Advances In Multidisciplinary & Scientific Research
A Multidisciplinary & Interdisciplinary Journal

Vol. 4 No. 1, March 2018

Understanding and Managing the Dynamics of Computer Viruses

***Odule, Tola John**
Department of Mathematical Sciences
Olabisi Onabanjo University
P.M.B. 2002 Ago-Iwoye, Ogun State, Nigeria
E-mail: tola.odule@oouagoiwoye.edu.ng

Kaka, Olatubosun Abiodun
Department of Computer Science
Tai Solarin College of Education
Omu-Ijebu, P.M.B. 2018, Ijebu-Ode

ABSTRACT

An exposition of the nature, structure and mode of operation of computer viruses is hereby presented, with the assertion that computer viruses are best managed through a *proactive* measure using a *cryptographically strong* checksumming algorithm that generates a *fingerprint* for each executable stored on the system. Verification is then performed prior to running an executable file to ascertain that the system is in a 'safe state' since the last run. This ensures inoculation against all present and future viruses with absolute guarantee.

Keywords*:* Executable-file, cryptographic-checksum, interrupt-vector, encryption, signature.

Aims Research Journal Reference Format:

Odule, J.T & kaka, O.A. (2018): Understanding and Managing the Dynamics of Computer Viruses. Advances in Multidisciplinary & Scientific Research Journal. Vol.4. No.1, Pp 113-120. Article DOI: dx.doi.org/10.22624/AIMS/V4N1P15

1. INTRODUCTION

A virus is an infectious computer program that attaches itself to executable files, boot sector of a floppy disk, partition sector of a hard disk, batch files and device drivers to hide and propagate itself without immediately revealing its presence on the computer system. Viral programs have three main variants [1] namely: Trojan horses, worms and virus. The main difference among these variants is in their mode of operation.

A Trojan horse, in every sense, resembles and works like a normal computer program. Each time the program calls for keyboard input and a response is made, it may result in the reprogramming of the keyboard keys such as 'd' key expanding to 'del' thus deleting a specified medium. This type of viral program is normally propagated via device drivers like ANSI.SYS using the ANSI escape sequences or the dynamic link libraries using one of several sophisticated methods. Worms, unlike a virus replicate in their entirety, creating exact copies of themselves. Worms are usually found on computer networks and multi-user computers, and use inter-computer or inter-user communications as the transmission medium. Worms have the effect of slowing down the processing speed of the computer, which may be disruptive in a real-time process. Viruses and worms are best defined by replication, executable path, side effects and disguise [2].

2. CLASSIFICATION OF VIRUSES

Viruses can be classified into two categories according to the executable item they infect: *Bootstrap sector* virus and *parasitic* virus. *Bootstrap sector* virus [3] modifies the contents of either the disk bootstrap sector or the partition bootstrap sector, depending on the virus and type of disk, usually replacing the legitimate contents with its own version. The original version of the modified sector is normally stored somewhere else on the disk, so that on bootstrapping, the virus version will be executed first. This will then load the remainder of the virus code into memory, followed by the execution of the original version of the bootstrap sector. From then on, the virus generally remains memory-resident until the computer is switched off.

* Correspondence author

The mechanism of a *bootstrap sector* virus normally uses three distinct components:

- The bootstrap sector--replaced with a corrupted version, this is where the virus gains access.
- One previously unused sector—for storing the original bootstrap sector.
- A number of previously unused sectors—where the bulk of the virus code is stored.

Examples include *Brain* (floppy disk bootstrap sector), *Italian* (floppy disk and hard disk partition bootstrap sectors) and *New Zealand* (floppy disk and hard disk partition bootstrap sectors).

Parasitic Viruses [4] modify the contents of executable files and device drivers. They insert themselves at the end or at the beginning of the files, leaving the bulk of the program intact. The initial branching instruction in the program is modified, as obtained in *Vienna* and *Datacrime*, but program functionality is usually preserved. However, there are some viruses, which overwrites the first few hundred bytes of the program, making it unusable.

Although this approach may seem less infectious than one used by *memory-resident viruses*, [5] the infection of these viruses, in practice, is just as high, if not higher than that of *memory-resident viruses*. They are also more difficult to spot, since they do not change the interrupt table or the contents of available memory, and their infectious behaviour can be more unpredictable.

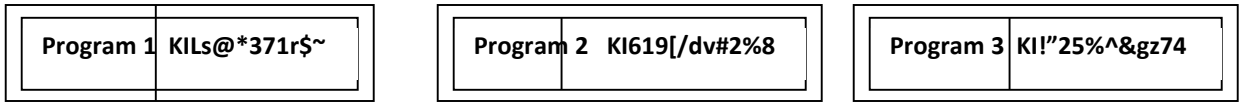
Hybrids: Some viruses use a combination of these two methods mentioned above. The *Typo* virus, [6] for instance, infects executables on invocation of an infected program, but also leaves a small *terminate-and-stay-resident* (TSR) element in memory after infection. The TSR section contains the payload, while the non-resident portion contains the replication code. In other hybrid viruses, these functions might be allocated differently.

3. MANIFESTATION OF VIRUSES

Viruses use hiding mechanisms, which allow them to replicate unnoticed, before delivering the 'payload'. Viruses employ two hiding mechanisms: *encryption* and *interrupt interception*.

Encryption

Some viruses use encryption of the virus code in order to make the majority of the virus code appear different in each infected application. This is designed to make the extraction of a fixed pattern-*signature*-more difficult, since the majority of the virus code changes on every infection [7] as shown in Figure 1.



(a) (b) (c)

Figure 1 Three viruses infected with an identical encrypted virus

Before the virus code can be executed, it must be decrypted in order to become meaningful sequence of instructions. The decryption routine must be in plain, unencrypted, form and it usually contains about ten to twenty bytes, which are identical and common to every infected executable shown as 'KI' in figure 1. The encryption key is usually linked, mathematically or otherwise, to the length of the executable.

The possibilities for introducing complications in encrypted viruses are practically endless. A two-stage encryption, where the key for encrypting the second stage is stored in an encrypted form in the first stage, may be used. Alternatively, cryptographically stronger algorithm may be used in place of the simple functions like exclusive-or (*XOR)* as in *Cascade* [8].

Interrupt Interception

Interrupt interception can be used very successfully to hide the presence of a virus in an infected PC. Most PC-based applications use software interrupts to communicate with the operating system in a portable way [9]. The branching addresses are stored in the interrupt table located at the beginning of memory, Figure 2, so that when an application issues interrupt, a jump occurs to a predetermined address.

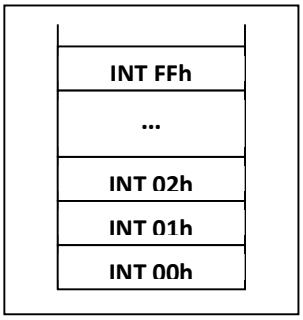
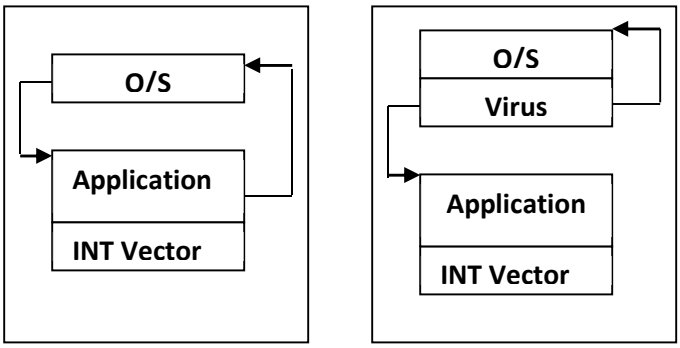


Figure 2: Interrupt Table

A virus may change one or more of these addresses so that any jump to the operating system is routed via the virus, which can then decide what to do with a particular request, as shown in Figure 3. The *Brain* virus [10] uses this technique.



(a) (b)

Figure 3: Interrupt routing before and after infection

Binary Viruses

Binary viruses are a special case of encrypted viruses. The principle is that a virus carries the replicating code in full, but only half of the payload. Only when the 'companion' virus is encountered, which carries the other half of the payload, the combination of the two payloads produces meaningful code, which can be executed as shown in Figure 4. This combination could be done by performing an exclusive-or (*XOR*) logic operation on the two halves. In a binary virus, the payload cannot be analysed unless one has access to the two halves of the virus.

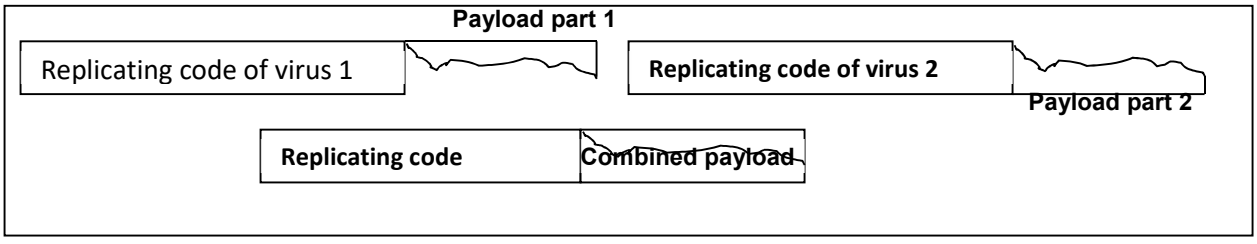


Figure 4: Binary virus—two parts combining to get a meaningful payload

Although this concept has been sometimes quoted as a dangerous development in virus sophistication, it has not been widespread. One of the very few viruses that seem to have incorporated this concept is the *dBASE* virus. As part of the payload, the original virus contains the sequence shown in Figure 5 on Intel machine and its classes [11]:

```
            CLI
            MOV         AX,3        ; Set count
LABEL:      MOV         CX,100h
            MOV         DX,0        ; Page 0 RAM
            MOV         DS,DX       ; Segment 0
            XOR         BX,BX       ; Offset 0
            PUSH        AX          ; Save the count
            INT         3h
            INT         3h
            POP         AX          ;Restore count
            INC         AX          ; Increase count by 1
            CMP         AL,1Ah      ; Count = 26?
            JL          LABEL       ;Repeat code if less
            ...
```

Figure 5 Specimen code sequence of a Binary virus

This sequence does not do much unless either of the following happens:

1. A companion virus changes the two INT 3h instructions into one INT 26h instruction.
2. A companion virus changes the interrupt table so that the INT 3h points to INT26h.

If either of the above happens, the payload becomes highly destructive. On triggering, the modified virus will overwrite the first 256 sectors of each drive from D: to Z: by using absolute disk write interrupt (INT 26h).

4. RECOGNITION OF VIRUS SIGNATURES

One common way of testing executable files for viruses is to search for recognition patterns known to be part of known viruses. These patterns are normally represented as hexadecimal digits and are referred to as hex patterns or *signatures* [12]. The hex patterns are normally from 10 to 16 bytes long, and there is a very small, but finite, chance that one of these patterns will be found in some uninfected and innocuous executable. Data in executable images are not completely random, and certain sequences of instructions used in a virus can occur in a perfectly legitimate program. The hex pattern from a virus is normally chosen so as to be unlikely to occur in a legitimate program, but there is a chance that this may happen. If a pattern checking program reports a pattern match, it does not necessarily mean that a virus *has* been found, but that a virus *may* have been found.

4.1 Dissection of a virus

Once a virus has been discovered, it is instructive to capture a *virus sample* for analysis, as this can be helpful to other sites infected with the same virus. Even if a virus is not completely analysed immediately, a *hex pattern* can be extracted, which helps to detect occurrences of the same virus elsewhere. The full analysis of a virus will invariably involve its full disassembly, i.e., reverse engineering its binary code into commented source code. Virus disassembly can be done with DEBUG, a tool supplied as part of MS-DOS.

4.2 Virus Disassembly

Disassembling a virus is an iterative process, which includes discovering first, which parts of the virus are data areas, which are not going to be disassembled, and which are instructions. Once this has been done, the output of DEBUG can be redirected to a file, which will contain the disassembled virus. Figure 6 is an example of the sequence of DEBUG commands contained in a file called INSTR used to analyse a hypothetical virus in the file VIR.COM.

```
U       100     102
D       103     10F
U       110     432
Q
```

Figure 6: Specimen sequence of DEBUG commands

DEBUG would then be invoked with the command: DEBUG VIR.COM <INSTR> VIR.ASM which instructs it to read input from the file INSTR and output to the file VIR.ASM, which will contain the disassembly of VIR.COM. Disassembly of *boot sector* viruses can be slightly more complicated, as they normally occupy more sectors than just the boot sector. One has to analyse the boot sector first in order to discover which other sectors the virus uses. The principle of redirecting DEBUG input and output can be used in the same way as for *parasitic* viruses.If a virus uses disk areas not accessible by DEBUG, for instance, hard disk bootstrap sector as programmed in *New Zealand*, the best approach is to write a small assembly language program, using DEBUG, to issue the appropriate BIOS interrupt(s) and read in the disk area in question. This can be written out to a file, using DEBUG, or analysed directly. The program below, written using DEBUG, starting at location 100h will read the hard disk boot sector into memory by using BIOS interrupt 13h service 02h. This service requires that ES:BX points to the memory location where the contents of the sector will be stored. In the program listing of Figure 7 on Intel machine or its classes, ES is set to the same value as DS and BX is set to 800h in the current data segment.

```
MOV     AX,DS
MOV ES,AX
MOV AX,0201     ; service 02h, 1 sector
MOV CX,0001     ; track 0, sector 1
MOV DX,0080     ; head 0, drive 0 (i.e., first hard disk)
MOV BX,0800
INT 13h         ; BIOS routine
JMP 10E         ; halt here
```

Figure 7: Sample program listing of a virus disassembly process

Running the program in Figure 7 by typing G 10E, placing the breakpoint at location 10E, location DS:0800 can now be either Dumped or Unassembled.Encrypted viruses present a slightly greater challenge, as they have to be decrypted before being disassembled. This is sometimes quite tricky, since the virus author may have used anti-DEBUG measures as found in *Cascade*.Once the disassembled virus has been written out to a file, analysis of the assembly code will reveal how the virus works, what it does and how it propagates.

One should normally have available good PC documentation, which includes lists of interrupts and then painstakingly work through the disassembly, documenting instructions, interrupts and memory locations. The picture will soon start to emerge. The replicating part of the virus will be isolated as well as its payload. Any payload trigger conditions should be analysed very carefully, as these are easy to misinterpret.

After this process, a hex pattern can be extracted, which can be used to search for the virus; 16 bytes are normally sufficient, provided the hex pattern is chosen carefully so that it represents a fairly unique set of instructions, unlikely to be found in other executables.

4.3 Virus mutations and pattern-checking programs

Pattern-checking programs rely on searching for a pattern known to exist within a virus. A version of a virus may be maliciously programmed such that it would not be recognised by a pattern-checker. The order of instructions, which are not dependent, could be changed or the same effect could be implemented using different instructions.

For example:

```
Mov ax,7f00h
Mov bx,0
```

Within a virus could be switched around to read:

```
Mov bx,0
Mov ax,7f00h
```

Any pattern-checker relying on the pattern produced by the first sequence of instructions, B800 7FBB 0000, would not recognise the mutated sequence BB00 00B8 007F. Sometimes the mutations of an existing virus will be so extensive that the virus will bear little resemblance to the original. Hex patterns extracted from the original are unlikely to be present in the new virus. The virus, *Fu Manchu* [13], is such an extensive mutation of *Jerusalem*, that it is classified as a new virus.

4.4 Identification of a Dissected Virus

In order to establish a positive identification of a *parasitic virus*, set up a dirty PC and make two copies of the same *experimental* executable. Infect one executable using the newly discovered virus and the other using the original virus. Run the DOS COMP command. If there are no differences, the virus is the same as the one already captured. If differences are discovered, the virus could still be the same, but it could be encrypted or self-modifying. This will have to be analysed more closely using the method outlined in section 4.2.1. *Boot sector viruses* are similarly identified.

4.5 Managing Virus

Virus countermeasures can be broadly classified into two groups: *proactive* and *reactive* measures. *Proactive* measures include the use of scanning and monitoring software—in the form of *terminate-and-stay-resident* (TSR) programs. The major drawback of these TSRs is that a well-written virus program can easily bypass or disable them. The mechanism used by TSRs [14] for intercepting disk reads and writes, i.e., to change the vectors in the DOS interrupt table, is exactly that used by most viral programs. In addition, monitoring activity degrades system performance and can be incompatible with network software, certain application programs etc.

Scanning software work on the principle of signature verification against known *database* collections. When a new virus appears, it is analysed and a characteristic pattern of 10 to 16 bytes recorded in the database. The *scanning* software will scan all executables on a disk, including the operating system and the bootstrap sector(s), and compare their contents with the known virus pattern. Of course, the demerits of such an approach is readily apparent: this type of software can only discover viruses that it knows about and, as such, the database has to be continually updated with new patterns as new viruses appear.

Another problem with the *scanning* software is the length of the virus pattern. If a short pattern is used, chances are that the scanning software will produce a number of false positives, finding the pattern in completely innocent software. If a long pattern is used, false positives will be reduced, while incidence of false negatives will be on the increase since any mutation of a virus will have a better chance of not matching the pattern. This is especially true of encrypted viruses. Most scanning software is slow, although it is possible to speed the search for viruses by not looking at all locations, but only at the locations known to be infected by a specific virus. Unfortunately, this complicates both the scanning software and the list of viruses, which become more difficult to keep up to date and absolutely correct, thus increasing the risk of false negatives. It is because of this that this approach may be found to be appropriate only in special circumstances. *Reactive* measures largely make use of checksumming algorithms.

The idea is that calculation of a checksum is performed on all executables on the system followed by periodic recalculations in order to verify that the checksum has not changed. If a virus attacks an executable, it will have to change at least one bit inside the executable, which will result in a completely different checksum. Even though this seems rather reactive, in that a virus attack will be detected after it happened, it could be made proactive with slight adjustment in the implementation.

The condition here is that prior to performing the initial checksum calculations, all executables are assumed to be 'clean' that is, virus-free. Installing all software from the manufacturers' original disks after performing a low-level format of the hard disk can do this. Then the checksumming algorithm is run on each of the executables to get a *fingerprint* or a *digital signature* of the executable. These *fingerprints* or *digital signatures* are stored in a database or individually stored in the executable file's header. Prior to running an executable on the system, a similar checksum is performed on the executable file and then compared with the initial checksum. A discrepancy in either of the checksums means that a virus has hit the executable.

The system may be programmed to take any action for instance, ringing the system bell for a specified length of time, displaying a prompt in a particular font and colour or both, in order to alert the user. The system may then be shut down, to clear the virus from memory, install a clean copy of the infected executable and another checksum of the executable calculated and stored. This saves the system from being infected with a virus. It is, however, assumed that the results of the checksumming algorithm must not be easily reproducible, lest a virus should do this on infection, preventing its detection, which necessitates the use of a *cryptographically strong* checksumming algorithm, eliminating the other two approaches namely: simple checksums and cyclic redundancy checks (CRCs). This assumption calls for the use of the so-called *one-way trapdoor function* [15] because of its mathematical properties.

The checksumming approach, as outlined here, is the only known method, which will detect all viruses, present and future, with absolute certainty. For a long-term anti-virus strategy, *cryptographically strong checksums* are the only approach, which should be used.

5. CONCLUSION

Proper documentation of all the known executable paths on a computer system formed an integral part of the methodology. The design principle focused on both virus-specific and non-specific algorithms. While the concept of virus-specific algorithms is appealing and somewhat straightforward in design, keeping the database of known virus patterns up-to-date made the approach non-beneficial in the long term. Software solutions could be obsolete almost as soon as they were procured because of the emergence of new viruses, or mutations of existing ones, on the scene. For a long-term foolproof protection against present and future viruses, a virus non-specific algorithm is favoured. Though the design and implementation were more rigorous and involved, the *cost-benefit* analysis more than justify these initial hurdles. From the practical point of view, the determining factor, in either case, is the prevailing exigency.

REFERENCES

1. Lance J. Hoffman, editor (1990). *Rogue Programs: Viruses ,Worms, and Trojan Horses*. VanNostrand Reinhold, New York, NY.
2. Leonard Adleman (1990). An abstract theory of computer viruses. In *Lecture Notes in Computer Science, vol 403*. Springer-Verlag.
3. Alan Solomon (1991). *PC VIRUSES Detection, Analysis and Cure*. Springer-Verlag, London.
4. David Ferbrache (1992). *A Pathology of Computer Viruses*. Springer-Verlag.
5. Eugene H. Spafford. An analysis of the internet worm. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, pages 446–468. Springer-Verlag, September 1989.
6. Peter J. Denning, editor (1990). *Computers Under Attack: Intruders, Worms and Viruses*. ACM Press (Addison-Wesley).
7. Donn Seeley (1990). Password cracking: A game of wits. *Communications of the ACM*, 32(6):700–703.
8. Yisrael Radai (1991). Checksumming techniques for anti-viral purposes. *1st Virus Bulletin Conference*, pages 39–68.
9. Brown, Ralf and Jim, Kyle (1991). *PC Interrupts: A Programmer's Reference to BIOS, DOS and Third Party Calls,* Addison-Wesley.
10. Jan Hruska (1990). *Computer Viruses and Anti-Virus Warfare*. Ellis Horwood, Chichester, England.
11. Eugene H. Spafford, et al (1989). *Computer Viruses: Dealing with Electronic Vandalism and Programmed Threats*. ADAPSO, Arlington, VA.
12. Eugene H. Spafford (1991). Computer viruses: A form of artificial life? In D. Farmer, C. Langton, S. Rasmussen, and C. Taylor, editors, *Artificial Life II, Studies in the Sciences of Complexity*, pages 727–747. Addison-Wesley, Redwood City, CA. Proceedings of the second conference on artificial life.
13. Sandeep Kumar and Eugene H. Spafford (1992). A generic virus scanner in C++. In *Proceedings of the 8th Computer Security Applications Conference*, pages 210–219, Los Alamitos CA. ACM and IEEE, IEEE Press.
14. Duncan Ray, Editor (1988). The MS-DOS Encyclopaedia. *Microsoft Press.*
15. M. Naor and M. Yung (1989). Universal One-Way Hash Functions and their Cryptographic Applications. Proceedings of the Twenty First Annual Symposium on the Theory of Computing, ACM.