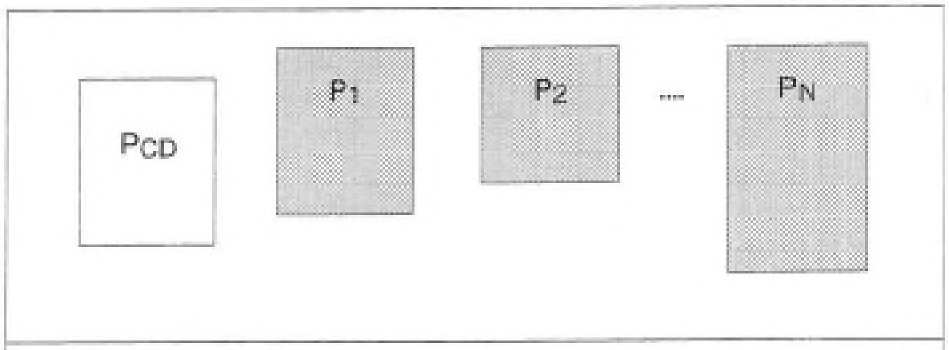


Fig. 5 Programs P_i compressed and uncompressed by program P_{CD}

Similarly, the program P_{CD} is more time-efficient, because it works on demand only, and not like the virus V_C, which works every time it steals control. If the compression program P_{CD} is memory resident, it works autonomously, which removes the last illusory advantage of the useful viruses, for which some of their proponents argue.

In practice we also use compression program P_C, which transforms the given programs into self-expanding form (Fig. 6). A compression program P_C compresses given programs P_1 - P_N and then adds to them a decompression module, D, which automatically decompresses them at their activation. The addition of the decompression module influences the integrity of the given programs, but if the authors of the programs agree with this process, their author rights are not violated, and if they themselves apply such compression to their programs, the integrity of the programs is not affected.

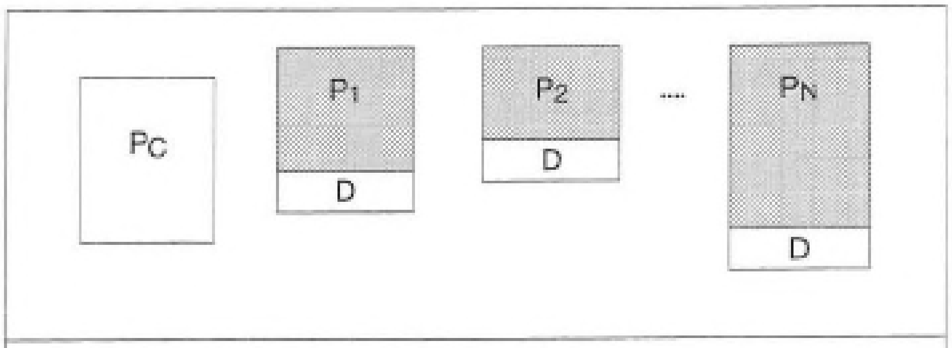


Fig. 6 Self-expanding programs P_i compressed by program P_C

The procedure, which was demonstrated in the comparison of compression viruses and compression programs, can be generalised in the following statement:

(3) *For each virus which is able to execute an action, it is possible to implement a program, which is not a virus and which is able to execute the same action.*

Therefore for each virus V_U (Fig. 3), which is able to execute a useful action using the module M_{UA}, it is possible to work out a useful program P_U (Fig. 7), which is able to execute the same useful action as the virus V_U. P_U needs no self-replication code, therefore it is not a virus. Instead it contains a module of selective application, M_{SA}, by which the useful action is selectively applied according the commands of the user of the program P_U. In an extreme, the program P_U can contain a copy of the module M_{SA}, which would guarantee the equivalency of an execution of the useful action. Using statement (3), the above comparison of features shown for compression viruses and compression programs is valid for every pair V_U - P_U, which executes the same action.

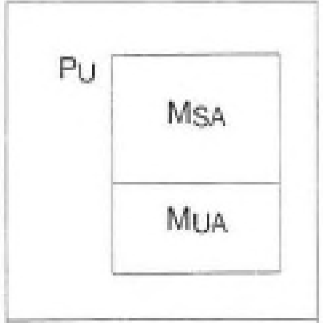


Fig. 7 Useful program

Now we are at the end of the comparison of the features of the useful viruses, V_U, and the useful programs P_U. Their comparison overview is given in Table 1. From it and from statement (3) the following statement results:

(4) Useful viruses are useless.

This is so because useful programs are unambiguously more advantageous, as useful viruses have only one from the given list of positive features, which is the ability to execute useful actions. Otherwise the usage of viruses for useful purposes is *hazardous*, because it is accompanied by several risks.

feature	useful virus V_U	useful program P_U
useful action	+ able to execute	+ able to execute
self-replication	- basic ability, without it virus is not a virus	+ does not need
parasitic ability	- basic ability, without it virus is not a virus	+ does not need
controllability	- autonomous, uncontrollable	+ user controllable
predictability	- indeterminate behaviour	+ predictable behaviour
memory usage	- unpredictably excessive	+ need not be excessive
processor usage	- unpredictably excessive	+ need not be excessive
	- *is risky and therefore negative feature*	+ *is positive feature*

Table 1. Comparison of basic features of useful viruses and useful non-viral programs

CONCLUSION

To justifiably speak about the existence of harmless computer viruses, it would be necessary to prove, or at least to show, how to implement them in such a way that it would be possible to guarantee their harmlessness. That would disprove the validity of statement (1) and at the same time open the possibility of the existence of unambiguously useful viruses.

To justifiably speak about the existence of useful computer viruses, it would be necessary to prove, or at least to show, that there exists an action which can be implemented in the framework of a virus and which cannot be implemented in the framework of any non-viral program. That would disprove the validity of statement (3) and therefore (4). Another possibility would be to prove, or at least to show, that there exists an action which is more effective to implement in the framework of a virus than in the framework of any non-viral program. That would disprove the validity of statement (4), but not (3). To justifiably speak about these viruses as being unambiguously useful, statement (1) may not be valid.

Until such proofs can be given, claims about the existence of harmless and useful viruses are but the products of *wishful thinking* of their proponents, and attempts to create and use them are hazardous. In the case of the useful viruses it is an unnecessary hazard, because useful non-viral programs do not carry the risks which viruses do. The hazardousness of the viruses results from the cumulative effect of risks connected with their usage. These are mainly risks resulting from the parasitic self-replication of the viruses, from their uncontrollable and indeterminate activity, and from their unpredictably excessive usage of memory and processor.

Software engineering looks for programming techniques whose use minimises the risks of incorrect software function. This is why we consider as unsuitable those programming techniques, which the hazardousness of the computer viruses is based on. From the viewpoint of software engineering, *viral programming techniques are dirty* at least as unstructured or non-modular programming is, since their exploitation is dangerous.

It is known that all viruses in some way violate the integrity of the infected programs, which is a given due to their parasitic nature. This interferes with the author and user rights of the respective infected programs. The authors and users of those programs have the right to protection by law, to recompensation and to the prosecution of the culprits who spread viruses actively or support their spread through negligence.

REFERENCES

[Cohen-85] Fred Cohen, 'Computer Viruses', 1985.

[Cohen-92] Frederick B. Cohen, ''Wrong' Said Fred', *Virus Bulletin*, January 1992, pp. 5-6.

[Kaspersky-93] Eugene Kaspersky, 'Cruncher - The First Beneficial Virus?', *Virus Bulletin*, June 1993, pp. 8-9.

[Kaspersky-94] Eugene Kaspersky, 'AVV - The Anti-Virus Virus', *Virus Bulletin*, January 1994, pp. 10-11.

[Timson-93] Harriet Timson, 'Cruncher - Zipping or Zapping', *Virus News International*, May 1993, p. 31.

THE EFFECT OF COMPUTER VIRUSES ON OS/2 AND WARP

John F. Morar, David M. Chess

High Integrity Computing Laboratory, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA

Fax +1 914 784 7007 · Email morar@watson.bm.com

OVERVIEW

Although the number of OS/2 viruses can be counted on the fingers of one hand, systems running OS/2 still require protection against thousands of DOS-based viruses which can infect boot records and DOS programs on OS/2 systems.

OS/2 is a 32-bit multitasking operating system which can simultaneously run programs written for OS/2, DOS and *Microsoft* Windows™. DOS programs running under OS/2 execute in a Virtual DOS Machine (VDM) which is designed to provide an environment which appears the same as real DOS. Ironically, moving toward the goal of a perfect virtual DOS machine increases the probability that an infected DOS program will execute properly, and effectively propagate its virus to other DOS programs stored on the system. Indeed, DOS programs executing under OS/2 can frequently spread file infecting viruses to other DOS programs.

Boot sector viruses interact primarily with the Basic Input and Output System (BIOS) which is common to all IBM PC (and compatible) personal computers. Boot sector viruses typically receive control during the boot process, before the operating system is loaded; this allows them to infect boot sectors independent of the operating system in use. Boot sector viruses under OS/2 don't usually spread to diskettes because of the details of how OS/2 uses diskettes. However, they can have other detrimental effects on the system, and therefore need to be removed.

Viruses designed to infect native OS/2 executables are more complicated to write than their DOS counterparts, but they will likely be a problem at some point in the future. We are currently aware of only two OS/2 viruses.

Both of these viruses are very simple and neither of them has been detected in the wild ('in the wild' viruses are those which have been detected spreading in real life situations.)

OS/2 is far more versatile than DOS/Windows. It has the ability to run multiple DOS and Windows sessions, provides facilities for booting multiple operating systems, and allows file names up to 255 bytes long.

These additional capabilities offer new places for viruses to hide, and necessitate additional anti-virus capabilities not available in native DOS/Windows anti-virus software. Although native DOS/Windows anti-virus programs can often execute under OS/2, they do not provide an adequate level of protection. They will not have access to files with long names, nor will they find all the boot records which may be located on the machine. Under some versions of OS/2, the boot sectors are not writable by DOS/Windows programs and can therefore not be disinfected by DOS/Windows anti-virus products.

This susceptibility to DOS viruses is not unique to OS/2. Indeed, Windows NT and Windows 95 are also fertile ground for the spread of DOS viruses. Each of these operating systems requires individually tailored anti-virus protection software.

BOOT SECTOR VIRUSES UNDER OS/2

Boot sector viruses are responsible for the overwhelming majority of in-the-wild virus infections. They reside in the boot records found on each disk and diskette. The primary method for checking a particular machine for boot sector viruses is to scan all the boot records located on the personal computer. One consequence of OS/2's versatility is the possibility of additional boot records not found in DOS systems.

OS/2 provides optional facilities for installing more than one operating system on a single personal computer.

Two methods are provided for choosing which operating system is to be booted: Dual Boot and Boot Manager. Each of these methods involves manipulating which boot sectors become active. Using either of these techniques results in additional boot sectors not found on DOS and Windows-based systems. Effective OS/2 anti-virus programs will scan all the boot records located on the personal computer, even those which are not active at the time the scan is being performed.

In particular, on a Dual Boot system the system boot record of the operating system which is not currently active is stored on the hard disk, under a special name.

OS/2 anti-virus software should know to scan for boot sector viruses in files with these names. In a Boot Manager system, a special Boot Manager boot record exists which is neither a master boot record nor an operating system boot record; OS/2 anti-virus software must know how to scan and repair this special kind of boot record.

OS/2 offers a choice of two file systems; the File Allocation Table (FAT) file system and the High Performance File System (HPFS). Some boot sector viruses assume that all file systems are FAT, and write to specific disk locations in ways which can damage HPFS boot partitions. The risk of such complications for OS/2 systems in high virus risk environments can be minimized by using the FAT file system for all boot partitions.

DOS FILE INFECTING VIRUSES UNDER OS/2

DOS programs running on an OS/2 system execute inside a Virtual DOS Machine (VDM), a controlled environment in which OS/2 provides DOS programs with all the usual DOS services, and in general simulates a DOS environment.

Multiple VDMs can be used to simultaneously execute multiple DOS programs. Many infected DOS programs execute properly in OS/2 VDMs and can effectively propagate a virus to other DOS programs stored on the system.

File infecting viruses frequently install a memory resident component in the DOS operating system (or in the VDM in the OS/2 case); this component infects new programs as they are executed, or executable files as they are opened, or it may follow any of a variety of other strategies. Because the DOS simulation provided by the VDM supports this kind of memory-resident component, viruses of this kind often continue to operate in a VDM.

(Some file infecting viruses use undocumented and unsupported features of DOS to function; these will often fail in OS/2 VDMs.)

Memory-resident viruses cannot spread directly between separate VDMs; however, any program executed from within an infected VDM will likely become infected. If that program (once infected) is later executed in another VDM, that VDM can also become infected, in the sense that the virus will have installed its resident portion in that VDM as well.

The best protection for VDMs under OS/2 is to install memory-resident virus protection in each VDM as it is opened. This function can be performed automatically by anti-virus software tailored to the OS/2 environment.

Occasionally, a file-infecting virus designed for DOS will also attempt to infect OS/2 executable files. Although the structure of an OS/2 executable file is superficially similar to a DOS EXE file, it is in fact far more complex. If a DOS program attempts to infect an OS/2 executable, it will almost always fail, rendering the OS/2 executable unable to execute under OS/2, and making it impossible to repair fully the file. In some cases, trying to start an OS/2 program in an infected VDM can cause the OS/2 program's 'DOS stub' (the part of an OS/2 program which prints 'This program cannot be run in DOS mode') to become infected. An OS/2 program infected in this way can sometimes even spread the virus when started under DOS, or in an OS/2 VDM. It is therefore important to check both DOS and OS/2 executables for file-infecting viruses on OS/2 systems.

NATIVE OS/2 VIRUSES

There are currently only two OS/2 viruses known to us.

- **OS2vir1**: This virus functions by (roughly) replacing all EXE files in the current directory with a copy of itself. Since infected files no longer perform their normal functions, this is a very noticeable virus and therefore unlikely to spread. It is distributed as source code, and as distributed, prints out messages as it runs saying which files it's "infecting".
- **Jiskefet**: Replaces EXE files with a new file which contains within itself the original EXE file. When the infected file is executed, it recreates the original EXE file under another name and then executes the original file. This is a technological advance over OS2vir1 since the function of the original program is preserved. However, Jiskefet is not very effective at finding new files to infect. Similar viruses in the DOS world have never spread well, suggesting that Jiskefet will also not pose any significant threat to OS/2 systems.

In spite of the current unsophisticated attempts at OS/2 viruses, there is no insurmountable technological barrier to generating effective viruses for any of the currently shipping 32-bit operating systems; it makes sense to prepare now, by installing the best available anti-virus software designed specifically for the operating systems that you are actually using.

TAKING ADVANTAGE OF OS/2 FACILITIES

Like any other modern product, anti-virus products should take advantage of the power and flexibility which OS/2 brings to the user. An anti-virus product should, for instance, be able to run in the background at pre-selected times, to avoid interfering with the user's daily work. An anti-virus product should have a full graphical user interface, allowing any necessary user interactions to take place on the desktop, rather than through older command-line interfaces. Advanced file systems, like the one provided with OS/2, require the system to be shut down so the file system can close all files and store all data. The next action after a shutdown is to restart the system, either immediately or at some later time. The shutdown process is an excellent time to scan any diskette left in the A drive for boot sector viruses. Diskette scanning during shutdown avoids possible infection when the system is again restarted.

Even a non-bootable diskette can be infected with a boot sector virus, and can spread the virus if an attempt is made to boot from the infected diskette.

SUMMARY: REQUIREMENTS FOR PROTECTING OS/2 SYSTEMS

To be effective in protecting an OS/2 system from viruses, an anti-virus product must:

- run as a native OS/2 application, in order to check files and directories which DOS applications cannot see
- check all boot records on the system, including BootManager boot records and the files used by Dual Boot to store boot records
- provide protection for all DOS VDMs and Windows sessions running under OS/2
- check to see if there is an infected diskette in the diskette drive immediately before shutting down
- perform scheduled scans of the system, in the background, exploiting OS/2 multitasking abilities
- take advantage of the sophisticated user interface facilities in OS/2 to run cleanly on the desktop, rather than requiring command-line interaction.

Our development of IBM AntiVirus for OS/2 has been motivated by the need to satisfy all the requirements described in this article.

* IBM, OS/2 and OS/2 Warp are registered trademarks of *International Business Machines Corporation.*

* All other products are trademarks or registered trademarks of their respective companies.

HEURISTIC SCANNERS: ARTIFICIAL INTELLIGENCE?

Righard Zwienenberg

Computer Security Engineers, Postbus 85 502, NL-2508 CE Den Haag, The Netherlands
Tel +31 70 362 2269 · Fax +31 70 365 2286 · Email rizwi@csehost.knoware.nl

Though not explicitly stated, heuristic anti-virus methods have been in use for almost as long as the virus threat has existed. In the 'old days', FluShot(+) was a very popular monitor, alerting the user when it detected 'strange and dangerous' actions. This can be regarded as simple heuristic analysis, because FluShot did not know if the action was legitimate or not. It just warned the user.

During the last couple of years, several resident behaviour-blockers have been developed, used and dismissed again. In most cases, the user finds warnings irritating, aggravating and incomprehensible. The only resident protection they normally use - if any - is a resident scanner. This makes life easier for the users, because the resident scanner clearly indicates that a file or disk is infected by a certain virus when it pops up its box. The disadvantage, which the user doesn't see, is that it does not detect new viruses.

Also, the less popular (but very important) Integrity Checkers may be regarded as heuristic tools. They warn the user when the contents of files have been changed, when files have grown in size, received new time and date stamps, etc. They often display a warning such as: 'file might be infected by an unknown virus' in the case of a changed executable. Especially in a development environment, Integrity Checkers can be really irritating. The user already knows that his executable has changed, because he just changed and recompiled the source code. But how is the Integrity Checker to know that? Using a list of executables to skip is not safe, because a virus may indeed have infected an executable on the list. In that case, the change was not caused by a recompilation. However, the integrity checker can't tell the difference!

Based on these early attempts, the first generation of scanners with minor heuristic capabilities were developed. The heuristics they used were very basic and usually generated warnings about peculiar file date and file time stamps, changes to file lengths, strange headers, etc. Some examples:

```
EXAMPLE1.    COM 12345 01-01-1995   12:02:62

EXAMPLE2.    COM 12345 01-01-2095   12:01:36

EXAMPLE3.    EXE   Entry point at 0000:0001
```

The heuristics of the current, second, generation of scanners are much better. All the capabilities of the first generation scanners are obviously retained, but many new heuristic principles have been added: code analysis, code tracing, strange opcodes, etc. For example:

```
0F POP CS
```

Strange opcode – an 8086-only instruction!

```
C70600019090      MOV WORD PTR [100],9090
C606020190        MOV BYTE PTR [102],90
E9                JMP 0100
```

Tracing through the code shows that it jumps back to the entry point:

```
B9                MOV CX,....
BE....            MOV SI,....
89F7              MOV DI,SI
AC                LODSB
34A5              XOR AL,A5
AA                STOSB
E2                LOOP ....
```

This is obviously decryption code.

A (third generation?) scanner type based exclusively on heuristics exists, performing no signature, algorithmic or other checks. Maybe this is the future, but the risk of a false alarm (false positive, true negative) is quite high at this moment. In large corporations, false alarms (false positives) can cost a lot of time and thus money.

We are not going to examine this scanner type except to note that it may lead us into a new generation, or area, of system examination and protection: *Rule-based Examination Systems.*

RULE-BASED EXAMINATION SYSTEMS

Rule-based systems are as such not a novelty. They already exist, also in the security field. In this field they are often characterised by applying very few, but very broad, rules.

What we are going to look at here are Rule-based Examination Systems seen as large heuristic analysers.

Looking at this sequence of opcodes:

```
B8DCFE            MOV AX,FEDC
CD21              INT 21
3D98BA            CMP AX,BA98
75..              JNE getint21
E9....            JMP wherever
getint21:
B92135            MOV AX,3521
CD21              INT 21
```

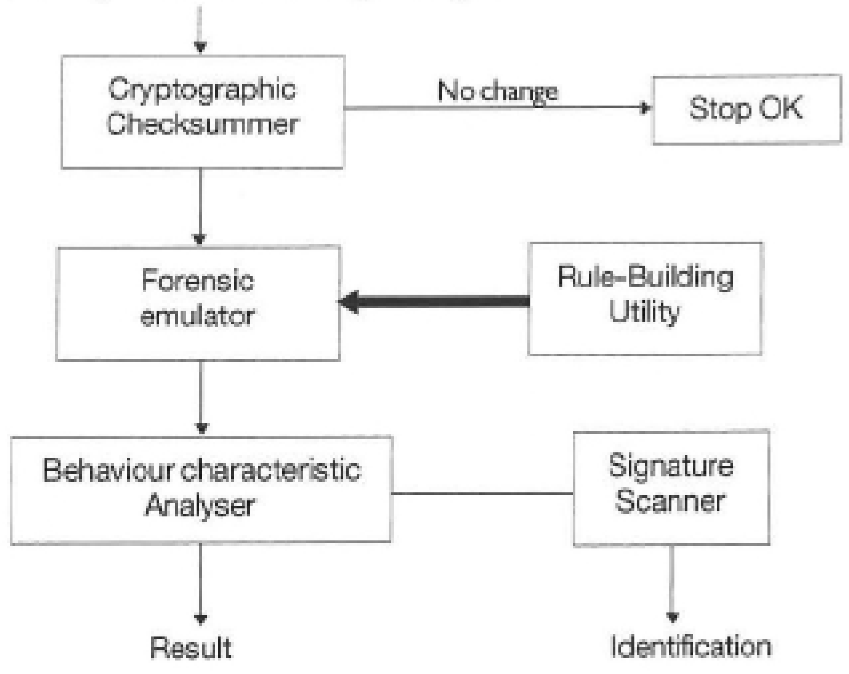
everyone in the field of computer security can see that we may have a virus here (or at least suspicious or badly programmed code). The problem is how to convert something we see in a split second into one or more specific and relevant behaviour characteristics, which we can feed into an examination system. This in turn is able to tell us whether or not we are looking at a virus.

With most of the rules used by the first generation of heuristic scanners, this was not at all difficult. Most were simple comparisons (<,>,==,!=) of the type: 'If a file date exceeds the current date, or is after the year 2000, give an alert'; 'If the seconds field of the file time shows 62 seconds, we can conclude that this is pretty strange and give an alert'. This generation of heuristics, of course, did not have the power to analyse the code in the example shown above.

The second generation of heuristic scanners has more possibilities. Bearing those in mind, defining a rule to cover the example above is not difficult, but imagine a complex decryption routine preceding the actual (virus/Trojan/suspicious) code or – most likely – legitimate code. For example:

```
re-vector int3
re-vector int1
disable keyboard
get int1 offset into di
get int3 offset into si
add counter-1 to si to point to
                 encrypted data
add counter-2 to di to point to
                 encrypted data
get word into ax
perform some calculations with ax
                 to decrypt word
store word
increase counter-1
increase counter-2
look if end of encrypted code has
                 been reached
jmp back if more code to decrypt
enable keyboard...
```

In case this is just one of the instances generated by a complex mutation engine, it will be hard to derive a heuristic rule directly to detect a virus using this engine.



One of the solutions, maybe the best one, is to include a code emulator in the analysing system as illustrated in the figure above, which shows a part of a working network security system. The file to be checked is first given to a checksummer. If the file is already known to the system, a hash code is generated across the file, and this is compared to a stored value. If these are identical, no further action is taken, and the file is declared clean. If not, the file is fed to the emulator, and the results from the code emulation are given to an analyser as described below.

Including a code emulator is possible, and as a matter of fact has already been done. It should have special knowledge of a variety of possible tricks used in malicious code; it should know when to stop emulating (e.g. at the end of a decryption routine); it should be able to realise when anti-debug tricks are used, etc. Both in order to obtain portability, and to avoid obvious pitfalls, it must adhere to one basic and important rule: *Never actually execute an instruction, only emulate it.*

In short, the task of the emulator is first to make sure that the code is decrypted (in case it was encrypted), and then to derive and combine relevant behaviour characteristics to pass on to the analyser, which analyses and organises these behaviour characteristics and compares the results of the analysis with a set of rules.

ARTIFICIAL INTELLIGENCE

From the point of view of the developer it would be nice if such a system were able to learn about behaviour characteristics and generate new rules automatically. If the system bypasses an instance of virus/Trojan/suspicious code because the current rules are no longer sufficient, special examination tools should be able to extract the necessary information from the code in question and create new rules enabling the system to detect this trojan/virus/suspicious code, and hopefully every other form derived from this one. In other words: *Artificial Intelligence.*

For security reasons, these additional tools with their special functionality should not be given to users. Evil-minded knowledgeable persons could use them to do an in-depth disassembly to research the possibilities of bypassing the rules generated by the system. Security through obscurity may not be safe, but it does help...

EMULATOR DESIGN ISSUES

When designing a code emulator for forensic purposes, a number of special requirements must be met.

One problem to tackle is the multiple opcodes and multiple instructions issue:

```
87 C3      XCHG AX,BX
93         XCHG BX,AX
87 D8      XCHG BX,AX
```

The result is the same, but different opcodes are used.

```
PUSH AX    PUSH AX
PUSH BX    MOV AX,BX
POP AX     POP BX
POP BX
```

These give the same result. More than the five different code sequences shown above exist to exchange the contents of registers AX and BX. The technique of expressing the same functionality using many different sets of opcode sequences is used by encryptors generated by polymorphic engines. Some being over 200 bytes in size, they only contain the functionality of a cleanly coded decryptor of 25 bytes. Most of the remaining code is redundant, but sometimes seemingly redundant code is used to initiate registers for further processing.

It is the job of the emulator to make sure that the rule-based analyser gets the correct information, i.e. that the behaviour characteristics passed to the analyser reflect the actual facts. No matter which series of instructions/opcodes are used to perform 3D02h/21h, the analyser only has to know that the behaviour of that piece of code is:

Open a file for (both reading and) writing.

On the one hand, this may not seem that difficult. Most viruses do perform interrupt calls, and when they do, we just have to evaluate the contents of the registers to derive the behaviour characteristic. On the other hand, this is only correct if we talk about simple, straightforward viruses. For viruses using different techniques (hooking different interrupts, using call/jmp far constructions) it may be very difficult for the emulator to keep track of the instruction flow. In any case, the emulator must be capable of reducing instruction sequences to the bare functionality in a well-defined manner. We call the result of this reduction a *behaviour characteristic*, if it can be found in a pre-compiled list of characteristics to which we attach particular importance.

Another problem is that the emulator must be capable of making important decisions, normally based on incomplete evidence (we obviously want to emulate as little code as possible before reaching a conclusion regarding the potential maliciousness of the software in question).

Let us illustrate this with a small example:

```
MOV AX,4567
INT 21
CMP AX,7654
JNE jmp-1
JMP jmp-2
```

This is an example of an 'Are you there?' call used by a virus. When tracing through the code, the emulator obviously does not know whether jmp-1 or jmp-2 leads to the code which installs the virus in case it is not already there. So, should the emulator continue with the jmp-1 flow or the jmp-2 flow? Now, a simple execution of the code will result in just one of these flows being relevant, whereas a forensic emulator must be able to follow all possible program flows simultaneously, until either a flow leads to a number of relevant behaviour characteristics being detected, at which time the information is passed to the analyser, or a flow has been followed to a point where one of the stop-criteria built into the emulator is met. The strategy used in this part of the emulator is a determining factor when it comes to obtaining an acceptable scanning speed.

Hopefully, this has illustrated some of the problems associated with designing a forensic emulator. It is a very difficult and complex part of this set-up.

Once the emulator has finished its job it passes information, a list of behaviour characteristics which it has found in the code, on to the analyser.

BEHAVIOUR RULES

Before the analyser is able to compare the behaviour characteristics found by the emulator to information in its behaviour database, this database needs to be defined. Assume that we have a COM and an EXE file infecting virus with the following behaviour:

```
!     MODIFY FILE ATTRIBUTE REMOVING READ-ONLY FLAG
!     OPEN A FILE FOR (BOTH READING AND)WRITING
!*    WRITE DATA TO END OF FILE
!*    MODIFY ENTRY POINT IN HEADER or WRITE TO BEGINNING OF
      FILE
-     MODIFY FILE DATE AND FILE TIME
-     CLOSE FILE
-     MODIFY FILE ATTRIBUTE
```

If we want to develop a behaviour rule for this virus, it will look like this:

```
1.   MODIFY_FILE_ATTRIBUTE + OPEN_FILE
     + WRITE_DATA_TO_EOF +
     MODIFY_EP_IN_HEADER

2.   MODIFY_FILE_ATTRIBUTE + OPEN_FILE
     + WRITE_DATA_TO_EOF +
     WRITE_DATA_TO_BOF
```

where rule 1 is a rule for the EXE-file, and rule 2 for the COM-file.

Since a lot of viruses and virus source codes are widely available, a number of different instruction sequences resulting in this functionality will probably show up. Normally, derived viruses contain minor changes to bypass a single scanner by just changing the order of two or more instructions, but sometimes larger code sequences can be changed without changing the functionality of the virus. It is trivial to change the code, so it will first modify the entry-point in the header or change the start-up code, and afterwards write the virus code. In order to detect these changes (variants) the next rules may be added:

```
3.   MODIFY_FILE_ATTRIBUTE + OPEN_FILE
     + MODIFY_EP_IN_HEADER +
     WRITE_DATA_TO_EOF @CODE LINE =

4.   MODIFY_FILE_ATTRIBUTE + OPEN_FILE
     + WRITE_DATA_TO_BOF +
     WRITE_DATA_TO_EOF
```

Another example (an MBR infector):

```
-    PERFORM SELF CHECK
!    HOOK INT13
!    BECOME RESIDENT
!    INTERCEPT READ/WRITE TO MBR
!    READ MBR
-*   WRITE MBR TO OTHER LOCATION
!*   WRITE NEW MBR
```

Rule:

```
HOOK_INT13 + INTERCEPT
READ/WRITE_TO_MBR + WRITE_NEW_MBR
```

The signs in front of the descriptors in the examples above hint at the weighing procedure used by the analyser to attach significance to the behaviour characteristics supplied by the emulator. A '-' means that the characteristic does not have to be present, an '!' that it must be present (but does not in itself indicate malicious code). A '*' indicates a high weighing value. Thus '-*' means that the characteristic does not have to be present in the sequence of actions, but if it is, this is a highly important fact.

If rules 1 - 4 above are examined more closely, it can be concluded that they describe behaviour found in a number of viruses from different families.

A single behaviour rule may detect an unlimited number of viruses. That is the power behind using behaviour characteristics. While at present we in most cases need a new signature or new (changed) algorithm to detect a new variant of a virus or a new virus family, the behaviour characteristics will continue to do their work. This is extremely important, because it removes the necessity for the virus researcher and the anti-virus developer to react to a new virus unless it is technologically innovative. And those are few and far between.

Of course, some viruses will be developed which will not be caught by any of the rules in the behaviour database. These must be taken care of just like we do right now with any new virus; but instead of creating a signature, we create a new rule.

With a little luck, a new virus behaves like a virus already covered by a rule. If we attach a level of importance to each part of a behaviour characteristic, we can use this in the analyser to arrive at a conclusion. Depending on the level of importance of each individual component of a behaviour characteristic detected, the system may decide to give a message to the user, such as 'may be infected by an unknown virus', or 'suspicious code'.

The reason for attaching a level of importance to each individual part of a behaviour characteristic is that it makes it easier to sort out cases where combinations of individually innocent behaviour characteristics put together constitute malicious code – or vice versa. Filedate, from *Norton's Utilities,* is able to change file date and file time; as a matter of fact, this is the purpose of the utility. The ATTRIB command is developed to change file attributes. Evidently, changing file attributes is in itself insufficient evidence of malicious behaviour. A virus needs to write to a file as well. So a file write is mandatory for code to be considered suspicious and is heavily weighted. A change of attributes is not that important, and thus given a lower weighting.

If the user so wishes, the file or part of the (decrypted) code on which the analysing system triggered can be checked by a signature scanner to see if a virus can be identified.

CREATING RULES AUTOMATICALLY

An important part of the system is a *Rule Building Utility.* Whenever a new virus or Trojan emerges, it may be processed by this utility, which is similar to the emulator, albeit with some important differences. The emulator only collects behaviour information without knowing anything about the importance of a particular type of behaviour, or if the behaviour is suspicious.

The Rule Building Utility has to learn the level of importance of behaviour characteristics, has to know which behaviour is mandatory for a virus or Trojan, which behaviour is used by a virus but may be omitted, etc. Because research and development time is very expensive, the utility must be able to remember this for similar behaviour characteristics, and only ask for additional unknown information when needed, saving the researcher valuable time.

```
Behaviour A:                Behaviour B:

SEARCH FIRST FILE           SEARCH FIRST FILE
DELETE FILE                 DELETE FILE
SEARCH NEXT FILE            CREATE NEW FILE
                            WRITE CODE INTO FILE
                            SEARCH NEXT FILE
```

When rules have been defined for behaviour B and a file (behaviour A, which was reported being suspicious) is processed, the utility must be able to realise that this behaviour is not as indicative of potential maliciousness as behaviour A. As a matter of fact, if behaviour A is taken on its own, it might well be a DEL *.* command.

At first, the utility will, ask for input frequently, because it needs to build up its database. However, over a period of time this type of utility should make life easier for the researcher.

CONCLUSION

The number of viruses is increasing rapidly: this is a known fact. The time will soon arrive when scanning using signatures and dedicated algorithms will either use too much memory or just become too slow. With storage media prices dropping fast, lots of systems now come equipped with very large hard disks, which will take more and more time, and thus money, to scan using traditional techniques. A properly designed rule-based analysing system feeding suspicious code to a scanner, which can identify the suspicious code as a known virus or Trojan, or perhaps dangerous code needing further investigation, is bound to save a lot of time.

Although it is impossible to prove that code is not malicious without analysing it from one end to the other, we in *Computer Security Engineers Ltd* believe it possible to reduce significantly the time used to check files by using all the available system knowledge instead of only small bits of it, as it is done today. Using virus scanning as the primary, or in many cases the only, anti-virus defence is an absurd waste of time and money, and furthermore blatantly insecure!

ABOUT CSE

Computer Security Engineers Ltd is one of the pioneers of anti-virus system development. The anti-virus system PC Vaccine Professional was first published in 1987, and since the start of 1988 a new version has been published each and every month. From 1988, cryptographic checksumming was introduced as the primary line of defence, scanning as the second. In 1992, the emphasis shifted, and behaviour blocking was introduced as the first line of defence, followed by checksumming and – in the case of an alarm from one of these countermeasures or to examine incoming diskettes – scanning for known viruses. Most recently, the basic philosophies underlying PC Vaccine professional, or PCVP as the system is also known, were expanded into a powerful and easily-maintained network perimeter and in-depth defence based on the well-known military tenets of: (1) keep them out and (2) if you can't keep them out, find and destroy them as fast as possible.

VIRUS DETECTION – 'THE BRAINY WAY'

Glenn Coates & David Leigh

Staffordshire University, School of Computing, PO Box 334, Beaconside, Stafford, ST18 0DG, UK
Tel +44 1782 294000 · Fax +44 1782 353497

ABSTRACT

This paper explores the potential opportunities for the use of Neural Networks in the detection of computer viruses.

Neural computing aims to model the guiding principles used by the brain for problem solving, and apply them to a computing domain. It is not known how the brain solves problems at a high level; however, it is widely known that the brain uses many small highly interconnected units called 'neurons'.

Like the brain, a neural network can be trained to solve a particular problem or recognise a pattern by example. The outcome is an algorithm-driven recogniser which does not exhibit the same behaviour as a deterministic algorithm. According to the way in which it has been trained, it may make 'mistakes'. That is, it may declare a positive result for a sample which is actually negative, and vice-versa. The ratio of correct results to incorrect ones can usually be improved by more or better training.

Can such pattern recognition be harnessed to the use of virus detection? It could be argued that the characteristics of virus patterns, no matter how they are expressed, are suitable subjects for detection by Neural Networks.

INTRODUCTION

The received wisdom is that neural computing is an interesting 'academic toy' of little use, apart from modelling the animal brain. If this is true, then it is surprising that 7 out of 10 of the UK's leading blue chip companies are either investigating the potential of neural computing technology or are actually developing neural applications [Con94]. If leading edge companies are prepared to spend money on this 'academic toy', then maybe there are advantages to be gained from its use.

Without investigating new techniques (for example heuristic scanning), one must accept that the rapid rise in new viruses will exert a heavy speed penalty from existing virus scanners. As a result of this rise in virus numbers and sophistication, there will be an increasing conflict between acceptable speed and acceptable accuracy. It is easy to become complacent and rely on increasing processor power to bail us out of this problem, but processor design is increasingly becoming a mature technology.

What follows are the results of a feasibility study into the utilisation of neural networks within the field of virus detection.

WHAT IS A NEURAL NETWORK?

The workings of the brain are only known at a very basic level. It contains approximately ten thousand million processing units called neurons, each of these neurons is connected to approximately ten thousand others. This network of neurons forms a highly complex pattern recognition tool, capable of conditional learning. Figure 1 illustrates a model of the biological neuron alongside its corresponding mathematical model.

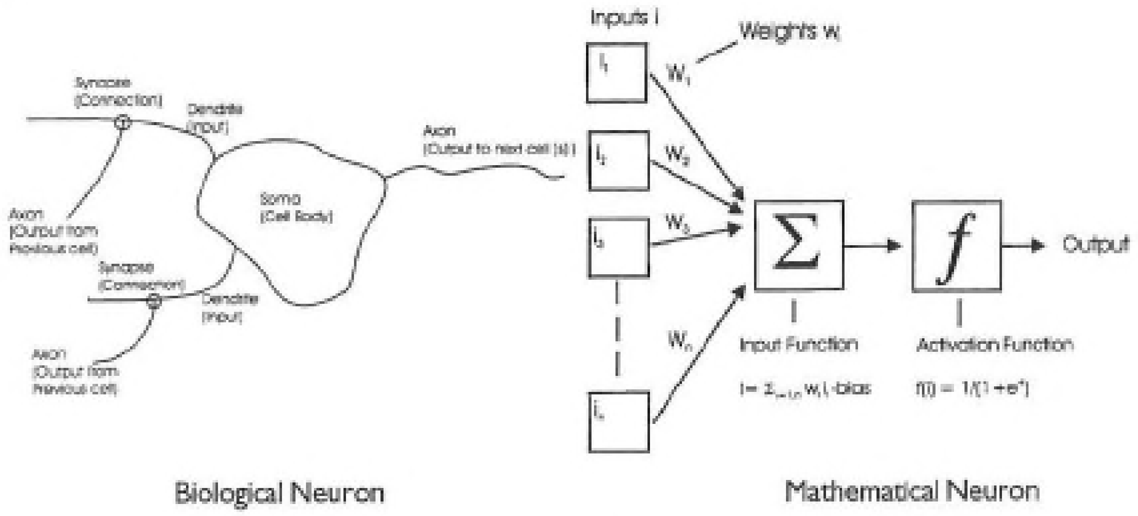


Figure 1

The individual neuron is stimulated by one or more inputs. In the biological neuron, some inputs will tend to excite the neuron, whilst others may be inhibitory. That is to say, some carry more 'weight' than others. This is mirrored in the mathematical model via the use of a 'weighting mechanism'. The neuron accumulates the total value of its inputs, before passing through a threshold function to determine its final output. This output is then fired as an input to another (or a number of) neurons, and so on. In the biological neuron, the axon performs the threshold function. The mathematical model would typically use a sigmoid function or a simple binary 'yes/no' threshold function. The reader is referred to [Mar93] for further discussion.

NEURAL NETWORK DEVELOPMENT

When approaching a problem using a neural network, it is not always necessary to know in detail what is to be done before planning its use. In this sense, they are quite unlike procedurally-based computer programs, which have to be written with a distinct goal in mind if they are to work properly. It is not even like a declarative program, for the same rule should apply. It is, perhaps, more like an expert system, where the outcome depends on the way in which an expert has answered a pre-defined series of questions.

In this approach, a 'standard' three-layer neural network is constructed using the 'back propagation' learning algorithm. The architecture consists of an input layer, a hidden layer, and an output layer. Training is carried out by submitting a 'training set' of data to the network's input, observing what output is given, and adjusting the variable weights accordingly. Each neuron in the network processes its inputs, with the resultant values steadily 'percolating' through the network until a result is given by the output layer. This output result is then compared to the actual result required for the given input, giving an error value. On the basis of this error value, the weights in the network are gradually adjusted, working backwards from the output layer. This process is repeated until the network has 'learnt' the correct response for the given input [DTI95]. Figure 2 illustrates this.

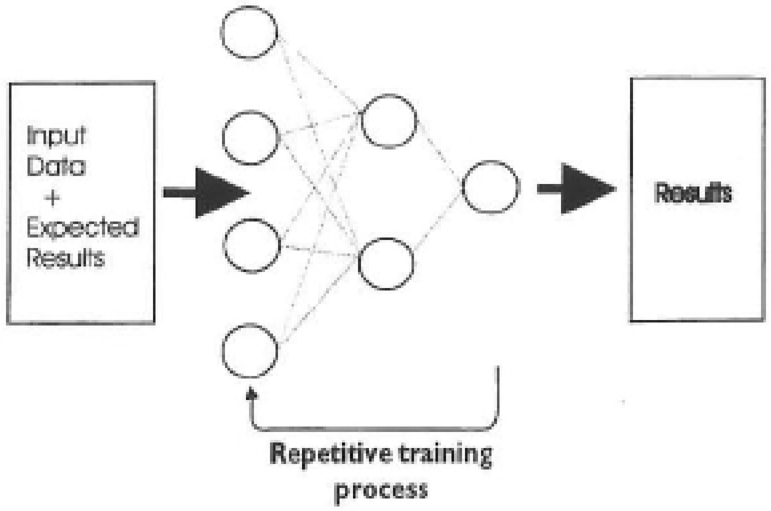


Figure 2

In this instance, the inputs represent the virus information, or other data concerning a virus-infected file. There are only two possible outputs, corresponding to 'possible virus found' and 'file appears to be OK'. The training data is divided into two classes, one containing the data for an infected file; and the other, uninfected files. When a suitable output is generated for the training data, the network is checked with a separate 'validation set'. If the output for the validation set is not acceptable, it is merged with the original training set and the entire process is repeated . This process is described schematically in figure 3.

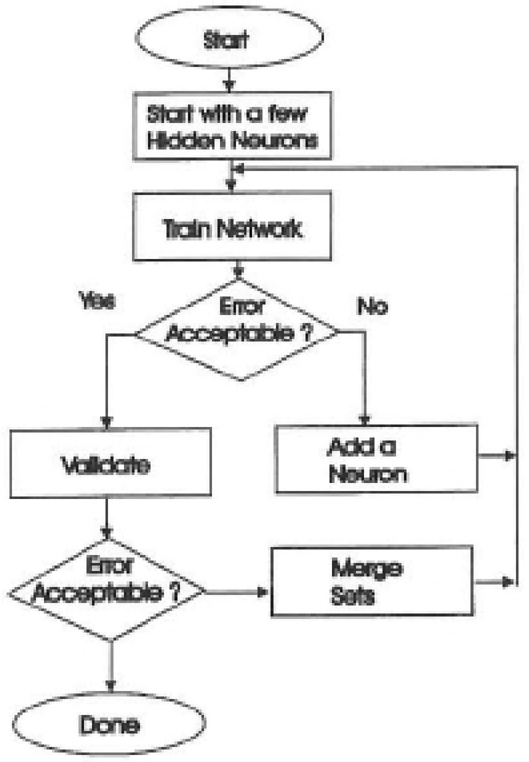


Figure 3

The result should be a very robust fuzzy recogniser capable of coping with unseen data. Because neural networks can process deeply hidden patterns, some have provided decisions superior to those made by trained humans.

EXISTING SYSTEMS

In 1990, a neural network was developed which acted as a 'communications link' between the mass of virus information available and end-user observations. By answering a set of standard questions regarding information on virus symptoms, the virus could be classified, and a set of remedies given. Due to the nature of neural networks, the system could cope with incomplete and erroneous data provided by the end user. Even when faced with a new mutation, the system still gave suitable counter-measures and information. See [Gui91] for a full discussion.

IDENTIFICATION OF VIRUS CODE PATTERNS VIA NEURAL NETWORKS

A neural network could be constructed to learn the actual machine code patterns of a specific virus. However, as most viruses are mutations of existing viruses, a network could be made to identify a virus family. This carries the advantage of being capable of identifying future variants. This would result in a set of sub-networks linked together to provide the end solution.

At the lowest level this could be done at the bit level. Figure 4 illustrates this.

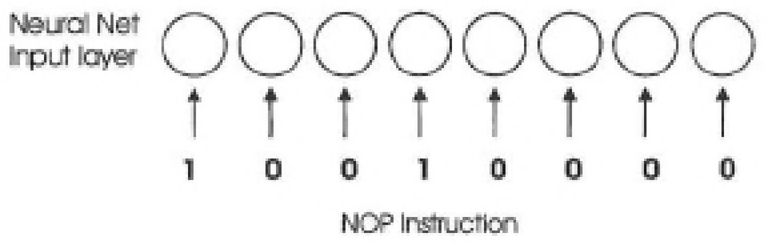


Figure 4

Although recognition at this level would be very difficult (if not impossible for a human) a neural network would be capable of it. The only limiting factors would be the volume and quality of the training data. The number of input neurons for a ½K virus code segment with a one-neuron output would be 4096. Given this, according to the 'geometric pyramid rule', the number of neurons in the hidden layer would be 64.

The number of virus samples for effective recognition would be in the region of at least 525,000. This figure should then be trebled for the number of non-infected files. Others would argue far more, due to the problems associated with false positives.

At a higher level, the input data could be presented at the byte level, where each byte would correspond to a single input neuron. In this context, the number of hidden neurons would be reduced to 22, and the number of virus samples would be at least 23,000. Again, the same applies for the number of non-infected files. This figure could be reduced further by pre-processing the code segment by extracting operand information, which could also increase accuracy and training time.

The *British Technology Group*, with the involvement of *Oxford University*, conducted research into such a solution. Although no formal documentation was produced, the results are believed to be negative.

From this, it can be seen that the use of neural networks in virus detection only seems practical at a high level. After all, a virus expert armed with a 'Virus Detection Language' and a 'Generic Decryption Engine' can provide a 100% accurate scanning result with advanced polymorphic viruses such as Pathogen in a relatively short period of time.

A NEURAL NETWORK POST-PROCESSOR

Rather than utilising a neural network to solve the virus alone, one could be used to process high level information, for example, that generated by a heuristic scanner.

Currently, most heuristic scanners use a form of emulation in order to determine the behaviour of a program file. Should that program appear to execute a suspicious activity, a 'flag' is set indicating this. However, some of these flags indicate more virus-like activity than others. In order to solve this problem, the flags are weighted via a score. Therefore, a flag indicating a 'suspicious memory reference' may be given more weighting than a flag indicating an 'inconsistent EXE header'. The total weights of the set flags are computed, and if a set threshold value is met, the heuristic scanner issues a suitable warning.

In the example of a well-known heuristic scanner, 35 of these flags are used. The weights are applied on an experimental basis. Initially, the weights are applied using a 'best-guess' approach, based on the virus experts' knowledge. The results of this are then tested on a virus collection and on a clean set of files. The results are analysed, and the weights adjusted accordingly. This cycle continues until satisfactory results are obtained. Figure 5 illustrates this cycle.

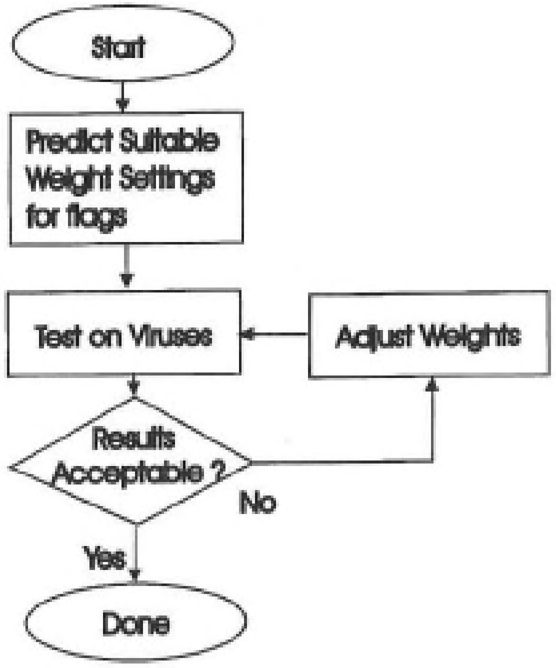


Figure 5

This process will probably increase in complexity over the next few years. In the above example, the number of flags could literally double due to the increase in knowledge, new techniques employed by the virus writers, and further development of heuristic scanners. It is imminent that the cycle of adjust, test, re-adjust will become far more complex and time-consuming. For example, why should flag-x be given a weight of 8, and not 7 or 9, and flag-y be given a weight of 1, and not 2?

Already, one can see that the illustrated cycle is very similar in nature to that used in neural network training. Indeed, a neural network could be used in place of the weighting mechanism and bias imposed by the virus expert. Based on the results of other neural network applications, the results should be very accurate, because the neural network will 'learn' the 'optimum' weights. The human element is removed, and the entire learning process is automated.

In terms of network size, the number of input neurons would be 35, with 6 hidden neurons, and 1 output neuron. In theory, the minimum number of infected file samples required for training would be at least 432. However, there would be no detrimental effects from training the network with higher samples, in order to reflect current virus numbers.

CONCLUSIONS

Neural computing is no longer seen as a pure academic subject. Indeed, many companies are now looking towards the use of neural networks as serious tools. Many systems are currently in use, with very high success rates.

It has been found that it may be feasible to use neural computing technology in the virus detection field. However, at a low level the results are unclear. There seems to be greater accuracy using deterministic techniques.

Using a neural network as a pre-/post-processing tool could offer a powerful addition to the virus expert's toolbag. Just one example is with the heuristic scanner. The authors believe other uses will also exist.

ACKNOWLEDGMENTS

As Bernard of Chartres said, echoed by Sir Isaac Newton: 'If [we] have seen further, it is by standing on the shoulders of giants'. The assistance given by the following people is gratefully acknowledged: Jan Hruska, Frans Veldman, Alan Solomon, Martin Slade, Robert Mortimer and Michael Twist. The continuing support of the staff at *Staffordshire University* and at *Visionsoft* has also been gratefully appreciated.

REFERENCES

[Con94] 'Adopting The Neural Approach', *Control Magazine*, Issue 5, March/April 1994.

[DTI95] *UK Department Of Trade And Industry*, Neural Computing Technology Programme, 1995.

[Gui91] Dr. Daniel Guinier, 'Computer 'virus' identification by neural networks', *SIGSAG*, 1991.

[Mar93] Timothy Masters, 'Practical Neural Networks in C++', *Academic Press*, 1993. ISBN 0-12-479040-2 Further Reading.

'Neural Computing – an introduction', R Beale and T Jackson, *IOP Publishing*, 1990. ISBN 0-85274-262-2.

Vesselin Bonchev, Future Trends in Virus Writing, *Proceedings of the Fourth International Virus Bulletin Conference*, 1994.

Glenn Coates and David J. Leigh, 'Virus Detection using a Generalised Virus Description Language', *Proceedings of the Fourth International Virus Bulletin Conference*, 1994.

DATA SECURITY PROBLEMS ASSOCIATED WITH HIGH CAPACITY IDE HARD DISKS

Roger Riordan

Cybec Pty Ltd, PO Box 205, Hampton, VIC 3188, Australia

Tel +61 3 521 0655 · Fax +61 3 521 0727 · Email riordan@tmxmelb.mhs.oz.au

INTRODUCTION

Every article on viruses starts with the advice 'Before running any anti-viral or integrity checking software you should always boot from a clean DOS floppy disk, to ensure there are no stealth viruses in memory where they could interfere in the test'. Just before we sent out our February update my eye was caught by a reference to problems being caused by special drivers used with large IDE drives. By chance we had just bought a 1G hard disk for a home PC, so I took a copy of VET home and installed it. Then I booted from a floppy. When I ran an uninstalled copy of VET it was unable to check drive C. This was bad enough, as it meant the traditional advice was useless. But then I ran VET from the reference disk I had just made. It announced 'Master Boot Record has been changed; would you like to replace it?' Clearly we had a problem.

This paper will discuss how the problem has arisen, the nature of the problem, what it means to users, and what they can do to avoid the dangers these drives have introduced.

BACKGROUND

Engineers traditionally regard a safety factor of 10 as conservative; if you are designing a bridge, or a building, you think of the largest conceivable load and then design the structure to handle something 10 times as big. This thinking has carried over into the computer industry; when IBM designed the PC most personal computers had 64K of memory, so they allocated a luxurious 640K. Unfortunately the unprecedented speed of development has rendered this thinking dangerously inadequate. Sizes of memory chips and disk drives have been doubling every two to three years, and so a safety factor of 10 will be used up in only three to four years. Given that it may well take a couple of years to bring a new design to market, the old joke about the computer shop with the banner 'New Model', and the salesman saying 'It's on sale; it's superseded', is all too likely to be true.

When Microsoft designed the disk handling logic for MSDOS they had to decide how to allocate all the various steps on the way from the user asking to read a particular file to the disk controller being told to read a particular sector. The first mistake they made was to make DOS do the low level sector handling, instead of leaving it to the BIOS, and the next was in allocating disk parameters to registers in the call to Int 13, which provides the interface between DOS and the BIOS [1].

The scheme they used was:

AH	Read/write command	AL	Number of sectors to read
CH	Number of first cylinder	CL	Number of first sector
DH	Starting Head	DL	Drive

This was OK for floppies, but when IBM introduced the XT (in about 1985) this allocation meant that the number of the starting cylinder could not be more than 255. This was too small, so Microsoft decided to steal the top two bits of CL, and add them to CH, increasing the maximum number of cylinders to 1024, but limiting the maximum number of sectors to 63. This was messy, but not critical. The BIOS extensions to handle the hard disk were in ROM on the disk controller card.

Then, almost immediately, the AT was introduced, and the BIOS was extended to handle the hard disk directly. The extension loaded the various parameters into registers in the disk controller, using I/O instructions to port addresses in the range 1F1 to 1F6, and finally passing a read/write command to port 1F7. The allocation used was:

1F2	Number of sectors to read	1F3	Number of first sector
1F4	Number of 1st cylinder (low)	1F5	Number of 1st cylinder (high)
1F6	Head (bits 0-3 only)	1F7	I/O command.

This arrangement was fine in itself, as it would handle any conceivable size of drive, but it was seriously incompatible with the Int 13 interface, as the maximum values which could pass unmodified through both interfaces were:

Number of sectors	63	(6 bits from CL in Int 13)
Number of cylinders	1024	(8 bits from CH, two from CL in Int 13)
Number of heads	16	(4 bits of 1F6 in drive controller)
Length of sector	512	(DOS standard)

Thus the maximum disk size had effectively been limited to 16*1024*63*512 = 528.482 Mbyte. This was not noticed at the time, as hard disks were typically 5 or 10 Mbytes, but by the end of the 80s hard disks with up to 1 Gbyte were becoming increasingly common. At first these used the SCSI interface, but this required a non-standard BIOS and a relatively expensive interface card. Meanwhile the far simpler IDE interface had become standard, and had improved to the point where the SCSI interface offered little, if any, improvement in performance, and there was a strong incentive to devise a way for big drives to be used with it, without having to replace the BIOS.

In 1994, several companies devised ways around this limitation, and inexpensive high capacity (ie greater than 528 Mbyte) IDE drives began to appear [2]. Most of these use variations of a scheme referred to as Extended CHS (Cylinder, Head, Sector) addressing, and replace the normal Master Boot Record with a special boot sector. This loads an extension to the BIOS from the normally unused area following the MBR. This copies itself to the top of normal memory (or to an area in high memory), hooks Int 13, and then loads a normal MBR. After this the PC boots normally, but all calls to Int 13 are caught, and the parameters translated before they are passed on to the drive controller.

When IBM designed the XT they put the MBR in sector 1 of head 0, cylinder 1, and left the whole of the rest of this track empty. However many 'compatibles' put the DOS boot sector in sector 2, with the File Allocation Table (FAT) following immediately after it. This incompatibility was not a significant problem until 1988, when the unknown authors of the Stoned virus realised this unused area made an ideal place for their virus to hide the original MBR.

This was fine on true clones, but when Stoned infected one of these non-standard PCs it overwrote the FAT, with disastrous results. Indeed I became involved in the AV industry only because the college where I worked had a lab full of Olivetti M24s, and these were crashing faster than the technicians could re-install the software on them.

Stoned, and its many descendants, have provided a strong incentive to the industry not to put anything important in track zero, and it has come to be regarded as a convenient working area. However, as we have noted, the special device drivers for the big IDE drives are loaded here, and so the old incompatibility, which rendered Stoned so serious, has been re-introduced, without any warning to the unsuspecting customers buying these drives [3].

THE DRIVERS

We believe that at least three drivers are in use, but only two seem to be common in Australia. These are:

1. Disk Manager, written by *Ontrack Computer Systems*, and sold with Western Digital drives, and
2. EZ-Drive, written by *Micro House International, Inc*, and sold with Conner drives.

Both have a non-standard MBR (in the normal location in track 0, head 0, sector 1), and store the body of the driver in track zero, head zero, but with EZ-Drive the rest of the drive is arranged normally, whereas Disk Manager moves the whole of the normal disk structure down one head (so that the normal MBR is in Cylinder 0, Head 1, Sector 1, and the DOS boot sector is in Cylinder 0, Head 2, Sector 1, and so on. For want of a better name we refer to the non-standard MBR as the Extended Boot Record (EBR). Thus in this paper, the EBR is the sector loaded first, while the MBR is the sector loaded after the driver has been installed. In normal use the EBR is invisible, and utilities (and viruses) will only see the MBR, which will appear to be in the normal location.

Disk Manager occupies sectors 2 to 30, leaving sector 7 free for Stoned (the designers had apparently not heard of the many viruses which use other sectors), and sector 63 has some data labelled ‘Disk Tables’. It is likely this location was chosen because it is relatively unlikely to be overwritten by accident.

EZ-Drive occupies sectors 2 to 14, with no provision for viruses. Sector 2 contains the ‘normal’ MBR. This has no boot program at all; everything but the actual partition record and sector marker is zeroed. There are several tables which could be drive data. All these sectors are used by known viruses.

When they are active both drivers ‘stealth’ the EBR, and show the MBR, with apparently normal partition information. Thus any virus which infects the EBR, and is compatible with the driver, will be hidden from normal anti-viral software.

Both drivers offer to ‘boot from a floppy’ after they have installed themselves (and any compatible viruses), so that you can then access the hard disk without running anything else from it. The On Track driver is already compatible with quite a few viruses, and will protect them from older AV software, and the dark side will probably try to produce more ‘compatible’ viruses to take advantage of this protection. But at least if the virus is compatible with the driver it means that the virus can be removed by the normal means after booting from a normal boot disk, without the AV software having to know about the driver.

CHANGE OF DRIVE LAYOUT

When a PC with one of these drives is booted from a floppy, not only the MBR, but also the actual physical layout of the drive, will appear to have been changed. Thus for two test drives:

Drive	Booted from floppy		Booted normally	
	Heads	Sectors/Track	Heads	Sectors/Track
Conner/Micro House 540Mb	15	17	15	63
Western Digital/Ontrack 1Gb	6	55	16	63

This discrepancy is not too serious for the Micro House driver, as all the relevant information is contained in the first 17 sectors, and these can be read using Int 13. However the On Track driver appears to store the vital drive parameters in head zero, sector 63, and the normal MBR in head 1, sector 1, as seen when the driver is resident. But if you have booted from a floppy and want to read these using Int 13, you would have to read head 1, sectors 8 & 9.

In general the new address is not immediately obvious, though it can be determined either by interrogation of the disk controller, using poorly documented commands, or by reading successive sectors until the read command fails, and then moving to the next head. Fortunately however, the function to read using direct I/O commands can read an arbitrary number of consecutive sectors, regardless of the layout. Thus if we read 64 sectors, starting with head 0, sector 1, we read the whole of track zero, including the drive parameters, and also the normal MBR.

(As an interesting aside, despite what some utilities may report, it would appear that the initial values for the Western Digital drive are almost certainly the true physical values; it is difficult enough to imagine fitting three platters in a drive of that size, let alone eight!)

DISASTER CONTROL

Disk Manager has provision for generating a boot disk which loads the driver as a device driver, so that you can access the hard disk without running any software from it. Provided the EBR (and probably also the MBR) are intact this enables you to access the hard disk even if the device driver has been severely damaged. This feature is not mentioned in the documentation, but is offered as a menu option during installation.

EZ-Drive apparently cannot generate a boot disk of this type. However the installation menu does offer 'Antiviral Boot Sector Code'. The help menu for this says:

> This feature will defend your system against boot sector viruses. This does not include all viruses, so do not rely exclusively on this or any single virus protection package. If a virus is ever detected, a message will be posted, the virus will be removed, and the machine will reboot.

As we will see, the value of this is problematic.

No documentation is supplied with EZ-Drive. The documentation supplied with Disk Manager gives no warning about incompatibility problems with either viruses or low level utilities.

EFFECT OF VIRUSES

The hard disk boot sectors can get infected in four different situations. These are:

i. The PC is booted directly from an infected floppy

ii. An infected floppy is inserted when the driver, after being loaded from the hard disk, offers to boot from a floppy

iii. The PC is booted from an On Track special boot disk

iv. A file infected with a multi-partite virus is run.

So far as the EZ-Drive driver is concerned these all have effectively the same result. In all cases we have tried, or can think of, the driver will be over-written and the PC rendered unbootable. In our tests the 'Antiviral Boot Sector Code' was never invoked, and it would seem that the key clause in the help menu is 'If a virus is ever detected ..'.

With the *On Track* driver, in case i. the virus will infect the EBR in head 0, sector 1, and will normally overwrite one or more sectors in the same track. Many viruses will destroy the driver, but Stoned and other viruses which only overwrite sector 7 will fit in the hole provided for them, and the PC will continue to boot. In this case the, virus will be installed in memory before the driver, and the infected sector will be hidden from normal AV software.

In all the other cases the virus will infect the normal MBR in head 1, sector 1, and overwrite one or more following sectors. As these are unused, the PC will boot normally. In this case, the virus will be installed in memory after the driver.

In either case, if the PC is able to boot, the virus will be able to spread.

If the EBR is infected, normal AV software will be able to remove the virus, after booting from a clean floppy, but the PC will remain unbootable until the device driver is replaced. This can most easily be done if a copy of track zero is saved during installation. Both drivers offer some facilities for repairing damage, but these have not yet been fully evaluated.

If the MBR becomes infected normal AV software should be able to remove the virus, provided it can disable the virus in memory. However if the PC is clean booted the AV software will not be able to find the virus, unless the boot disk installs the On Track driver.

It would appear that DOS boot sector infectors should operate normally, and should not pose any special problems.

Given this range of possibilities, disinfection of these drives poses considerable problems, especially when the software is being operated by unskilled users.

THE PROBLEM

If you are responsible for any PCs fitted with one of these drives, you should be aware of the following dangers.

1. Most anti-viral and integrity checking programs recommend that you boot from a clean DOS disk before running them, to ensure that no stealth viruses are in memory. However if you do so you will be unable to access your hard disk at all. Furthermore the MBR (and even the drive parameters) will be completely different, and if the software has taken copies of the normal MBR it may offer to replace it, with disastrous results.

2. Most boot sector viruses will destroy the driver, rendering the contents of the hard disk inaccessible
3. Some viruses will allow the PC to operate normally, but will be screened from normal AV software by the drivers
4. Some utility software uses the unused area on track zero as a work space, again making the hard disk inaccessible. We have already recovered several drives damaged by 'disk optimisers'.

HOW VET HANDLES THESE DRIVES

When VET 8.3 checks a hard disk, it reads the MBR using Int 13 in the normal way, and then it reads it again, using direct port access. This bypasses the driver (and probably many security products) and returns the true contents of head zero, sector 1. VET refers to the value obtained this way as the Extended Boot Record, or EBR. It then compares the results, and if they differ, assumes that it is dealing with a 'big' drive. In this case it reads the whole track under head zero, and saves it. VET also recognises the MBR and the EBR for both the On Track and Micro House drivers.

VET will detect and warn if the PC has been booted from a floppy, and has a facility to compare track zero with the template, and if necessary replace it. If VET is asked to check the boot sectors, and detects a virus in the EBR, and a template for the clean track is available, it will automatically offer to replace the damaged track.

COMPATIBILITY PROBLEMS

Windows 95, and probably Windows NT, issue a warning when VET uses direct I/O to read the EBR. This is not fatal, but may alarm users.

Some early PCs may give erroneous results when VET reads the EBR, and could be classified erroneously as big drives, and it is possible that some security products may either complain, or confuse VET.

THE SOLUTION

If you are in charge of any PCs using these drives (and which do not have the new motherboards described in the next section) there are a number of steps you should take. These include:

1. Prepare a rescue disk for these PCs. At the minimum this should include a copy of the special driver, the same version of DOS, and any utilities required to get your system running. If your anti-viral software permits it, make a separate rescue disk for each PC, and use it to save a copy of track zero on the disk. (And make sure you label the disks clearly, so you know which PCs they belong to.)

   I would strongly advise against buying any drive using a driver which does not permit you to prepare such a disk.
2. Ensure that all anti-viral, security and utility software in your organisation is aware of these drives. Be particularly wary of users with their cherished 'private' utilities hidden away.
3. Warn the users (again!) about the dangers viruses pose to their system, and ensure that up-to-date AV software is readily accessible at all times AND USED!
4. Review your backup procedures!

THE FUTURE

An ironic feature of this affair is that the problem these drivers overcome is a short term one. As well as the Extended Cylinder, Head, Sector addressing already described, the large drives can all be addressed using a second addressing mode, known as Logical Block Addressing (LBA).

In this system, which Microsoft should have introduced years ago, all addresses are passed to the drive simply as an absolute address. Instead of fiddling around with drive parameter tables, cylinders, heads, sectors and all the rest, DOS would simply ask 'Read 73 blocks, starting at block 34,156'. Thus DOS has much less book-keeping to worry about, as all the dirty details of mapping the request to the physical arrangement of the disk can be left to the drive controller.

This frees the drive maker from the restrictions imposed by CHS addressing, and makes it far simpler to introduce more sophisticated designs. For example, in existing drives the capacity is set by the inmost tracks, where the head is moving more slowly, so that the bits are packed more closely. LBA addressing will make it easier for the drive designers to increase the capacity by putting more sectors on the outer tracks where the head is moving faster.

This advance is being utilised in two ways.

The motherboards in the latest PCs feature a modified BIOS which traps Int 13 and converts the CHS address back to an LBA address, which it then passes on to the drive. This means these PCs can use the large drives, with existing versions of DOS, without encountering the problems already described.

Windows 95, and presumably equivalent versions of competing operating systems, will use LBA addressing throughout, and will be able to access the new drives directly.

Thus everyone in the industry is being forced to worry about a problem which will have been solved almost as soon as it has been introduced. Unfortunately the large number of these drives being fitted to existing PCs means that we will probably have to go on dealing with it for another ten years.

CONCLUSION

The computer industry generally, and the PC industry in particular, is notorious for its disregard for standards, and the number and inelegance of the 'kludges' it resorts to. These drivers are one more example of this; a kludge introduced to overcome a short-term problem which will cause many users to lose data, and raise problems for other sections of the industry for years to come.

Fortunately, knowledge, as usual, is power, and if you are aware of the problem, and use the right software, these drives should not cause you any serious problems.

REFERENCES

[1] 'IDE Takes Off', John Bryan, *Byte*, March 1994

[2] 'Breaking the 528 MB DOS Barrier', Quantum Corp, *PC Update*, April 1995.

[3] 'Breaking the 528 MB DOS Barrier; the Downside', R.H.Riordan, *PC Update*, June 1995.

SCANNERS OF THE YEAR 2000: HEURISTICS

Dmitry O. Gryaznov

S&S International Plc, Alton House, Gatehouse Way, Aylesbury, Bucks, HP19 3XU, UK
Tel +44 1296 318700 · Fax +44 1296 318777 · Email grdo@sands.co.uk

INTRODUCTION

At the beginning of 1994, the number of known MS-DOS viruses was estimated at around 3,000. One year later, in January 1995, the number of viruses was estimated at about 6,000. By the time this paper was written (July 1995), the number of known viruses exceeded 7,000. Several anti-virus experts expect this number to reach 10,000 by the end of the year 1995. This large number of viruses, which keeps growing fast, is known as the glut and it does cause problems to anti-virus software – especially to scanners.

Today, scanners are the most frequently used type of anti-virus software. The fast-growing number of viruses means that scanners should be updated frequently enough to cover new viruses. Also, as the number of viruses grows, so does the size of the scanner or its database, and in some implementations the scanning speed suffers.

It was always very tempting to find a final solution to the problem; to create a generic scanner which can detect new viruses automatically without the need to update its code and/or database. Unfortunately, as proven by Fred Cohen, the problem of distinguishing a virus from a non-virus program is algorithmically unsolvable as a general rule.

Nevertheless, some generic detection is still possible, based on analysing a program for features typical or not typical of viruses. The set of features, possibly together with a set of rules, is known as heuristics. Today, more and more anti-virus software developers are looking towards heuristical analysis as at least a partial solution to the glut problem.

Working at the Virus Lab, *S&S International Plc*, the author is also carrying out a research project on heuristic analysis. The article explains what heuristics are. Positive and negative heuristics are introduced and some practical heuristics are represented. Different approaches to a heuristical program analysis are discussed and the problem of false alarms is explained and discussed. Several well-known scanners employing heuristics are compared (without naming the scanners) both virus detection and false alarms rate.

1 WHY SCANNERS?

If you are following computer virus-related publications, such as the proceedings of anti-virus conferences, magazine reviews, anti-virus software manufacturers' press releases, you read and hear mainly 'scanners, scanners, scanners'. The average user might even get the impression that there is no anti-virus software other than scanners. This is not true. There are other methods of fighting computer viruses – but they are not as popular or as well known as scanners; and anti-virus packages based on non-scanner technology do not sell well. Sometimes people who are trying to promote non-scanner based anti-virus software even come to the conclusion that there must be some kind of an international plot of popular anti-virus scanner producers. Why is this? Let us briefly discuss existing types of anti-virus software. Those interested in more detailed discussion and comparison of different types of anti-virus software can find it in [*Bontchev1*], for example.

1.1 SCANNERS

So, what is a scanner? Simply put, a scanner is a program which searches files and disk sectors for byte sequences specific to this or that known virus. Those byte sequences are often called *virus signatures*. There are many different ways to implement a scanning technique; from the so-called 'dumb' or 'grunt' scanning of the whole file, to sophisticated virus-specific methods of deciding which particular part of the file should be compared to a virus signature. Nevertheless, one thing is common to all scanners: they detect only *known* viruses. That is, viruses which were disassembled or analysed and from which virus signatures unique to a specific virus were selected. In most cases, a scanner cannot detect a brand new virus until the virus is passed to the scanner developer, who then extracts an appropriate virus signature and updates the scanner. This all takes time – and new viruses appear virtually every day. This means that scanners have to be updated frequently to provide adequate anti-virus protection. A version of a scanner which was very good six months ago might be no good today if you have been hit by just one of the several thousand new viruses which have appeared since that version was released.

So, are there any other ways to detect viruses? Are there any other anti-virus programs which do not depend so heavily on certain virus signatures and thus might be able to detect even new viruses? The answer is yes, there are: *integrity checkers* and *behaviour blockers (monitors)*. These types of anti-virus software are almost as old as scanners, and have been known to specialists for ages. Why then are they not used as widely as scanners?

1.2 BEHAVIOUR BLOCKERS

A behaviour blocker (or a monitor) is a memory-resident (TSR) program which monitors system activity and looks for virus-like behaviour. In order to replicate, a virus needs to create a copy of itself. Most often, viruses modify existing executable files to achieve this. So, in most cases, behaviour blockers try to intercept system requests which lead to modifying executable files. When such a suspicious request is intercepted, a behaviour blocker, typically, alerts a user and, based on the user's decision, can prohibit such a request from being executed. This way, a behaviour blocker does not depend on detailed analysis of a particular virus. Unlike a scanner, a behaviour blocker does not need to know what a new virus looks like to catch it.

Unfortunately, it is not that easy to block all the virus activity. Some viruses use very effective and sophisticated techniques, such as tunnelling, to bypass behaviour blockers. Even worse, some legitimate programs use virus-like methods which could trigger a behaviour blocker. For example, an install or setup utility is often modifying executable files. So, when a behaviour blocker is triggered by such a utility, it's up to the user to decide whether it is a virus or not – and this is often a tough choice: you would not assume that all users are anti-virus experts, would you?

But even an ideal behaviour blocker (there is no such thing in our real world, mind you!), which never triggers on a legitimate program and never misses a real virus, still has a major flaw. To enable a behaviour blocker to detect a virus, the virus must be run on a computer. Not to mention the fact that virtually any user would reject the very idea of running a virus on his/her computer, by the time a behaviour blocker catches the virus attempting to modify executable files, the virus could have triggered and destroyed some of your valuable data files, for example.

1.3 INTEGRITY CHECKERS

An integrity checker is a program which should be run periodically (say, once a day) to detect all the changes made to your files and disks. This means that, when an integrity checker is first installed on your system, you need to run it to create a database of all the files on your system. During subsequent runs, the integrity checker compares files on your system to the data stored in the database, and detects any changes made to the files. Since all viruses modify either files or system areas of disks in order to replicate, a good integrity checker should be able to spot such changes and alert the user. Unlike a behaviour blocker, it is much more difficult for a virus to bypass an integrity checker, provided you run your integrity checker in a virus clean environment – e.g. having booted your PC from a known virus-free system diskette.

But again, as in the case of behaviour blockers, there are many possible situations when the user's expertise is necessary to decide whether changes detected are the result of virus activity. Again, if you run an install or setup utility, this normally results in modifications to your files which can trigger an integrity checker. That is, every time you install new software on your system, you have to tell your integrity checker to register these new files in its database.

Also, there is a special type of virus, aimed specifically at integrity checkers – so-called *slow infectors*. A slow infector only infects objects which are about to be modified anyway; e.g. as a new file being created by a compiler. An integrity checker will add this new file to its database to watch its further changes. But in the case of a slow infector, the file added to the database is infected already!

Even if integrity checkers were free of the above drawbacks, there still would be a major flaw. That is, an integrity checker can alert you only **after** a virus has run and modified your files. As in the example given while discussing behaviour blockers, this might be well too late...

1.4 THAT'S WHY SCANNERS!

So, the main drawbacks of both behaviour blockers and integrity checkers, which prevent them from being widely used by an average user, are:

1. Both behaviour blockers and integrity checkers, by their very nature, can detect a virus only **after** you have run an infected program on your computer, and the virus has started its replication routine. By this time it might be too late – many viruses can trigger and switch to destructive mode **before** they make any attempts to replicate. It's somewhat like deciding to find out whether these beautiful yet unknown berries are poisonous by eating them and watching the results. Gosh! You would be lucky to get away with just dyspepsia!
2. Often enough, the burden to decide whether it is a virus or not is transferred to the user. It's as if your doctor leaves **you** to decide whether your dyspepsia is simply because the berries were not ripe enough, or it is the first sign of deadly poisoning, and you'll be dead in few hours if you don't take an antidote immediately. Tough choice!

On the contrary, a scanner can and should be used to detect viruses **before** an infected program has a chance to be executed. That is, by scanning the incoming software prior to installing it on your system, a scanner tells you whether it is safe to proceed with the installation. Continuing our berries analogy, it's like having a portable automated poisonous plants detector, which quickly checks the berries against its database of known plants, and tells you whether or not it is safe to eat the berries.

But what if the berries are not in the database of your portable detector? What if it is a brand new species? What if a software package you are about to install is infected with a new, very dangerous virus unknown to your scanner? Relying on your scanner only, you might find yourself in big trouble. This is where behaviour blockers and integrity checkers might be helpful. It's still better to detect the virus while it's trying to infect your system, or even after it has infected but before it destroys your valuable data. So, the best anti-virus strategy would include all three types of anti-virus software:

- a scanner to ensure the new software is free of at least known viruses **before** you run the software
- a behaviour blocker to catch the virus **while** it is trying to infect your system
- an integrity checker to detect infected files **after** the virus has propagated to your system but not yet triggered.

As you can see, the scanners are the first and the most simply implemented line of anti-virus defence. Moreover, most people have scanners as **the only** line of defence.

2 WHY HEURISTICS?

2.1 GLUT PROBLEM

As mentioned above, the main drawback of scanners is that they can detect only **known** computer viruses. Six or seven years ago, this was not a big deal. New viruses appeared rarely. Anti-virus researchers were literally hunting for new viruses, spending weeks and months tracking down rumours and random reports about a new virus to include its detection in their scanners. It was probably during these times that a most nasty computer virus-related myth was born that anti-virus people develop viruses themselves to force users to buy their products and profit this way. Some people believe this myth even today. Whenever I hear it, I can't help laughing hysterically. Nowadays with two to three hundred new viruses arriving monthly, it would be total waste of time and money for anti-virus manufacturers to develop viruses. Why should they bother if new viruses arrive in dozens virtually daily, completely free of charge? There were about 3,000 known DOS viruses at the beginning of 1994. A year later, in January 1995, the number of viruses was estimated at least 5,000. Another six months later, in July 1995, the number exceeded 7,000. Many anti-virus experts expect the number of known DOS viruses to reach the 10,000 mark by the end of 1995. With this tremendous and still fast-growing number of viruses to fight, traditional virus signature scanning software is pushed to its limits [*Skulason, Bontchev2*]. While several years ago a scanner was often developed, updated and supported by a single person, today a team of a dozen skilled employers is only barely sufficient. With the increasing number of viruses, R&D and Quality Control time and resource requirements grow. Even monthly scanner updates are often late, by one month at least! Many formerly successful anti-virus vendors are giving up and leaving the anti-virus battleground and market. The fast-growing number of viruses heavily affects scanners themselves. They become bigger, and sometimes slower. Just few years ago a 360Kb floppy diskette would be enough to hold half a dozen popular scanners, leaving plenty of room for system files to make the diskette bootable. Today, an average good signature-based scanner alone would occupy at least a 720Kb floppy, leaving virtually no room for anything else.

So, are we losing the war? I would say: not yet – but if we get stuck with just virus signature scanning, we will lose it sooner or later. Having realised this some time ago, anti-virus researchers started to look for more generic scanning techniques, known as *heuristics*.

2.1 WHAT ARE HEURISTICS?

In the anti-virus area, heuristics are a set of rules which should be applied to a program to decide whether the program is likely to contain a virus or not. From the very beginning of the history of computer viruses different people started looking for an ultimate generic solution to the problem. Really, how does an anti-virus expert know that a program is a virus? It usually involves some kind of reverse engineering (most often disassembly) and reconstructing and understanding the virus' algorithm: what it does and how it does it. Having analysed hundreds and hundreds of computer viruses, it takes just few seconds for an experienced anti-virus researcher to recognise a virus, even it is a new one, and never seen before. It is almost a subconscious, automated process. Automated? Wait a minute! If it is an automated process, let's make a program to do it!

Unfortunately (or rather, fortunately) the analytic capabilities of the human brain are far beyond those of a computer. As was proven by Fred Cohen [*Cohen*], it is impossible to construct an algorithm (e.g. a program) to distinguish a virus from a non-virus with 100 per cent reliability. Fortunately, this does not rule out a possibility of 90 or even 99 per cent reliability. The remaining one per cent, we hope to be able to solve using our traditional virus signatures scanning technique. Anyway, it's worth trying.

2.2 SIMPLE HEURISTICS

So, how do they do it? How does an anti-virus expert recognise a virus? Let us consider the simplest case: a parasitic non-resident appending COM file infector. Something like Vienna, but even more primitive. Such a virus appends its code to the end of an infected program, stores a few (usually just three) first bytes of the victim file in the virus body and replaces those bytes with a code to pass control to the virus code. When the infected program is executed, the virus takes control. First, it restores the original victim's bytes in its memory image. It then starts looking for other COM files. When found, the file is opened in Read_and_Write mode; then the virus reads the first few bytes of the file and writes itself to the end of the file. So, a primitive set of heuristical rules for a virus of this kind would be:

1. The program immediately passes control close to the end of itself
2. It modifies some bytes at the beginning of its copy in memory
3. Then it starts looking for executable files on a disk
4. When found, a file is opened
5. Some data is read from the file
6. Some data is written to the end of the file.

Each of the above rules has a corresponding sequence in binary machine code or assembler language. In general, if you look at such a virus under DEBUG, the favourite tool of anti-virus researchers, it is usually represented in a code similar to this:

```
START:                              ; Start of the infected program
           JMP VIRUSCODE            ; Rule 1:  the control is passed
                                    ; to the virus body
                                    ;

           <victim's code>

VIRUS:                              ; Virus body starts here

SAVED:                              ; Saved original bytes of the
                                      victim's code

MASK:      DB '*.COM',0             ; Search mask

VIRUSCODE:                          ; Start of the virus code
           MOV DI,OFFSET START      ; Rule 2: the virus restores
           MOV SI,OFFSET SAVED      ; victim's code
           MOVSW                    ; in memory
           MOVSB                    ;

        MOV DX,OFFSET MASK   ; Rule 3: the virus
MOV AH,4EH              ; looks for other
INT 21H                 ; programs to infect

MOV AX,3D02H            ; Rule 4: the virus opens a file
INT 21H ;

MOV DX,OFFSET SAVED     ; Rule 5: first bytes of a file
MOV AH,3FH              ; are read to the virus
INT 21H                 ; body

MOV DX,OFFSET VIRUS     ; Rule 6: the virus writes itself
MOV AH,40H              ; to the file
INT 21H                 ;
```

Figure 1. A sample virus code

When an anti-virus expert sees such code, it is immediately obvious that this is a virus. So, our heuristical program should be able to disassemble a binary machine-language code in a similar manner to DEBUG, and to analyse it, looking for particular code patterns in a similar manner to an anti-virus expert. In the simplest cases, such as the one above, a set of simple wildcard signature string matching would do for the analysis. In this case, the analysis itself is simply checking whether the program in question satisfies rules 1 through 6; in other words, whether the program contains pieces of code corresponding to each of the rules.

In a more general case, there are many different ways to represent one and the same algorithm in machine code. Polymorphic viruses, for example, do this all the time. So, a heuristic scanner must use many clever methods, rather than simple pattern-matching techniques. Those methods may involve statistical code analysis, partial code interpretation, and even CPU emulation, especially to decrypt self-encrypted viruses: but you would be surprised to know how many real life viruses would be detected by the above six simple heuristics alone! Unfortunately, some non-virus programs would be 'detected' too.

2.3 FALSE ALARMS PROBLEM

Strictly speaking, heuristics do not detect viruses. As behaviour blockers, heuristics are looking for virus-like behaviour. Moreover, unlike the behaviour blockers, heuristics can detect not the behaviour itself, but just *potential ability* to perform this or that action. Indeed, the fact that a program contains a certain piece of code does not necessarily mean that this piece of code is ever executed. The problem of discovering whether this or that code in a program ever gets control is known in the theory of algorithms as the Halting Problem, and is in general unsolvable. This issue was the basis of Fred Cohen's proof of the impossibility of writing a perfect virus detector. For example, some scanners contain pieces of virus code as the signatures for which to scan. Those pieces might correspond to each and every one of the above six rules. But they are never executed – the scanner uses them just as its static data. Since, in general, there is no way for heuristics to decide whether these code pieces are ever executed or not, this can (and sometimes does) cause *false alarms*.

A false alarm is when an anti-virus product reports a virus in a program, which in fact does not contain any viruses at all. Different types of false alarms, as well as most widespread causes of false alarms, are described in [*Solomon*] for example. A false alarm might be even more costly than an actual virus infection. We all keep saying to users: 'The main thing to remember when you think you've got a virus – **do not panic**!' Unfortunately, this does not work well. The average user will panic. And the user panics even more if the anti-virus software is unsure itself whether it is a virus or not. In the case, say, where a scanner definitely detects a virus, the scanner is usually able to detect all infected programs, and to remove the virus. At this point, the panic is usually over; but if it is a false alarm, the scanner will not be able to remove the virus, and most likely will report something like: 'This file seems to have a virus', naming just a single file as infected. This is when the user really starts to panic. 'It must be a new virus!' – the user thinks. 'What do

I do?!' As a result, the user well might format his/her hard disk, causing himself a far worse disaster than a virus could. Formatting the hard disk is an unnecessary and un-justified act, by the way; even more so as there are many viruses which would survive this act, unlike legitimate software and data stored on the disk.

Another problem a false alarm can (and did) cause is negative impact on a software manufacturing company. If an anti-virus software falsely detects a virus in a new software package, the users will stop buying the package and the software developer will suffer not only profit losses, but also a loss of reputation. Even if it was later made known that it was a false alarm, too many people would think: 'There is no smoke without fire', and would treat the software with suspicion. This affects the anti-virus vendor as well. There has already been a case where an anti-virus vendor was sued by a software company whose anti-virus protection mistakenly reported a virus.

In a corporate environment, when a virus is reported by anti-virus software, whether it is a false alarm or not, the normal flow of operation is interrupted. It takes at best several hours to contact the anti-virus technical support and to ensure it was a false alarm before normal operation is resumed – and, as we all know, time is money. In the case of a big company, time is big money.

So, it is not at all surprising that, when asked what level of false alarms is acceptable (10 per cent? 1 per cent? 0.1 per cent?), corporate customers answer: 'Zero per cent! We do not want any false alarms!'

As previously explained, by its very nature heuristic analysis is more prone to false alarms than traditional scanning methods. Indeed, not only viruses but many scanners as well would satisfy the six rules we used as an example: a scanner does look for executable files, opens them, reads some data and even writes something back when removing a virus from a file. Can anything be done to avoid triggering a false positive on a scanner? Let's again turn to the experience of a human anti-virus expert. How does one know that this is a scanner, and not a virus? Well, this is more complicated than the above example of a primitive virus. Still, there are some general rules too. For example, if a program relies heavily on its parameters or involves an extensive dialogue with a user, it is highly unlikely that the program is a virus. This leads us to the idea of *negative heuristics*; that is, a set of rules which are true for a non-virus program. Then, while analysing a program, our heuristics should estimate the probability of the program to be a virus using both positive heuristics, such as the above six rules, and negative heuristics, typical for non-virus programs and rarely used by real viruses. If a program satisfies all our six positive rules, but also expects some command-line parameters and uses an extensive user dialogue as well, we would not call it a virus.

So far so good. Looks like we found a solution to the virus glut problem, right? Not really! Unfortunately, not all virus writers are stupid. Some are also well aware of heuristic analysis, and some of their viruses are written in a way which avoids the most obvious positive heuristics. On the other hand, these viruses include otherwise useless pieces of code, the only aim of which is to trigger the most obvious negative heuristics, so that such a virus does not draw the attention of a heuristical analyser.

2.4 VIRUS DETECTION VS. FALSE ALARMS TRADE-OFF

Each heuristic scanner developer sooner or later comes to the point when it is necessary to make a decision: 'Do I detect more viruses, or do I cause less false alarms?' The best way to decide would be to ask users what do they prefer. Unfortunately, the users' answer is: 'I want it all! 100 per cent detection rate and no false alarms!' As mentioned above, this cannot be achieved. So, a virus detection versus false alarms trade-off problem must be decided by the developer. It is very tempting to build the heuristic analyser to detect almost all viruses, despite false alarms. After all, reviewers and evaluators who publish their tests results in magazines read by thousands of users world-wide, are testing just the detection rate. It is much more difficult to run a good false alarms test: there are gigabytes and gigabytes of non-virus software in the world, far more than there are viruses; and it is more difficult to get hold of all this software and to keep it for your tests. 'Not enough disk space' is only one of the problems. So, let's forget false alarms and negative heuristics and call a virus each and every program which happens to satisfy just some of our positive heuristics. This way we shall score top most points in the reviews. But what about the users? They normally run scanners not on a virus collection but on a clean disks. Thus, they won't notice our almost perfect detection rate, but are very likely to notice our not-that-perfect false alarms rate. Tough choice. That's why some developers have at least two modes of operation for their heuristical scanners .The default is the so-called 'normal' or 'low sensitivity' mode, when both positive and negative heuristics are used and a program needs to trigger enough positive heuristics to be reported as a virus. In this mode, a scanner is less prone to false alarms, but its detection rate might be far below what is claimed in its documentation or advertisement. The often-used (in advertising) figures of 'more than 90 per cent' virus detection rate by heuristic analyser refer to the second mode of operation, which is often called 'high sensitivity' or 'paranoid' mode. It is really a paranoid mode: in this mode, negative heuristics are usually discarded, and the scanner reports as a possible virus any program which happens to trigger just one or two positive heuristics. In this mode, a scanner can indeed detect 90 per cent of viruses, but it also produces hundreds and hundreds of false alarms, making the 'paranoid' mode useless and even harmful for real-life everyday use, but still very helpful when it comes to a comparative virus detection test. Some scanners have a special command-line option to switch the paranoid mode on; some others switch to it automatically whenever they detect a virus in the normal low sensitivity mode. Although the latter approach seems to be a smart one, it takes just a single false alarm out of many thousands of programs on a network file server to produce an avalanche of false virus reports.

2.5 HOW IT ALL WORKS IN PRACTICE: DIFFERENT SCANNERS COMPARED

Being myself an anti-virus researcher and working for a leading anti-virus manufacturer, I have developed a heuristic analyser of my own. And of course, I could not resist comparing it to other existing heuristic scanners. We believe the results will be interesting to other people. They underscore what was said about both virus detection and false alarms rates. As the products tested are our competitors, we decided not to publish their names in the test results. So, only FindVirus of *Dr Solomon's AntiVirus Toolkit* is called by its real name. All the other scanners are referred to with letters: Scanner_A, Scanner_B, Scanner_C and Scanner_D. The latest versions of the scanners available at the time of the test were used. For FindVirus, it was version 7.50 – the first version to employ a heuristic analyser.

Each scanner tested was run in heuristics-only mode, with normal virus signature scanning disabled. This was achieved by either using a special command-line option, where available, or using a special empty virus signature database in other cases.

The test consisted of two parts: virus detection rate and false alarms rate. For the virus detection rate *S&S International Plc* ONE OF EACH virus collection was used, containing more than 7,000 samples of about 6,500 different known DOS viruses. For the false alarms test the shareware and freeware software collection of SIMTEL20 CD-ROM (fully unpacked), all utilities from different versions of MS-DOS, IBM DOS, PC-DOS and other known files were used (current basic *S&S* false alarms test set).

When measuring false alarms and virus detection rate, all files reported were counted; reported either as 'Infected' or 'Suspicious'. Separate figures for the two categories are given where applicable.

In both parts of the test, the products were run in two heuristic sensitivity modes, where applicable: normal or low sensitivity mode, and paranoid or high sensitivity mode. The automatic heuristic sensitivity adjustment was prohibited, where applicable.

The results of the tests are as follows:

Virus Detection Test

	Files scanned	Files triggered (infected + suspicious) Normal			Paranoid	
FindVirus	7375	5902	(N/A)	80.02%	N/A	
Scanner_D	7375	5743	(0+5743)	77.87%	6182 (0+6182)	83.54%
Scanner_C	7375	5692	(0+5692)	77.18%	N/A	
Scanner_A	7375	4250	(N/A)	57.63%	6491 (N/A)	87.74%
Scanner_B	7392(*)	3863	(2995+868)	52.38%	6124 (2992+3132)	82.68%

(*) Scanner_B was tested couple of days later, when 17 more infected files were added to the collection.

Table 1. Virus detection test results.

False alarms test

	Files scanned(*)	Files triggered (infected + suspicious) Normal			Paranoid	
FindVirus	13603	0	(N/A)	0.000%	N/A	
Scanner_A	13428	11	(N/A)	0.082%	371 (N/A)	2.746%
Scanner_B	13471	17	(0+17)	0.126%	382 (0+382)	2.836%
Scanner_D	13840	24	(0+24)	0.173%	254 (0+254)	1.824%
Scanner_C	13603	28	(0+28)	0.206%	N/A	

(*) Different number of files reported as scanned is due to the fact different products treat somewhat different sets of file extensions as executables.

Table 2. False alarms test results

3 WHY 'OF THE YEAR 2000'?

Well, first of all simply because I could not resist the temptation of splitting the name of the paper into three questions and using them as the titles of the main sections of his presentation. I thought it was funny. Maybe I have a weird sense of humour. Who knows...

On the other hand, the year 2000 is very attractive by itself. Most people consider it a distinctive milestone in all aspects of human civilisation. This usually happens to the years ending with double zero; still more to the end of a millennium, with its triple zero at the end. The anti-virus arena is not an exclusion. For example, during the EICAR'94 conference there were two panel sessions discussing 'Viruses of the year 2000' and 'Scanners of the year 2000' respectively. The general conclusion made by a panel of well-known anti-virus researchers was that, at the current pace of new virus creation by the year 2000, we well might face dozens (if not hundreds of thousands) of known DOS viruses. As I tried to explain in the second section of this paper (and other authors explained elsewhere [*Skulason, Bontchev2*]), this might be far too much for a current standard scanners' technique, based on known virus signature scanning. More generic anti-virus tools, such as behaviour blockers and integrity checkers, while being less vulnerable to the growing number of viruses and the rate at which the new viruses appear, can detect a virus only **when** it is already running on a computer or even only **after** the virus has run and infected other programs. In many cases, the risk of allowing a virus to run on your computer is just not affordable. Using a heuristic scanner, on the other hand, allows detection of most of new viruses with a regular scanner safe manner: **before** an infected program is copied to your system and executed. And very much like behaviour blockers and integrity checkers, a heuristic scanner is much more generic than a signature scanner, requires much rare updates, and provides an instant response to a new virus. Those 15-20 per cent of viruses which a heuristic scanner cannot detect could be dealt with using current well-developed signature scanning techniques. This will effectively decrease the virus glut problem five fold, at least.

Yet another reason for choosing the year 2000 and not, say, 2005 is that I have strong doubts whether the current computer virus situation will survive the year 2000 by more than a couple of years. With new operating systems and environments appearing (Windows NT, Windows'95, etc.) I believe DOS is doomed. So are DOS viruses. So is the modern anti-virus industry. This does not mean viruses are not possible for new operating systems and platforms – they are possible in virtually any operating environment. We are aware of viruses for Windows, OS/2, Apple DOS and even UNIX. But to create viruses for these operating systems, as well as for Windows NT and Windows'95, it requires much more skill, knowledge, effort and time than for the virus-friendly DOS. Moreover, it will be much more difficult for a virus to replicate under these operating systems. They are far more secure than DOS, if it is possible to talk about DOS security at all. Thus, there will be far fewer virus writers and they will be capable of writing far fewer viruses. The viruses will not propagate fast and far enough to represent a major problem. Subsequently, there will be no virus glut problem. Regrettably, there will be a much smaller anti-virus market, and most of today's anti-virus experts will have to find another occupation...

But until then, DOS lives, and anti-virus developers still have a lot of work to do!

REFERENCES

[*Bontchev1*]	Vesselin Bontchev, 'Possible Virus Attacks Against Integrity Programs And How To Prevent Them', *Proc. 2nd Int. Virus Bulletin Conf.*, September 1992, pp. 131-141.
[*Skulason*]	Fridrik Skulason, 'The Virus Glut. The Impact of the Virus Flood', *Proc. 4th EICAR Conf.*, November 1994, pp. 143-147.
[*Bontchev2*]	Vesselin Bontchev, 'Future Trends in Virus Writing', *Proc. 4th Int. Virus Bulletin Conf.*, September 1994, pp. 65-81.
[*Cohen*]	Fred Cohen, 'Computer Viruses – Theory and Experiments', Computer Security: A Global Challenge, Elsevier Science Publishers B. V. (North Holland), 1984, pp. 143-158.
[*Solomon*]	Alan Solomon, 'False Alarms', *Virus News International*, February 1993, pp. 50-52.

COMPUTER VIRUSES AND ARTIFICIAL INTELLIGENCE

David J Stang

Norman Data Defense Systems Inc, 3028 Javier Road, Suite 201, Fairfax, VA 22031, USA
Tel +1 703 573 8802 · Fax +1 703 573 3919

INTRODUCTION

The world of computing has talked much about 'artificial intelligence', but unfortunately the last decade has not seen much intelligence in software. The task of defending systems against computer viruses is one in which artificial intelligence could certainly be applied, with potentially valuable results.

The purpose of this paper is to show how traditional anti-virus practices (namely, scanning) cannot keep up in today's more sophisticated computer virus era. Instead, one must look towards advanced techniques for generic prevention, detection, and removal of viruses.

BACKGROUND: THE PROBLEMS OF DETECTION

Computer viruses have been around since 1986. Since then, four distinct phases have emerged, each with a different impact on anti-virus scanners:

PHASE 1: SIMPLE, STATIC VIRUSES

In 1986, there were 8 viruses, all written in what we will term 'the traditional approach'. That is to say, each virus was written with static code, resulting in every copy of the virus looking the same. The traditional approach to detecting these viruses was to search for static code with a scanner and alarm when a match was found. The key to scanning is to have a 'scan string' which identifies each virus. These scan strings can only be extracted if the anti-virus vendor has a sample of the virus; and the goal of scanning is to detect a virus which has already infected a file or a boot sector.

PHASE 2: ENCRYPTION

In 1987, virus authors began writing encrypted viruses, such as Cascade, in an attempt to defeat scanners. In Cascade, all but the first 35 bytes of the virus are encrypted, with each copy being different. This difference was accomplished by an encryption algorithm which used the original file size as the key. The actual number of different copies (except for the first few bytes) is the number of different file sizes available. Scanner developers solved such problems by scanning for those 35 static, stable bytes. This feat was manageable, and soon all scanners detected all copies of Cascade.

PHASE 3: ENCRYPTION WITH VARIABLE JUNK IN DECRYPTION ALGORITHM

In the late 1980's, virus authors came up with another method of defeating scanners: by inserting variable bytes in various places, replacing the static code reminiscent of 1986. The best of such viruses placed these variable bytes in the decryption algorithm itself. Scanner developers solved this new problem this way: define scan strings which contain markers for variable bytes, develop scanning algorithms to find some bytes, skip a variable number of bytes, and search for a match on the next bytes. For instance, to detect the fully encrypted virus Sverdlov (also known as 'Hymn of the USSR', 'USSR.1962', etc, and written in September, 1991), a scanner could use a scan string such as 'FE EB 02 ?? ?? 83 EE 08'. Using this method, scanning was slowed, but the scanners could still 'win'.

PHASE 4: POLYMORPHICS ROAM THE EARTH

In 1990, the beginning of the fourth and current phase, virus authors began writing polymorphic viruses, in which the stable bytes – the decryption algorithm – became shorter and shorter, and in which a number of different encryption algorithms were nested. Some viruses, such as Whale (August, 1990), were also able to be encrypted in memory, meaning that a traditional scan of memory might not identify them any better than a traditional scan of files. Some scanner vendors responded with large additional chunks of code in their scanner engines – code which was able to decrypt a specific polymorphic virus. Other vendors responded initially with more brute force in the scan string approach. For example, to detect the 32 morphs of Whale, Norton Anti-Virus version 1.5 provided 32 scan strings.

This approach seemed to work for a time, but then polymorphic engines came on the scene, including the Cybertech Mutation Technology, Dark Avenger's Mutation Engine (MtE), Simulated Metamorphic Encryption Generator (SMEG), Trident Polymorphic Engine (TPE), Virus Creation Laboratory (VCL), and more. Such engines take a non-polymorphic virus as input and then output the virus with polymorphic qualities. The availability of such engines has obviously made the task of writing a polymorphic virus straightforward. With these engines, virus authors no longer need to write their own polymorphic code. This cuts down on their production cycle and, as a result, the number of polymorphics has doubled about every 8 months, rather than the 8.5 month doubling time for viruses in general. Today, over 200 viruses are known to use a polymorphic engine, and another 50 polymorphic viruses exist which do not use any 'off-the-shelf' engine. Quite understandably, scanners have a difficult time detecting such polymorphic viruses.

It is difficult to estimate how many different polymorphic viruses exist:

1. Scanners have trouble detecting all copies of a single polymorphic virus, much less discriminating between two viruses which use the same engine. Thus, all viruses written with MtE are likely to be identified by a scanner as 'MtE'. If car buyers could only distinguish cars based on engine manufacturer, they would not think they had much to choose from.

2. No one seems to be able to agree on exactly what a polymorphic virus is. We could say that it was a virus which never had more than a string of *n* constant, consecutive bytes in common across copies of itself. But then we would need to define *n*. If *n* is less than a certain size, scanners will sometimes raise a false alarm. If the position of these bytes is variable, the false alarm rate goes up. If the bytes are bytes in common with many other uninfected programs (e.g., '(c) Microsoft') then false alarms are inevitable.

Some of these engines make the viruses highly polymorphic and difficult to detect using a traditional scanner. For instance, Girafe, which uses the Trident Polymorphic Engine (TPE), is able to create 16^{16} different copies of itself – about 18,446,774,000, 000,000,000 different morphs! Anti-virus vendor response was predictable: try to reverse-engineer a polymorphic virus, then try to detect its engine. However, vendor success has been limited. Some products, such as Central Point Anti-Virus seem unable to detect any polymorphic viruses whatsoever (see *Virus Bulletin*, July 1994). Few products are able to detect all copies of such polymorphic viruses as NATAS or Satan Bug, and no product is able to detect all copies of Commander Bomber.

Another characteristic of the fourth phase – the large numbers of new viruses being produced – augments the strain on 'scanners' due to the sheer number of scan strings which must be produced. In the past 5 years, the total number of viruses has doubled about every 8.5 months. As of January 1, 1995, there may have been 7,198 or so different viruses[1]. If this number is correct, and the doubling rate is 8.5 months, then at the time of writing this, there would be about 16,000 different viruses. Surely this number exceeds the discrimination power of scanners.

There are many reasons for this glut of new viruses.

There is ample help for doing so in published books, journals (such as *Computer Virus Developments Quarterly*, which published source code for a 'Windows 95' virus two months before Windows 95 shipped), virus authoring software, and heavily commented source code. Virus Exchange BBSs (VXBBSs) are electronic bulletin boards, to which hackers connect in order to communicate electronically and exchange files and viruses. VXBBSs typically stock the source code for 1,000 or more viruses, along with samples of viruses which a budding author can disassemble and study. Virus authors believe in sharing their virus-writing skills, and therefore provide their source code for others to revise and/or adapt.

In contrast, not everyone can write an anti-virus product. The effort required to write a product which stops all viruses is far greater than the effort required to write a single virus which gets past some anti-virus products. There is no published help for doing so, either. Vendors strive to be profitable, so don't share source code for their products.

With perhaps 500 active virus authors in the world, and only about 50 active anti-virus vendors, the stage is set for overtaking products with problems. Over the past 4 years, viruses have emerged at a faster rate than scanner detections have improved. Where once a scanner could detect 100% of the world's viruses, today few scanners detect more than 80%, and some, such as MSAV/MWAV with DOS 6.22, may only detect about 25%. (MSAV/MWAV in DOS 6.22 is exactly the same program which shipped with DOS 6.00, and only detects 1,404 viruses – about 25% of those which some other products can detect). In relative terms, scanners have gotten worse.

BACKGROUND: THE 'PROBLEMS OF REMOVAL'

Users and organizations live by a single cardinal rule: any virus found must be removed. Although some believe that deleting the infected file and restoring the file from backups is sufficient and satisfactory, often no backups exist, and sometimes the backups are infected. Removing the virus from an infected file or boot sector is a worthy and reachable goal.

As with scanners, new virus technologies have been problematic for virus removal.

The traditional approach for dealing with removal was to 'hard code' the instructions for removal into the product: move the file pointer, copy bytes, then truncate the file. With hard-coded removal instructions, precision in both the identification and removal algorithms is critically important, for an error in either means that the anti-virus product simply damages the file or sector. Many users who have experienced such damage have stoically considered this to be one of the hazards of war; an unfortunate event.

The removal problem for vendors is complicated by the large number of viruses that are out there. Today, it is no longer adequate for a scanner to casually identify a virus based on a dozen bytes matched, then proceed to brutalize the file in which they were found, because the actual virus might be a derivative of the original which the scanner has identified – one which requires different removal steps. For instance, one single scan string can detect most members of the Jerusalem family, but various variants of Jerusalem add different numbers of bytes to the infected file.

[1] See Norman Technical Report #9, 'How Many Viruses Are There?'

Here are just a few examples:

Variant of Jerusalem	Bytes Added
Suriv 1	897
Slow	1701-1716
Jerusalem 2187	2187

Any removal instructions which treat each of these different samples in the same manner are likely to leave the user in an unfortunate condition.

Removal is further complicated when a virus is derived from two or more other viruses. Some scanners use short scan strings, with which they might detect the file or boot sector to be infected with one of the progenitors, not the derivative. Furthermore, if the scanner proceeds to attempt to remove the virus, the scanner would surely fail to remove the derivative virus and/or succeed in damaging the file.

Encryption makes for difficult removal as well. If a file virus encrypts just 1 byte of a file, then traditional removal requires that the scanner know how to decrypt that byte; for, without decryption, removal is not possible.

Anti-virus vendors have responded to the issue of removal with a variety of tactics. They have argued that only 100 or so viruses have ever been found in the wild, thus the ability to remove thousands of viruses is not important. Unfortunately, no one has done any real research on which viruses are found in the wild, and it is likely that there are thousands, rather than hundreds, which have surfaced in one or more offices in the past year.

Anti-virus vendors have also coped by suggesting that the virus at hand is a 'new variant'; that the user somehow has the misfortune of being one of the first to ever have this virus. Users are in no position to judge the truth of this statement, but the fact is that because viruses take a great deal of time to become widely distributed and common, most of these 'new variants' are actually 2 or 3 years old.

OTHER PROBLEMS WITH THE CURRENT ANTI-VIRUS PARADIGMS

We have noted that the current anti-virus paradigms have problems with both virus detection and removal, and that these problems are growing over time. There are other problems as well, leading us to the conclusion that a new paradigm is necessary.

1. **Scanning takes time**. It takes a finite amount of time to scan a machine for viruses – perhaps 5 minutes or more. If the country's 70 million employees who use PCs spend 5 minutes a day scanning, and earn $15 an hour, the annual cost of scanning (260 days a year) is $22,659,000,000. The 'costs of scanning' exceed the purchase price of anti-virus software after just a few weeks of scanning.
2. **Scanning is mis-timed**. If a machine is scanned every day at 9:00 AM, and infected one day at 9:05 AM, the virus has an entire day to spread throughout the machine and the organization. A virus can spread surprisingly far in this length of time. Even if the scanner is able to detect this virus, scanning presumes that the virus has already infected some disks or files. Traditional scanning is thus a means of detecting a problem which has already occurred, rather than a way of preventing a problem.
3. **Scanners need constant upgrading**. With 10 new viruses each day, full detection via a scanner requires daily upgrades to the product. But most organizations find upgrades a nightmare, and will not be able to upgrade this frequently. Organizations need something which does not require frequent upgrades.

4. **Scanners are slowing**. The more viruses a scanner must search for, the more places within a file it must search, and the more files it must search across, the slower the search must be. Since strings must be stored in memory, and memory is limited, we will soon see two-pass products, which load one set of strings, scan, then load a second set and scan. While vendors have used indexed searching techniques to speed their task, and computers are faster than they once were, their drives are also getting larger. Tomorrow's scanners will inevitably be slower than today's and yesterday's.

5. **Scanner maintenance is beyond the capability of vendors**. If there are only 10,000 viruses today, and the average virus requires only 1 hour to analyze, write detection instructions, and test those instructions, then creating just the detection instructions is a 10,000 hour job – about 4 person years. To compound the problem, polymorphic viruses can each require weeks of analysis by the world's most skilled programmers to determine how to detect it. It is no wonder that during the past year, many vendors have either failed or been absorbed by bigger companies. Today, the U.S. has only a few vendors with American-made products: *IBM*, *McAfee*, *Symantec/Norton*, *Norman Data Defense Systems* and *RG Software*. The programming strain of the traditional scanning paradigm has been one of the causes of this shrinking in the anti-virus vendor industry.

IN SEARCH OF A NEW PARADIGM

If the old scanning paradigm is falling farther behind in dealing with the virus problem, what must we look toward in a new paradigm? Here are some minimal requirements:

- The new paradigm must be able to prevent viruses from infecting boot areas and files, and prevent them from gaining control of the machine. The adage about an ounce of prevention is still true.
- The new paradigm must not require any user intervention. Background, transparent operation is critical for users who do not have the time to do a daily (or hourly!) scan.
- The new paradigm must not slow the user or the machine. Today's computers are used in business, where time is money.
- The new paradigm must be able to remove all viruses without necessarily resorting to prior knowledge of the specific nature of the virus. This is a requirement because it will be increasingly likely, in the future, that the virus in your machine is not a virus which your vendor has ever seen.
- The new paradigm must provide automatic recording of events and automatic notification of the organization's virus response team.

These requirements point to a proactive, preventive solution which includes artificial intelligence: behavior blocking (dynamic code analysis) and static code analysis.

BEHAVIOR BLOCKING (DYNAMIC CODE ANALYSIS)

If we cannot realistically expect to detect viruses with scan strings – because of their proliferation, because of the increase in polymorphics, etc – then we must find some other way of detecting them. Ideally, users should strive to buy a product which prevents viruses from infecting rather than merely detecting them after they have infected. This kind of protection is available with behavior blocking.

DEFINITION

Behavior blocking is defined as the process of dynamic code analysis. The sequence of actions of a program is monitored to determine if the actions are consistent with the behavior of viruses. Because a blocker cannot merely monitor action, but must prevent certain actions, the smart behavior blocker must employ some technique to prevent the actual results of a sequence of steps, (for instance, the behavior blocker could permit a program to execute in a 'virtual machine' until it had determined that the sequence of actions was legitimate or was virus-like; if legitimate, the actions could be echoed to the real machine. The virtual machine could include virtual CPUs, virtual drives, etc). The techniques used by one behavior blocker may differ from those used by another, but the underlying principle will be the same: a sequence of code execution will be monitored until it is determined that the sequence is safe or is harmful; if harmful, the code will not be permitted to execute and the user will be notified.

HISTORY OF THE IDEA

The idea of behavior blocking is not entirely new. Andy Hopkins was one of the first to offer a behavior blocker named Bombsqad, a TSR which would alert the user whenever a boot sector was written to, a file was written to, and so on. It simply watched the usage of DOS interrupts, and when a designated event took place, stopped the event and alerted the user, awaiting permission to continue. The problem with his approach was that it could not distinguish between a virus and a user. So the user would attempt to write to a file, and the warning would be triggered. After a number of false alarms , the user would eventually abandon the behavior blocker.

Bombsqad is no longer used. Many detractors of the technique assumed that behavior blocking had to cause false alarms, and dismissed the approach. Today, behavior blocking is not widely accepted as an approach, in part because of the inadequacy of the early demonstrations of the technique.

A number of products come with components which could be considered to be behavior blockers. For example, products from *ESaSS* and *Prescription Software* include resident **file attribute monitors**. If a virus is to infect an executable file (in a 'parasitic' way[2]), it can do so only if the file attributes permit it. If a file is marked read-only, then the virus must change this attribute to read-write before infecting. Of course, viruses have been clearing file attributes, infecting, then restoring attributes since the Jerusalem virus (1987). A resident attribute monitor can intercept the process of clearing attributes, or the process of changing the read-only attribute to read-write, or the process of changing any attribute. It can be selective or clumsy, be programmed to permit exceptions (such as changes caused by ATTRIB) or not, and can even be self-learning (let the user come up with an attribute changer of his own, a self-learning resident attribute monitor can be told to ignore these kinds of changes, and won't bother the user again when this same program is used to change other attributes).

Another approach to behavior blocking can be found in *Norton's* resident scanner, which intercepts Ctrl-Alt-Del, and checks the floppy drives for boot viruses. If it finds the floppy infected, it does not permit the warm reboot, but rather warns the user that the disk is infected. This effectively prohibits a reboot from infecting a machine with a boot virus, 'blocking' the infection.

Smart behavior blocking has been in use worldwide for several years. Developed by RE Solutions in Malaysia, and marketed in Asia by *Extol* and elsewhere by *Norman Data Defense Systems*, a device driver is able to 'see' the difference between an uninfected TSR (such as IPX or NETX) loading into memory and an infected TSR loading into memory. It is able to see the difference between the behavior of a database program which writes bytes to a database, and a virus which writes bytes to an executable. This behavior blocking device driver, called NVC.SYS, is part of 'Armour', *Norman's* virus protection package.

HOW SMART BEHAVIOR BLOCKING WORKS

A *smart* behavior blocker is able to disentangle the complex behavior of a virus from the complex behavior of a user running complex software. The basic design of such an instrument requires that viruses be very well understood by the designer, and that detailed sequences of behavior, not simple coarse behaviors, be examined and analyzed. The designer of a smart behavior blocker must use statistical analysis to determine the probabilities that particular behavior sequences are those of a virus or of a user. A coarse behavior

[2] A virus that 'infects' by creating separate code that is called before the 'infected file (as with DIR-II or a companion virus) will not be thwarted by file attributes. Such viruses are rare.

blocker might simply stop writing to a COM file, which may be an entirely valid action (e.g. some MS-DOS patches write directly to executables). A smart behavior blocker might reason as follows:

Action	Comment
A process opens a file of type 'COM'	Nothing wrong so far.
The process reads to the end of the file and then adds to the end of it, increasing its size.	Still nothing wrong, but suspicious.
The process returns to the beginning of the file and patches the code to point to the segment which was appended to the file.	Definitely something wrong. Like virus activity, and must be stopped, reversed, and reported.

ADVANTAGES OF BEHAVIOR BLOCKING

Behavior blocking has its advantages:

- Because the behavior blocker prevents the virus from infecting, it eliminates the need for removal from all but the original infected file.
- A behavior blocker can have a long shelf-life. Advanced algorithms can be used so that upgrades need not be done with the frequency of scanner updates. Because the behavior blocker does not scan for specific viruses but instead looks at the program's behavior, it need not be upgraded each time a new virus is discovered.

BEHAVIOR BLOCKING VS. RESIDENT SCANNING

Behavior blockers are sometimes confused with resident scanners. In fact, they are completely different technologies. Resident scanners are scanners that stay in memory, and scan a file or boot area when triggered by some event. Therefore, the problems with resident scanners are the same as those we have already enumerated for scanners. In addition, resident scanners necessarily occupy large amounts of memory, something most users cannot spare these days[3]. To cut down on memory consumption, most resident scanners do not contain any code to detect polymorphic viruses. This is not surprising – the code required would exceed the amount of memory a user is willing to allocate to such a program, and the resident scanner would slow the machine far too much. But a smart behavior blocker can detect and stop all polymorphic viruses at the time they try to go resident or infect a file. After all, a resident polymorphic virus goes into memory exactly the way a non-polymorphic virus does; a polymorphic's infection process is exactly the same as a non-polymorphic's process.

Smart behavior blocking's effectiveness is shown by *Norman* tests, in which only 12 viruses of 1,000 new 1994 viruses (less than 1.2%) got past the **1993** version of the *Norman* device driver. In comparison, the most recent versions of traditional non-resident scanners missed more than 20% of these viruses.

FUNCTIONS OF A BEHAVIOR BLOCKER

What might a behavior blocker be asked to do? Thinking of 'behavior' alone, here is a short list:

- Prevent a virus from going resident by loading low and allocating memory.
- Prevent a virus from going resident by loading to the top of conventional memory and not allocating memory.

[3] It is not necessary for a resident scanner to occupy a lot of conventional or upper memory, however. Thunderbyte's TBSCANX and its companion TBDRIVER can be loaded so as to use 0Kb of conventional memory, and only 4Kb of upper memory. The remainder of memory consumption is effectively hidden from the user, in the form of extended or expanded memory.

- Prevent a virus (or any other code) from writing to the Master Boot Sector or boot sector.
- Prevent a virus from adding its code to programs, whether resident or direct action.
- Prevent a virus from tunnelling around anti-virus or other software in an effort to gain control of the machine without detection.

If we did not split hairs, and permitted a resident behavior blocker to do other forms of anti-virus work, and wished it to be generic (working with all viruses), then we could add to this list of functions:

- Determine if a boot virus is resident at the time the behavior blocker is loaded, and optionally disable the resident code.
- Determine if a floppy disk is infected with some boot virus, and if so, clean it.

We might also expect that the behavior blocker would signal users audibly and visually (whether in DOS or Windows), would recommend the proper course of action, would record the event in a log, and would transmit relevant information to a server if the user is on a network. It happens that all of these capabilities are provided by *Norman's* NVC.SYS.

DRAWBACKS OF BEHAVIOR BLOCKING

Even smart behavior blocking may never be always smart enough. Some programs, for instance, write to themselves in a virus-like way. Installation and upgrade routines may patch existing files in a virus-like way. WINCIM and other communication programs may create temporary marker files when a batch download is requested, then dribble bytes into these existing files during a download, in a virus-like way. Programmers compiling to a file which already exists may overwrite the existing file in a virus-like way. Access control products, and products which use stealth boot virus technology to redirect the view of the Master Boot Sector (such as *MicroHouse*' EZDrive and *OnTrack*'s Disk Manager), can look like they are stealth boot viruses in memory. Odd boot sectors on a floppy can be detected by generic examiners as boot viruses, and get an unneeded cleaning.

In the end, whether a behavior blocker works well for a user, without false alarms, depends on exactly what other software they are using. In the future, the very, very smart behavior blocker will probably be able to deal with exceptions, so that certain pre-defined or user-defined events do not trigger it.

STATIC CODE ANALYSIS FOR DETECTION

Behavior blocking requires behavior, or live action. A program which loads and executes is behaving. But a behavioral analysis is not the only 'generic' way to detect a virus.

CHECKSUMMING

One common method of detecting a virus in a file is to checksum the file and store the information. Later, when the file is checksummed again, the checksummer can compare the current result with the previously stored value and warn the user of any difference.

The checksum approach has so many problems with its common implementations that it has fallen out of favor with knowledgeable users.

Problems include:

- Many checksummers ignore boot sectors and Master Boot Sectors, yet perhaps 90% of office infections are of boot sectors, rather than files.

- Checksummers which do not scan memory might be run at the same time a virus which infects on open, close, or directory scan is resident in memory. In such a case, the checksummer itself will become infected, as well as the files which it checks.
- Checksummers cannot report on why a file has changed, and therefore triggers false alarms on self-modifying programs.
- Checksummers cannot name, remove, or tell us anything about viruses.

There is a pleasant variation of checksumming which we have found useful. Since the master boot record and the boot record are small, and are rarely changed 'legally', why not back them up, then compare the backup copy with the current version; if they have changed, tell the user, and offer to repair by replacement? *Norman* has used this approach in its BootGuard (BG.EXE) and rescue disk program (RC.EXE) with considerable success. RC.EXE can also determine that CMOS has changed in some way, and can restore it.

STATIC CODE ANALYSIS

A superior alternative to checksumming is that of static code analysis, both for detection and removal. Static code analysis is defined as the process during which the code is not loaded and executed but rather studied while it is 'dormant'. Examples of static code analysis include 'heuristic scanning' and code disassembly.

Static code analysis for detection of viruses generally involves looking at the 'sum of the parts'. If the tool finds a few very good reasons to think something contains a virus, or finds a number of fairly good reasons, it can conclude that the file is infected.

Products able to do heuristic analysis of static code (i.e. a file or sector which was stored on a drive) and conclude whether or not the code contained a virus have been around for years. Some implementations have been slow, some produce false alarms, but generally the approach has merit. Products of note which offer heuristic scanning include TBScan (from the *ESaSS*), F-Prot (from *Frisk Software*), NSCAN and ViewBoot (both from *Norman*).

In an appendix, we provide samples of static code analysis. For a file virus, we provide copies of the output from *ESaSS's* TBScan and Norman's NScan. (F-Prot will, if run with the switches /analyze /only, produce the line '[filename] seems to be infected with a virus'. We did not think this enough information to provide a page in the appendix. If F-Prot is told to do heuristics, but it recognizes the virus by name, it does not provide anything but the name of the virus found.) For a boot virus, we provide copies of the output from *Norman's* ViewBoot. (Neither F-Prot nor TBScan provide static code analysis of boot sectors.) This output includes both the ViewBoot report and the ViewBoot disassembly.

Static code analysis offers these potential benefits:

- Potentially it can provide accurate information on a virus sample. If a product identifies a virus as 'Jerusalem', static code analysis can, potentially, provide more useful information about that particular sample than V-Base or other pre-published reports, for the name 'Jerusalem' is not adequately precise to determine which strain it is, and thus, exactly what it does.
- It can, potentially, determine that a suspect sample is infected, or is not infected with a virus. This can be useful if the virus is new, and not detected and named by a standard scanner.

However, static code analysis can suffer from these drawbacks:

- Static code analyzers must be carefully tuned, or they will show very high false alarm rates. Such alarms ultimately lead to distrust of the static code analyzer, and cause all manner of mischief and mayhem until the alarm is determined to be a false alarm.

- Static code analyzers typically do not name the virus, when they find one. While the static code analyzer cannot be faulted for this, it leaves us with an incomplete understanding of what we have found.
- Static code analysis can be slower than traditional scanning. Since traditional scanning is already slower than users might like, static code analysis is likely to not be used quite as often as might be appropriate.
- Static code analysis, in itself, constitutes detection, but not removal. Unless the code analyzer contains algorithms for removal, it leaves the user in the precarious position of knowing they have a problem, but not knowing what to do about it. There are, however, generic approaches to removal – approaches which work. We discuss these approaches in the next section.
- A static code analyzer must know how to decrypt an encrypted virus, or it will (falsely) conclude that the file is not infected.

STATIC CODE ANALYSIS FOR REMOVAL

In the same way we use static code analysis for detection, we can use static code analysis for removal. The goal is generic removal – removing a virus without identifying it by name and without having prior knowledge about the particular virus. Many sceptics might say that generic removal is not possible, but some companies, such as *ESaSS* and *Norman*, have been doing this with great success for years.

NON-SPECIFIC CLEANING OF FILES

The basic concept is simple. Consider the lowly COM file. On the first pass, the first 3 bytes of the COM file can be captured and stored along with some appended code which indicates the file's original size, date, and time. When a virus infects such a file, it replaces the first 3 bytes, stores them, and typically appends to the bottom of this file. On the second pass with the generic remover, the remover can see that the code it has added is no longer at the bottom. If the virus has left the code intact, then removal involves simply replacing the first 3 bytes (now a jump to the virus) with the original 3 bytes and then truncating the file to the correct size, date, and time. With an EXE file, the process involves first safely storing the header of the file, along with the size, date, time, and program entry point in a separate file. Repair can be done by replacing the header; removing prepended code, as needed, returning the entry point to its original state; then truncating to the correct length.

What if a file becomes infected with a virus which encrypts 1 or more bytes of the file? The generic remover can discern that the result of its effort is not successful, because the stored checksum mismatches the checksum which would be produced on its straightforward repair. In such a case, the remover can build a 'box' in memory, and run the virus in the box, allowing it to single-step against a 'virtual CPU'. The virus begins by decrypting itself (and the program it has infected). With each step, the generic remover studies the results to determine if the checksum is now correct. If so, it copies the decrypted file from memory to the drive, and closes the box on the virus, shutting it off long before it does any damage.

In the case of Windows files, overwriting becomes a serious problem for traditional removers. Many DOS viruses which infect Windows files assume that the file is a DOS file, and overwrite the first few thousand bytes of Windows program code, adding themselves to what they see as a DOS program with a DOS EXE header. Because of this destruction, Windows EXE files do not seem to survive infection by DOS EXE infectors. However, with generic removal, the Windows program code can be stored safely away and restored when needed. This restoration is enough to remove the virus and return the Windows program to its original condition.

NON-SPECIFIC CLEANING OF THE BOOT AREA

There are other approaches to generic removal. Boot viruses provide the simplest example. Consider a non-stealth, non-encrypting boot virus (which is to say, most virus infections). The virus typically copies the original Master Boot Record or boot record to some other sector, places its code in the sector of the code it has just copied, and ends with a jump to the displaced code. Then, when the machine boots, it reads the infected code, then the displaced unmodified original code.

Generic removal is straightforward: simply determine that the sector is infected (for instance, count the number of occurrences of calls to Interrupt 13h. Healthy Master Boot Records and healthy boot records each contain 2 occurrences; infected sectors contain more or less). Now look in all the obvious places for such a virus to have moved the original code: the slack space (side 0, cylinder 0, sectors 2 through n), the bottom of the root directory, the final cylinder, and any clusters marked bad in the FAT. Once the healthy record is located, simply copy it back to where it belongs. This is the approach taken by *Norman's* NVCLEAN, and it works with almost all boot viruses. It fails in those rare cases of viruses that overwrite the sector, such as Da'Boys.

If the virus is a stealth, encrypting virus (such as Monkey), the process can be different: use a pair of independent mechanisms to determine whether Interrupt 13h is owned by hardware or software. If owned by software, take a 'photograph' of the master boot record and boot record as the resident virus would permit. Such a picture is perfect, showing the sector as it should look. Now disable the virus by borrowing Interrupt 13 from it, and take another look. Either the Master Boot Sector or boot sector will look different, and we are in a position to repair by reversing the virus' original displacement of the sector. Thus in the case of Monkey, we can now copy the code from side 0, cylinder 0, sector 3 (the original Master Boot Record, which we saw in unencrypted form when Monkey was active) to side 0, cylinder 0, sector 1, overwriting Monkey and 'cleaning' the machine. This is the approach taken by *Norman's* NOSTELTH.

Another approach to dealing with the stealth, encrypting virus is easier on the programmer; a bit harder on the user: boot dirty (from the infected hard disk) and take a picture of the Master Boot Sector and boot sector. Copy the picture to a floppy. Now reboot clean. When drive C yields an 'invalid drive specification', simply write the code photographed to the appropriate sectors. This can be done with *Norman's* BootGuard.

THE FUTURE OF ARTIFICIAL INTELLIGENCE IN THE WAR AGAINST VIRUSES

It is a mistake to assume that nothing has been done with artificial intelligence concepts in fighting viruses. In truth, the most popular anti-virus products seem to use none of these techniques in their own design. But good anti-virus products do use artificial intelligence. Each of the techniques described in this paper has successful implementation in commercial products. For instance, *Norman's* products, which incorporate all of the new paradigm techniques described here, boast about 2 million users around the world.

But there is much to be done. For instance, users still shop for traditional scanners, having learned this paradigm back in 1989, when the paradigm had merit and the implementations of today's new paradigm left much to be desired. User acceptance is required before the vendors of traditional anti-virus products begin to look into the new intelligent paradigm.

User acceptance will always be affected by the false alarm rate. If a product triggers false alarms too often, users will reject it. That is not to say that reviewers will find fault. Typically, a reviewer aims a scanner at a directory full of viruses and garbage. A traditional scanner, which is well-written, will separate the viruses from the garbage by following the flow of program execution before reaching a decision. If virus code is in an unreachable area, the program is damaged, not infected. This means that the very best scanners can achieve detection rates of only 60% in some tests, where the number of garbage files is high. But any product which triggers false alarms under such conditions is likely to win praise from the reviewer.

APPENDIX: ESASS'S TBSCAN ANALYSIS OF A JERUSALEM STRAIN

```
Thunderbyte virus detector v6.24 - (C) Copyright 1989-1994, ESaSS B.V.

TbScan report, 02-21-1995 12:55:00

Parameters: c:\oops\*.com heuristic lo



C        \OOPS\JERUSTD.COM infected by Jerusalem related virus
c        No checksum / recovery information (Anti-Vir.Dat) available.
F        Suspicious file access. Might be able to infect a file.
M        Memory resident code. The program might stay resident in memory.
U        Undocumented interrupt/DOS call. The program might be just tricky
         but can also be a virus using a non-standard way to detect itself.

Found 1 files in 1 directories, 1 files seem to be executable.
```

1 file is infected by one or more viruses

APPENDIX: NORMAN'S NSCAN ANALYSIS OF A JERUSALEM STRAIN

```
Scanning Results.
Report prepared by NSCAN on 02-21-1995 at 12:58:12.
Contact your help desk or Norman Data Defense Systems Inc. at 703-573-8802
with questions.


There is a 100% chance that C:\OOPS\JERUSTD.COM contains a virus.
  * appears to infect when files are loaded and executed.
  * opens files, moves file pointer, reads files, writes to files, closes
    files.
  * gets, sets attributes.         * gets, resets file date.
  * adds the text ->I2V21<- to the end of files it infects.
Use of Memory:
  * allocates memory.       * resizes memory.          * goes resident.
Interrupt Usage:
  * disables, enables interrupts.
  * uses interrupts: 21h, 08h, 24h.
Symptoms:
  * deletes files.
  * displays a message or graphic.
  * effects may be date-activated.
  * checks for date of 13, Friday, 1987.
Stealth Index: 6 (above average: resets attributes, sets file date, hides in
    memory, disables error handler.)
4E00B8003DCD21725A2EA370008BD8B80242B9FFFFBAFBFFCD2172EB0505002EA31100B90500BA6B



--------------------------------- SUMMARY ---------------------------------
Switches              : C:\OOPS ANALYZE
Total files scanned   :      3
Total bytes scanned   :         3,325
Total infected files  :      1
----------------------------------------------------------------------------
```

APPENDIX: NORMAN'S VIEWBOOT ANALYSIS OF A BOOT VIRUS

```
Analysis of Boot Record of A:

Prepared by ViewBoot, a product of Norman Data Defense Systems Inc.


Date: 02-21-1995

Time: 13:01:48


1.   Positive Identification

— >  This sector contains the Form.A.DS5 virus or variant! <—

— >  Norman checksum: 00081694

(The .DS means this virus is one identified by David Stang, but not yet
accurately named by other products. The name is provisional.)

2.   Analysis of code in the sector

   û Number of FATs is normal: 2

   — Contains no code to wait for keyboard input if disk is not bootable. May
     be encrypted.
   — This boot sector does not contain bootstrap code. This is abnormal. The
     virus may occupy two sectors, or may be encrypted.

   — Contains no code to display error messages in event of trouble. May be
     encrypted.

   — Contains code to write sectors. This is very virus-like! This code is
     found 3 times!

   — Contains 8 occurrences of calls to Int 13h. Healthy boot and master boot
     sectors contain 2 and only 2 occurrences of this interrupt. Boot viruses
     usually contain more, sometimes less.

   — Contains code to get the date from the real-time clock. This is like some
     viruses, and unlike healthy sectors.

   û Normal boot signature (Uª) found.

   —> Conclusion: This sector is infected with a virus. <—
```

APPENDIX: DISASSEMBLY BY NORMAN'S VIEWBOOT

```
Disassembly of Boot Record of A:
Performed on 02-21-1995 at 13:02:01
By ViewBoot, a program from Norman Data Defense Systems 703-573-8802
(c) 1994 Norman Data Defense Systems Inc.

Start Hex              Meaning               Comment
0     EB5390           jump                  Boot Sector Jump
                       This jumps to the bootstrap routine.
3     4D53444F53352E30
                       db                    OEM name (MSDOS5.0)
                       This is the manufacturer's version of MS-DOS
                       — Start of BIOS parameter block —
11    0002             dw                    bytes per sector
13    01               db                    sectors per cluster
14    0100             dw                    number of reserved sectors
                       Usually 1 (0100), unless the manufacturer has
                       reserved additional sectors.
16    02               db                    number of FATs
                       The number of File Allocation Tables following the
                       reserved sectors. If two or more, the spares can be
                       used for data recovery.
17    E000             dw                    number of root directory entries
                       the maximum number of entries in the root directory
                       0002 means 200 (common for large hard drives)
19    400B             dw                    total number of sectors
                       on the drive. If 0000, then this value is provided
                       just below, with huge sectors.
21    F0               db                    media descriptor byte
                       F0 means 1.44Mb, 2.88Mb, 1.2Mb, or other media.
22    0900             dw                    number of sectors per FAT
24    1200             dw                    sectors per track
26    0200             dw                    number of heads
                       a number like 0700 means 8 heads - reverse the
                       bytes to read 0007, then remember that the first
                       head is 0, so 0007 = 8 heads
28    00000000         dd                    number of hidden sectors
                       a number like 11000000 means 11 hidden sectors
                       - the least significant byte goes first.
```

```
32   00000000          dd                    huge sectors - numer of
                       sectors if total number of sectors (above) is 0
                       - End of BIOS parameter block -
                       Otherwise, the value here is 00
38   29                db                    extended boot signature (29h)
39   6828DF15          dd                    volume ID number
43   4E4F524E45545554202020
                       db 11 dup(?)          volume label (NORNNETUT )
                       A volume label of NO NAME is created if the drive
                       or disk is formatted without specifying a volume
                       label (with /V)
54   4641543132202020  db 8 dup (?)          (FAT12 ) file system type
                       FAT12 means a 12-bit FAT
62   FA                cli                   Disable interrupts
63   01                ??
64   FE                ??
65   D6                ??
66   8A                ??
68   F0                ??
69   87                ??
70   E900F0            jmp loc
73   053B00            add ax,3B00
76   01                ??
77   04                ??
78   3B                ??
80   01                ??
81   0100              add [bx+si],ax
83   80                ??
84   01                ??
85   FA                cli                   Disable interrupts
86   33C0              xor ax,ax             Zero AX register
                       AX has been set to 0.
88   8ED0              mov ss,ax
90   BCFE7B            mov sp,FE7Bh
93   FB                sti                   Enable interrupts
94   1E                push ds
95   56                push si
96   52                push dx
97   50                push ax
```

```
98   07               pop es
99   B8C007           mov ax,C007h         ax now has the value 0
102  8ED8             mov ds,ax
104  33F6             xor si,si            Zero SI register
                      SI has been set to 0
106  26               ??
107  83               ??
108  2E               ??
109  13               ??
110  04               ??
111  02               ??
112  26A11304         mov ax,es:data
116  B106             mov cl,06h
118  D3E0             shl ax,cl            Shift w/zeros fill
120  8EC0             mov es,ax
122  33FF             xor di,di            Zero DI register
                      DI has been set to 0.
124  B9FF00           mov cx,00FFh
127  FC               cld                  Clear direction
128  F3A5             rep movsw            Rep when cx >0 Mov [si] to
                                           es:[di]
130  06               push es
131  B89A00           mov ax,9A00h         ax now has the value 9
134  50               push ax
135  BBFE01           mov bx,FE01h
138  B80102           mov ax,0102h         ax now has the value 102
141  8B               ??
142  0E               push cs
143  4D               dec bp
145  8B               ??
146  16               push ss
147  4F               dec di
149  CD13             int 13h              interrupt 13h, function 02h
                      This function reads one or more sectors into memory.
                      al = #sectors to read, ch = cylinders to read,
                      dl = drive, dh = side to read.
151  72FE             jc loc               Jump if carry Set
153  CB               retf
154  0E               push cs
```

```
155  1F              pop ds
156  E82F00          call sub
159  E84100          call sub
162  BB4C00          mov bx,4C00h
165  BE4100          mov si,data
168  BF4603          mov di,data
171  E8C700          call sub
174  B404            mov ah,04h              AH has now been set to 04h
176  CD1A            int 1Ah                 real time clock
                     AH has been set to 04h. When this interrupt is called,
                     function 04h gets the date, with cx returning the
                     year, and dx returning the month/day.
178  80FA18          cmp dl,18h
181  750C            jne 0C
183  BB2400          mov bx,2400h
186  BE4500          mov si,data
189  BF5D03          mov di,data
192  E8B200          call sub
195  5A              pop dx
196  5E              pop si
197  1F              pop ds
198  33C0            xor ax,ax               Zero AX register
                     AX has been set to 0.
200  50              push ax
201  B8007C          mov ax,007Ch            ax now has the value 7
204  50              push ax
205  CB              retf
206  33C0            xor ax,ax               Zero AX register
                     AX has been set to 0.
208  8EC0            mov es,ax
210  BB007C          mov bx,007Ch
213  B80102          mov ax,0102h            ax now has the value 102
216  8B ??
217  0E              push cs
218  49              dec cx
220  8B              ??
221  16              push ss
222  4B              dec bx
224  CD13            int 13h                 interrupt 13h, function 02h
```

```
                        This function reads one or more sectors into memory.
                        al = #sectors to read, ch = cylinders to read,
                        dl = drive, dh = side to read.
226  C3                 retn
227  0E                 push cs
228  07                 pop es
229  B280               mov dl,80h
231  B408               mov ah,08h          AH has now been set to 08h
233  BBF903             mov bx,F903h
236  CD13               int 13h             interrupt 13h, function 02h
                        This function reads one or more sectors into memory.
                        al = #sectors to read, ch = cylinders to read,
                        dl = drive, dh = side to read.
238  7214               jc loc              Jump if carry Set
240  B280               mov dl,80h
242  890E4900           mov cx,ds:data
246  89164B00           mov data,dx
250  B80102             mov ax,0102h ax now has the value 102
253  B90100             mov cx,0001h
256  32F6               xor dh,dh           Zero DH register
                        DH has been set to 0.
258  CD13               int 13h             interrupt 13h, function 02h
                        This function reads one or more sectors into memory.
                        al = #sectors to read, ch = cylinders to read,
                        dl = drive, dh = side to read.
260  726E               jc loc              Jump if carry Set
262  81C3               BE01 cmp word ptr [di],BE01h
266  B104               mov cl,04h 268 80 ??
269  3F                 ??
270  80                 ??
271  7407               je loc07
273  83                 ??
274  C3                 retn
275  10                 ??
276  E2F6               loop local loop
278  EB5C               jmp short loc
280  8A                 ??
281  77                 ??
282  01                 ??
```

```
283  8B                 ??
284  4F                 dec di
285  02                 ??
286  890E5100           mov cx,ds:data
290  89165300           mov data,dx
294  B80102             mov ax,0102h          ax now has the value 102
297  BBF903             mov bx,F903h
300  CD13               int 13h               interrupt 13h, function 02h
                        This function reads one or more sectors into memory.
                        al = #sectors to read, ch = cylinders to read,
                        dl = drive, dh = side to read.
302  7244               jc loc                Jump if carry Set
304  81BF3F00           cmp word ptr [di],3F00h 308 01 ??
309  FE                 ??
310  743C               je loc3C
312  817F0B00           cmp word ptr [di],0B00h
316  02                 ??
317  7535               jne 35
319  B80103             mov ax,0103h          ax now has the value 103
322  8B                 ??
323  0E                 push cs
324  49 dec cx
326  8B                 ??
327  16                 push ss
328  4B                 dec bx
330  CD13               int 13h               interrupt 13h, function 02h
                        This function reads one or more sectors into memory.
                        al = #sectors to read, ch = cylinders to read,
                        dl = drive, dh = side to read.
332  7226               jc loc                Jump if carry Set
334  BBFE01             mov bx,FE01h
337  49                 dec cx
338  890E4D00           mov cx,ds:data
342  89164F00           mov data,dx
346  B80103             mov ax,0103h          ax now has the value 103
349  CD13               int 13h               interrupt 13h, function 02h
                        This function reads one or more sectors into memory.
                        al = #sectors to read, ch = cylinders to read,
                        dl = drive, dh = side to read.
```

```
351  7213           jc loc              Jump if carry Set
353  E82B00         call sub
356  BBF903         mov bx,F903h
359  B80103         mov ax,0103h        ax now has the value 103
362  8B             ??
363  16             push ss
364  53             push bx
366  8B             ??
367  0E             push cs
368  51             push cx
370  CD13           int 13h             interrupt 13, function 02h
                    This function reads one or more sectors into memory.
                    al = #sectors to read, ch = cylinders to read,
                    dl = drive, dh = side to read.
372  C3 retn
373  33C0           xor ax,ax           Zero AX register
                    AX has been set to 0.
375  8EC0           mov es,ax
377  26             ??
378  8B             ??
379  07             pop es
380  89             ??
381  04             ??
382  26             ??
383  8B4702         mov ax,[bi+02h]
386  89             ??
387  44             inc sp
388  02             ??
389  FA             cli                 Disable interrupts
390  26             ??
391  89             ??
392  3F             ??
393  26             ??
394  8C             ??
395  4F             dec di
396  02             ??
397  FB             sti                 Enable interrupts
398  C3             retn
399  BEF903         mov si,data
```

LATE SUBMISSION

The following paper is a late addition to the proceedings and, therefore, appears out of sequence.

THE EVOLUTION OF POLYMORPHIC VIRUSES

Fridrik Skulason

Frisk Software International, PO Box 7180, 127 Reykjavik, Iceland
Tel +354 5 617273 · Fax +354 5 617274 · Email frisk@complex.is

The most interesting recent development in the area of polymorphic viruses is how limited their development actually is. This does not mean that there are no new polymorphic viruses, far from it - new ones are appearing constantly, but there is nothing 'new' about them - they are just variations on old and well-known themes.

However, looking at the evolution of polymorphic viruses alone only shows one half of the picture - it is necessary to consider the development of polymorphic virus detection as well. More complex polymorphic viruses have driven the development of more advanced detection methods, which in turn have resulted in the development of new polymorphic techniques.

Before looking at those developments that can be seen, it is perhaps proper to consider some basic issues regarding polymorphic viruses, starting with the question of why they are written.

That question is easy to answer - they are written primarily for the purpose of defeating one particular class of anti-virus product - the scanners. Considering virus scanners are the most popular type of anti-virus program, it is not surprising that they are the subject of attacks.

At this point it is worth noting that polymorphic viruses pose no special problems to a different class of anti-virus product, namely integrity checkers. This does not mean that integrity checkers should be considered superior to scanners - after all there is another class of viruses, the 'slow' viruses, which are easily detected by scanners, but which are a real problem for integrity checkers.

Fortunately, polymorphic slow viruses are not common at the moment. As a side note 'slow polymorphic' viruses also exist, and should not be confused with 'polymorphic slow' viruses. This category will be described at the end of this paper, together with some other 'nasty' tricks.

Considering how virus scanners work, a virus author can in principle attack them in two different ways - either by infecting an object the scanner does not scan, or by making the detection of the virus so difficult that the scanner, or rather the producers of the scanner may not be able to cope with it.

Polymorphic viruses attempt to make detection difficult - either too time consuming to be feasible, or beyond the technical capabilities of the anti-virus authors.

The success of virus authors depends not only on their programming skills, but also on the detection techniques used. Before describing the current techniques, however, a brief classification of polymorphic viruses is in order.

Polymorphic viruses are currently divided into three groups:

1) **Encrypted, with variable decryptors.** This is the largest and currently the most important group. Several methods to implement the variability are discussed below, but most of them should be familiar to readers of this paper.

2) **'Block-swapping' viruses.** Only a handful of viruses currently belong to this group, but they demonstrate that a polymorphic virus does not have to be encrypted. These viruses are composed of multiple blocks of code, theoretically as small as two instructions, that can be swapped around in any order, making the use of normal search strings nearly impossible.

3) **Self-modifying viruses using instruction replacement techniques.** This is where the virus may modify itself by replacing one or more instructions in itself with one or more functionally equivalent instruction when it replicates. So far this category is only a theoretical possibility, as no viruses have yet been written that use this technique. It is possible that some such viruses will appear in the future, perhaps only to written to demonstrate that it can indeed be done.

Considering that the viruses that currently fall into the second group are easy to detect using ordinary search strings, and that the third group is non-existent, the only polymorphic viruses currently of interest are encrypted ones.

For that reason the term 'polymorphic viruses' should, in the rest of this paper, really be understood to mean only viruses of the first group, that is, encrypted with variable decryptors.

So, how are those viruses detected?

Basically the detection methods fall into two classes - those that detect and identify only the decryptor and those that look 'below' the decryptor, detecting the actual virus. This is not a strict 'either-or' classification - a scanner may analyse the decryption loop to determine that it might have been generated by a particular virus, before spending time decrypting the code.

DECRYPTION-LOOP DETECTORS

There are several different methods that have been used to detect and identify decryption loops - which used to be the standard way of detecting polymorphic viruses - but there are several significant problems with these methods. The most common methods are described later, but if they are only used as the first step, and the virus then properly decrypted some of the following problems disappear:

- **Virus-specific.** Basically, the detection of one polymorphic virus does not make it any easier to detect another.

- **More likely to cause false positives.** As we get more and more polymorphic viruses, capable of producing an ever-increasing variety of decryptors, the chances of generating a false positive increase, as some innocent code may happen to look just like a possible decryptor.

- **Identification is difficult.** Many polymorphic viruses will generate similar decryptors, and it is entirely possible that a scanner will mis-identify a decryptor generated by one polymorphic virus as having been produced by another, unrelated virus. Also, in the case of variants of the same polymorphic virus, it may be possible to determine the family, but not the variant.

- **No disinfection.** Virus disinfection requires the retrieval of a few critical bytes from the original host file that are stored usually within the encrypted part of polymorphic viruses. This means that virus-specific disinfection is generally not possible, at it would require decrypting the virus.

On the positive side, detection of a particular decryptor may be quite easy to add, although that depends on the design of the scanner and the complexity of the virus. The decryption techniques are old, and several anti-virus producers have abandoned them, in favour of more advanced methods.

The most common detection methods in this group are:

- Search strings containing simple wildcards
- Search strings containing variable-length wildcards
- Multiple search strings
- Instruction usage recognition
- Statistical analysis
- Various algorithmic detection methods

SEARCH STRINGS CONTAINING SIMPLE WILDCARDS

The limitations of this method are obvious, as it can only handle a few 'not very polymorphic' viruses, which are sometimes called 'oligomorphic'. They may for example make use of a simple decryption loop, with a single variable instruction. The least variable polymorphic virus uses two different instructions, NEG and NOT, which differ by only one bit. Defeating this detection method is easy: just insert a random number of 'junk' instructions at variable places in the code. 'Junk' does not mean 'invalid', but rather any instruction that can be inserted in the decryption loop without having any effect. Typical examples include NOP, JMP $+2, MOV AX, AX and other similar 'do nothing' instructions.

SEARCH STRINGS CONTAINING VARIABLE-LENGTH WILDCARDS

This method takes care of decryptors that contain those junk instructions. However, there are two problems with this approach. Some scanners cannot use this method as their design does not allow variable-length wildcards, but that really does not matter, as the technique is very easy to defeat: just make the decryptor slightly more variable so that no single search string, even using a variable-length wildcard will match all instances of the decryptor. This can be done in several ways.

- ➡ Changing register usage: For example the DI register might be used for indexing, instead of SI, or the decryption key might be stored in BX instead of AX.
- ➡ Changing the order of instructions: If the order of instructions does not matter, they can be freely swapped around.
- ➡ Changing the encryption methods: Instead of using XOR, the virus author could just as well use ADD or SUB.

MULTIPLE SEARCH STRINGS

This is generally considered an obsolete technique, but many anti-virus producers used it back an 1990 when the Whale virus appeared. This virus could be reliably detected with a fairly large set of simple search strings. Today, however, most of them would probably use a different method. This detection method can easily be defeated by increasing the variability of the decryptor past the point where the number of search strings required becomes unreasonably large. There are other cases where the multiple search string technique has been used. One anti-virus company had access to the actual samples of a particular polymorphic virus that were to be used in a comparative product review. Rather than admitting that they were not able to detect the virus, they seem to have added a few search strings to detect those particular samples - and they did indeed score 100% in that test, although later examination revealed that they only detected 5% of the instances of the virus in question.

The reason 'X-raying' has mostly been abandoned is that it can easily be defeated, for example by using an operation the X-ray procedure may not be able to handle, by using three or more operations on each decrypted byte or by using multiple layers of encryption.

The last method to be developed does not suffer from that limitation, and can handle decryptors of almost any complexity. It basically involves using the decryptor of the virus to decrypt the virus body, either by emulating it, or by single-stepping through it in a controlled way so the virus does not gain control of the execution.

Unfortunately, there are several problems with this method:

- Which processor should be emulated? It is perfectly possible to write a virus that only works properly on one particular processor, such as a Cyrix 486 SLC, but the decryptor will just generate garbage if executed on any other processor. An intelligent emulator may be able to deal with this, but not the 'single-stepping' method.
- Single-stepping is dangerous - what if the virus author is able to exploit some obscure loophole, which allows the virus to gain control. In this case, just scanning an infected file would result in the virus activating, spreading and possibly causing damage, which is totally unacceptable. It should be noted that a very similar situation has actually happened once - however the details will not be discussed here.
- Emulation is slow - if the user has to wait a long time while the scanner emulates harmless programs, the scanner will probably be disabled, and obviously a scanner that is not used will not find any viruses.
- If the virus decryptor goes into an infinite loop and hangs when run, the generic decryptor might do so too. This should not happen, but one product has (or used to have) this problem.
- How does the generic decryptor determine when to stop decrypting code, and not waste unacceptable amount of time attempting to decrypt normal, innocent programs?
- What if the decryptor includes code intended to determine if it is being emulated or run normally, such as a polymorphic timing loop, and only encrypts itself if it is able to determine that it is running normally?
- What if the decryptor is damaged, so that the virus does not execute normally? A scanner that only attempted to detect the decryptor might be able to do so, but a more advanced scanner that attempts to exploit the decryptor will not find anything. This is for example the case with one of the SMEG viruses - it will occasionally generate corrupted samples. They will not spread further, but should a scanner be expected to find them or not?

Finally, it should be noted that there are other ways to make polymorphic viruses difficult than just attacking the various detection techniques as described above.

'Slow polymorphic' viruses are one such method. They are polymorphic, but all samples generated on the same machine at the same time will seem to have the same decryptor. This may mislead an anti-virus producer into attempting to detect the virus with a single search string, as if it was just a simple encrypted but not polymorphic virus.

However, virus samples generated on a different machine, or on a different day of the week, or even under a different phase of the moon will have different decryptors, revealing that the virus is indeed polymorphic.

Another recent phenomena has been the development of more 'normal-looking' polymorphic code. Placing a large number of 'do-nothing' instructions in the decryptor may be the easiest way to make the code look random, but it also makes it look really suspicious to an 'intelligent' scanner, and worthy of detailed study. If the code looks 'normal', for example by using harmless-looking 'get dos-version number' function calls, it becomes more difficult to find.

So, where does this leave us? Currently anti-virus producers are able to keep up with the virus developers, but unfortunately the best methods available have certain problems - the one most obvious to users is that scanners are becoming slower. There is no indication that this will get any better, but on the other hand there are no signs that virus authors will be able to come up with new polymorphic techniques which require the development of a new generation of detectors.