A PLATFORM INDEPENDENT COMPUTER VIRUS

By

Keith Allen McMillan

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the degree of

Master of Science

in
Computer Science

at
The University of Wisconsin-Milwaukee
May 1994

A PLATFORM INDEPENDENT COMPUTER VIRUS

By

Keith Allen McMillan

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the degree of

Master of Science

in

Computer Science

at

The University of Wisconsin-Milwaukee

May 1994

Yvo Desmedt Date

Graduate School Approval Date

A PLATFORM INDEPENDENT COMPUTER VIRUS

By

Keith Allen McMillan

The University of Wisconsin-Milwaukee, 1994

Under the Supervision of Professor Yvo Desmedt

ABSTRACT

Some modern computer systems are subject to "infection" of their programs by self reproducing computer viruses. While it has been shown that detecting such a virus in general is an undecidable problem [Coh84], there may be large classes of viruses against which effective defenses can be made. Before an examination of the defenses is possible, a more complete catalog of the capabilities of viruses is necessary in order to determine if such classes exist.

Towards the end of a more complete picture of the capabilities of computer viruses, the author presents a virus written in the TEX document preparation language, with assistance from the GNU Emacs program. Such a virus is capable of running and spreading under a number of different operating systems without being recompiled or otherwise adapted.

Yvo Desmedt Date

Acknowledgments

The first person that I must thank is my wife, Kendra, for putting up with me for three years of school after work, and homework on the weekends. I could not and would not have started or finished without her.

I also need to thank Donald Arsenau of the Tri-University Meson Facility in California, Peter Schmitt of the Vienna University Computer Center, Bernd Raichle of the University of Stuttgart, and Hiroshi Nakashima of Kyoto University for answering what must have seemed like an obscure question in `comp.text.tex`.

Dr. Yvo Desmedt provided me with the original idea for the virus, some of the preliminary work, guidance along the road, and the occasional prod to complete this research.

Finally, I want to thank Andrew Kailhofer for proofing and editing this thesis.

Contents

List of Tables

Chapter 1

Introduction

I begin by defining a computer virus as referred to in the rest of this thesis. Additionally, I address some cousins of the virus and how they differ. I also briefly review the function of the TEX and LATEX systems, and the files that they use and create.

1.1 Computer Viruses

Some modern computer systems, most notably personal computers of the MS-DOS or Apple Macintosh varieties, are subject to infection of their executable programs by self-reproducing fragments of code called *viruses*. While determining whether a computer program contains a virus is undecidable [Coh84], there may be large classes of viruses against which effective defenses can be mounted. In order to determine if this is so, a more complete catalog of the capabilities of computer viruses is needed.

A computer virus is a fragment of a computer program. This fragment is embedded in another program, referred to as the host program, usually without the knowledge of the user of the host program. When the program is executed, the viral fragment briefly assumes control and carries out the actions it has been programmed to perform. At the minimum, this code fragment has to copy itself, or subvert another program interpreting the host file into copy it, from the host file to another file of the correct type without intervention from the user in order to qualify as a virus. If the virus is performing adequately, this will happen without the user of the host program even becoming aware that the viral fragment is present.

Generally, computer viruses have several purposes:

- Propagation by copying itself, or causing itself to be copied, into another file — When the viral fragment executes, it locates an uninfected file of the correct type and duplicates itself into the new host program. The virus does not need to make an exact copy of itself, and in fact some viruses make changes to the code that they insert into a new host program. Since a virus can alter the code of its offspring, it is in theory possible for a virus to *mutate* as it spreads, but usually these modifications are restricted to non-operational sections of the viral code, re-arrangement of order independent instructions, or encryption keys [Sol92].

Computer viruses usually determine if a potential host is already infected by the presence of a *signature*. This can be a string of bytes in a predetermined place in the file, an impossible date on the host file, or any other aspect of the file that the virus can alter, and that is not likely to have the same value as the signature if the virus is not present. Viruses can spread without checking for previous infection in a new target file, but this can lead to quick detection, since the file size then grows without bound as it is subsequently re-infected.

- Protecting itself from detection — Regardless of whether a computer virus spreads only when its host program is executing, or copies itself into the computer's core memory and continues to propagate long after its host program exits, the virus needs to execute at least once in order to spread. It can only execute as long as the users of the system do not detect its presence, employ countermeasures, and remove the virus.

  Computer viruses employ a number of different means to escape detection. Some viruses encrypt themselves with keys that change for each infection. Some personal computer viruses, once memory resident, can monitor system calls and "subtract" themselves from the returned data that would reveal their presence [Sol92]. Fortunately for the computer community at large, the vast majority do nothing to conceal their presence, and thus are easy to detect. The continuing escalation between virus writers and those designing countermeasures means that this situation is unlikely to continue.

- An optional action or *payload* — The payload is some sequence of actions which are not related to propagation or concealment. These actions, which may or may not be malicious in nature, are frequently triggered by a logic switch. Examples of such a payload could be the deleting of all files created before 1992 on the 13th of the month, or compressing executable files over a certain size and adding code to the beginning of the executable to automatically uncompress them when the program is run [Coh84]. The virus program presented in this thesis does not carry a payload.

This thesis does not review all the capabilities of computer viruses. For a review of the capabilities of personal computer viruses, consult [Sol92, Fer92, SHF90, SH90, Ste90].

1.2 Cousins of Viruses

Other types self-reproducing programs are occasionally grouped under the category of viruses. One of these is the *rabbit*, a self-contained program whose only function is to reproduce itself and use all available processing power and memory of a computer system, preventing legitimate users of the system from using these resources. This type of attack is referred to as *denial of service*. The principal difference between a rabbit and a virus is that the former is a self contained program, where the latter is contained within another file.

Another close cousin of the virus is the *worm*, a self-contained and self-replicating program which is designed to spread in a networked environment, making copies of itself on as many machines as possible. The term *tapeworm*, referring to an autonomous program that moves between computers on a network, first appeared in John Brunner's novel *The Shockwave Rider* [Bru75], and extensive research into worms was carried by Xerox PARC in the 1980's [SH82]. The Internet Worm is probably the most famous example of a working worm. It is interesting to consider the similarity between a worm infecting multiple computers and a virus infecting multiple programs in a computer, and in fact, some authors [Fer92] maintain that the difference is only semantics.

1.3 TeX and LaTeX

The TeX document preparation system was developed by Dr. Donald Knuth, principally for the American Mathematical Society, as a typesetting system [Knu84]. Its use has spread widely throughout academia, and into industry to some extent, since it

Extension	Program	Use
.tex	TEX, LATEX	Input to TEX, LATEX. This file contains the text of document and instructions describing its layout.
.log	TEX, LATEX	Diagnostic output from processing.
.dvi	TEX, LATEX	Output in *Device Independent* format.
.aux	LATEX	Auxillary file for storage of information between passes, such as cross-references, information for table of contents.
.lot	LATEX	Information for list of tables.
.lof	LATEX	Information for list of figures.
.idx	LATEX	Information for index.
.glo	LATEX	Information for glossary.
.toc	LATEX	Information for table of contents.

Table 1: TEX and LATEX File Extensions

allows authors of documents to typeset difficult equations and complicated text easily. LATEX [Lam86] is a set of configuration files for TEX, and is not here considered a separate environment, since it is usually packaged with TEX and will not function without it.

TEX and LATEX are available in source code format free of charge, and thus have been ported to Intel/MS-DOS machines, Digital Equipment Corporation VAX machines running both VAX/VMS and the Ultrix variant of Unix, and indeed nearly every Unix variant, regardless of hardware manufacturer.

TEX deals with a number of different types of files. Input to TEX consists of a stream of printable characters, usually stored as a file with an extension of `.tex`, which describe the layout and contents of the document. Additionally, TEX creates a log file, with an extension of `.log`, in which it records diagnostic output from the processing of the input file. The output from TEX is stored in a file with an extension of `.dvi` (for *device independent*). Most of the files that TEX and LATEX create or use take their names from the portion of the filename preceding the `.tex` in the source file, thus if the input were stored in a file `report.tex`, the output would be stored by default in `report.dvi`. Since LATEX is merely an extension of TEX it also uses all of the same files.

In addition to the files that TEX creates, LATEX creates a file with an extension of `.aux`, in which it records information of transient nature such as cross-references and index items. Other files may be created by LaTeX during the processing of some documents, including `.lof` files containing information used for generating lists of figures, and `.lot` files which contain information for lists of tables.

The most important aspect of TeX to review here is that as it reads characters from the input stream, it assigns them category codes, or *catcodes*, which determine how TeX treats the characters it reads. TeX allows us to change the catcodes it will assign a given character using simple commands.

Chapter 2

Goal

Most viruses are only capable of spreading in a single environment. In the past, the environments that most virus writers chose were Intel/MS-DOS or Apple Macintosh computers. The relatively large numbers of these machines meant that they were readily available to virus writers. Also, since object code varies between hardware platforms, and even between different operating systems on the same hardware platform, the abundance of these machines has made them natural targets.

2.1 Viruses and Multi-user Systems

In the past, viruses have not been a threat to multi-user systems. This was due in part to the number of MS-DOS and Macintosh machines as mentioned above, and also to the stricter resource controls that multi-user systems usually have. Restrictions on ability to access files or sections of core memory is usually tightly controlled in multi-user systems, in contrast to personal computers. It is the author's opinion that the lack of multi-user system infecting viruses is due more to the larger percentages of single user systems than to resource controls in multi-user systems. There have been a few examples of research on viruses in multi-user environments, among them [Coh84], which were also restricted to a single, albeit multi-user, platform.

2.2 Platform Independent Viruses

To date, the author is aware of only one platform independent virus, written in the UNIX Bourne shell [Duf89]. Whether other platform independent viruses could exist was largely an open question.

2.3 Our Virus

The goal of my research was to produce a virus that could run on a number of platforms, both single and multi-user, without needing to be recompiled. This thesis presents a virus that is capable of spreading in any environment which supports TeX (more specifically, LaTeX) and the GNU Emacs text editor.

2.3.1 Why a New Virus?

Why would we want to create a new virus? It has been shown that detecting if a computer program is a virus is an undecidable problem [Coh84]. This means that no computer system can decide deterministically in any amount of time whether an arbitrary computer program contains a virus [Coh84].

There may, however, be large classes of computer viruses against which an effective defense can be made. With our current knowledge of computer viruses, no such grouping of viruses is apparent, and so a larger survey of just what computer viruses can and cannot do is needed.

2.3.2 Why Use TeX?

The principal reasons that I employ TeX and LaTeX files as the host "programs" for a virus are:

- Homogeneity — TeX and LaTeX, while an older software system, has not suffered from the usual plague of conflicting versions that sometimes afflicts popular software packages. So, while the C language may vary considerably between a computer vendor's implementation and the ANSI standard, TeX files work with any port of TeX. TeX represents a sort of "virtual machine" on which we can build a virus that will work on a number of different physical machines.

- Portability — TeX and LaTeX files are standard text files, and can easily be electronically mailed or transferred by other digital means. Their use in academia means that these types of files frequently travel across the Internet. This provides the virus with an easy means of moving from one machine to another.

- Prevalence — TEX and LATEX are frequently used in academic and commercial environments. Additionally, TEX, LATEX and GNU Emacs run on a wide variety of computer platforms, as was previously mentioned.

Chapter 3

The LaTeX Virus

In the interest of not unleashing a new and possibly troublesome virus on the world, the actual code for the virus will not be presented as a part of this document. Instead, I present an analysis of each of the functions of the virus, with narrative description of their function. In this regard I am using the work of M. W. Eichin and J. A. Rochlis [RE89] as a guideline. In this section, the LaTeX file containing the virus is referred to as the *vector*, the file being read from is called the *source*, and the file being written to is called the *target*.

The LaTeX virus infects LaTeX input files. When the file is processed by LaTeX, the virus attempts to infect all files in the current directory with an extension of `.tex`. Since I was unable to find a way to access directory listings from within TeX the virus determines the names of these files by reading a listing created by Emacs. If it fails to find a listing file, but finds an Emacs initialization file, it will insert code to create such a listing into the initialization file. The next time Emacs enters TeX mode, it will create a listing for the virus. Emacs serves only to provide us with a listing of targets, while LaTeX, by processing the macros of the virus, actually does the work of infection.

3.1 Host Program Structure

The virus places itself immediately after the `\documentstyle` macro call in an infected LaTeX file. At this point, the virus is before the actual text of the document, which occurs later in the input file, after the `\begin{document}` macro. In this way, the virus minimizes the risk of conflicting with the text of the document, and alerting the user to its presence.

If the virus were to be released into the wild, an infected LaTeX file would contain the virus signature string, "`%DoNotInfectMe`", as the first characters on the line immediately following the closing "}" of the `\documentstyle` macro. This string would prevent a LaTeX file from being reinfected. In the interest of safety, the virus was actually coded to only infect files that *contain* the signature string. Regardless, the virus only infects LaTeX files (determined by the presence of a `\documentstyle` macro call), and leaves TeX files unchanged. This is also a safety consideration, as the virus could easily be modified to infect both TeX and LaTeX files, with only a slightly higher chance of being detected.

Immediately following the signature are approximately 200 lines of LaTeX code. This could easily be reduced, as the code is currently structured for readability, not compactness. One of the first actions the virus performs is making the @ character a valid letter by changing its catcode to that of a letter. As mentioned before, the catcode determines how TeX treats characters in the input stream. By changing the category code for @, LaTeX and the virus both exploit the fact that the user cannot easily define macros with the @ character in the name in order to avoid macro namespace collisions. For the virus, this reduces the chance of detection. The virus then defines stream numbers for reading and writing the TeX files, declares some conditional macros, and defines the macros that constitute the routines of the virus. These are the macros that are detailed in order of invocation below.

The virus then changes the catcodes of most of the special characters (e.g. "{", the beginning-of-group character, and "%", the comment character) to the catcode for regular characters. This prevents TeX from acting differently when it reads them from a TeX file as the virus copies it. The catcodes for some of the special characters, most notably "\", the escape character, cannot be changed here, as that would prevent the virus from being able to call its own macros. Almost all of the macros detailed below change these remaining catcodes as the first step upon invocation, and this change ceases to be effective when TeX finishes the execution of that macro.

Following this, the virus checks for the existence of the listing file, created by Emacs, by attempting to open it. This file contains the list of files in the current directory with a `.tex` extension. If it does not find this file, it checks for a Emacs startup file in the current directory. If it does not find a startup file, the virus does nothing. If the virus finds an Emacs startup file, but no list of `.tex` files, it adds

Emacs LISP code to dump a directory of files with a `.tex` extension, and attaches it to the `tex-mode-hook`, which is invoked whenever Emacs enters TeX mode. If the virus finds a list of TeX files, it reads the first line and discards it. This line contains the name of the directory that Emacs read to create the listing. The virus then enters the outer loop.

3.2 The Outer Loop

The outer loop of the virus iterates through the lines contained in the file created by Emacs. Providing Emacs located any files with a `.tex` extension, they are listed one or more to a line in this file. After it has read a line, it calls the `\@parseline` macro. It continues reading lines from the file and calling the `\@parseline` macro until it detects the end of the file, at which point it returns.

3.3 `parseline` macro

The `\@parseline` macro directs the infection of target files. It enters a loop and calls the `\@getnextfile` macro to get the next name from the current line. `\@getnextfile` stores the rest of the names on the line in the variable `\@rest`. The virus then displays the name of the file it is going to try to infect, as a safety precaution. It then opens vector, source and target files, and calls the `\@copytostyle` macro to begin the copy process. `\@parseline` calls other macros after `\@copytostyle` returns, the details of which are explained below.

3.4 `copytostyle` Macro

The `\@copytostyle` macro copies everything up to the `\documentstyle` call in the source file to the `.aux` file associated with the source file, which the virus uses as scratch space. `\@copytostyle` first changes the catcodes of the remaining special characters to that of a letter, sets the flag `@cont` to true and then enters a loop which reads the next line from the source file. It writes this line to the `.aux` file. This file will be truncated after the virus finishes its infection.

If \@copytostyle has not already read a line containing "\documentstyle," it calls \@FetchXIV to fetch the first 14 characters of the current line, which it compares to the string "\documentstyle." If the strings match, \@copytostyle sets the flag @FoundDocstyle to keep from checking again. Subsequent iterations through the loop will not check for "\documentstyle." Once \@copytostyle has located a \documentstyle call, and on subsequent iterations of the loop, it will call \@findbrace to determine if the current line contains the closing "}". If it does, \@findbrace will set the variable @cont to false. We continue reading lines and either checking for "\documentstyle" or "}" until we find them, or we encounter the end of the file.

If the \@copytostyle macro reaches the end of file marker without finding a \documentstyle call, it assumes the file is not a LaTeX file, sets the global variable @IsLaTex to false and returns control to the \@parseline macro. If the file is a LaTeX file (as signaled by @IsLatex being true), \@parseline next calls \@checkforsig to determine if the file contains the signature string.

3.5 checkforsig Macro

The \@checkforsig macro determines if the source file contains the virus signature string, "%DoNotInfectMe." It first changes the catcodes of the remaining special characters to that of a letter, and reads the next line from the source file (which should be the one immediately following the close of the \documentstyle call). It stores the tokens it reads in a global macro for later writing to the target file. This assures that the virus, and in particular the signature string that is the first thing in the viral code that is inserted, is the *first* line in the infected file after the closing brace of the \documentstyle call. It then calls the same \@FetchXIV macro that is called by \@copytostyle using the current line as an argument, but compares the result against "%DoNotInfectMe". Under normal circumstances, the virus would check for the absence of this signature, and infect the file if it was not present. As mentioned before, the research version of the file will only infect a file that *contains* the signature. If the virus finds the signature string, it sets the flag @Sig to true. If @Sig is true after \@checkforsig returns, as would be the case if there were a signature in the file, \@parseline next calls \@findstyle.

3.6 findstyle Macro

The `\@findstyle` macro is essentially a seek routine to find the point in the vector program immediately after the closing "}" for the `\documentstyle` call. Since this is very similar to the function of `\@copytostyle`, `\@findstyle` is essentially a simpler version of that macro, differing in that it does not write the lines it reads to the `.aux` file. Also since the virus only infects LaTeX files, it makes no provision for not finding the `\documentstyle` call. On returning from `\@findstyle`, `\@parseline` calls `\@copytoend`.

3.7 copytoend Macro

The `\@copytoend` macro copies the virus code from the vector file to the `.aux` file, appending it to the lines read from the source file. It first changes the catcodes of the remaining special characters to that of a regular character. It then enters a loop in which it reads lines from the vector and writes them to the `.aux` file. `\@copytoend` then calls `\@FetchXIV` to return the first 14 characters of the line it just read, and compares them to "`\def\@EndVirus`," which marks the end of the viral code. It continues copying lines until the end of virus marker is written. At this point, the `.aux` file contains the source file, up to the end of the `\documentstyle` call, followed by the viral code. `\@copytoend` then returns.

3.8 Wrapping Up

On the return from `\@copytoend`, `\@parseline` calls the `\@writepostsig` macro to write the line `\@findstyle` stored in the global macro out to the `.aux` file. Again, this insures that the virus begins on the *first* line in the target file after the closing "}" of the `\documentstyle` macro. It then calls the `\@copyrestof` macro to copy the rest of the lines from the source file to the `.aux` file. When `\@copyrestof` returns, the `.aux` file contains an infected version of the source file. `\@parseline` now truncates the `.aux` file.

`\@parseline` now closes and reopens the `.aux` and source files, this time opening the `.aux` file for input and the source file for output, and copies the infected LaTeX file over the original. It then closes the `.aux` file and newly infected file, and loops back to read the next file name from the current line of the listing. When `\@parseline` has attempted to infect all the files on the current line, it returns. When the virus has infected all the LaTeX files named in the listing that Emacs produced, it finishes processing the original LaTeX vector file.

Chapter 4

Problems

There are several problems with the LaTeX virus as it stands. Some of these problems could be resolved with future work, and some are irresolvable. The intent of the research was to prove that such a virus is possible, which has clearly been demonstrated, and not to produce an overly efficient or robust virus.

4.1 Speed

The time required by LaTeX to process an infected file is longer than that required for the uninfected LaTeX file, even when the virus only infects a single target file. While this might be mis-attributed to the slowness of the TeX system, or a load peak on a sporadically loaded multi-user system, eventually it will arouse suspicion. If the infected file was to be processed on a consistently heavily or lightly loaded machine, the difference would be readily apparent. Subsequent work might improve the speed of the virus. Tables 2 and 3, included in Chapter 5, detail the execution time differences for a number of LaTeX file sizes.

4.2 Inaccurate Copy

Without changes to the source code for TeX it is not possible to make a completely accurate copy of the source file. In particular, there are two areas where the copy differs from the original.

- Control Characters — As TeX reads characters, it assigns them catcodes based on the current categories. Since the catcodes determine how TeX treats the characters it reads, we need to change the catcodes of special characters to a value that makes TEX treat them as letters. Since TEX makes no provision for changing the catcodes of a token once it has been initially categorized, this prevents us from being able to write them as anything other than sequences of letters. This manifests itself as the writing of "^^L", the TEX standard representation for control-L in the output in place of a control-L. Since it is not possible to force TEX to read control characters as characters and write them as control codes, the virus sets the catcodes for control characters to "ignore", and control codes aside from carriage returns are lost during the copy. Carriage returns (i.e. empty lines) are worked around as a special case by having TEX write an empty line if it reads one. The only particularly noticeable control code omission is formfeed.

In point of fact, the virus could actually overcome this problem, but it would require macros to be written to check each character read from a source file, comparing it to determine if it encountered two adjacent caret characters, and substituting the correct control code based on the next letter. This would require much greater processing time on the part of the virus, which would now be obliged to examine each token it reads instead of simply writing it out again. Additionally, the macros would need to be written with great care to assure that caret characters in protected environments, such as the LATEX `\verbatim` environment, were not inadvertently transformed into control sequences.

- Additional Spaces — A file copied by TEX will have an additional empty line at the end of the file, and an additional space at the end of every line that contains text. The former is a byproduct of the way that TEX signals end of file, and the latter a byproduct of the way that TEX writes the lines.

Neither of these items can be resolved without modification to the source code for TEX.

4.3 The Need for Emacs

Fortunately for the public at large, but unfortunately for the virus, I was unable to find a mechanism for getting direct access to a directory from within TEX. This is why the virus needs Emacs to dump a list of files. This issue might be resolved with future work.

4.4 Failure to Fully Parse TEX

Since the purpose of the virus was really to prove that it could be done, the implementation is not as robust as it could be. An example of this is the `\@copytostyle` macro, which looks for the closing "}" of the `\documentstyle` macro. It does not take into account possible nesting of parenthesis in parameters, although it does deal correctly with braces hidden by comments and parameters do not end on the same line as the `\documentstyle` macro.

Chapter 5

Results

The LaTeX virus was developed under SunOS 4.1.3 running on a Sun Microsystems SparcStation 2 and NetBSD 0.9a running on a Gateway 80386/33, and tested under the NetBSD operating system. It was developed using TeX version 3.14 and GNU Emacs version 19.22. The virus was implanted into a short LaTeX file, the "small.tex" file provided with the LaTeX distribution. Several test runs were made of the virus, with the average "wall time," the time from the start of processing to the end, shown in the accompanying tables. Ten trials were used to generate this average. Execution times for two "partial" viruses, with one of the features disabled, are included as well. This is an effort to profile the execution of the virus, and isolate areas where performance increases would be most beneficial.

- The partial virus identified as "Infect Only" only creates the infected copy in the `.aux` file, and thus excludes the routine that copies over the original LaTeX file.

- The partial virus identified as "No Findbrace" assumes that the closing "}" for the `\documentstyle` macro is on the same line as the call, and does not use the `\@findbrace` macro to locate the brace.

- The column "With Sig" lists times for the full virus to correctly identify a file that does not contain the signature that allows infection, and abort the infection attempt.

The execution times listed in Table 2 assume that the virus is infecting only a single new LaTeX file of about 180 lines.

While the numbers accurately reflect that the processing time of `sample.tex` and `small.tex` went up dramatically after they had been infected, the virus represents a

Target File	Lines In Infected File	Without Virus (Seconds)	Full Virus (Seconds)	Infect Only (Seconds)	No Findbrace (Seconds)	With Sig (Seconds)
small.tex	241	6.14	9.71	8.21	9.67	6.19
sample.tex	380	6.68	10.70	8.87	10.48	7.10
thesis.tex	919	16.71	20.76	19.54	20.99	17.40
btxdoc.tex	1319	14.15	18.23	16.47	18.15	14.59

Table 2: Average Processing Times Of Various Size Infected Files

substantial increase in size compared to the uninfected file. A real LaTeX file would be considerably larger, and the viral execution times would contribute less of the total time. Based on the partial virus information, performance increases in the wholesale copy routines, such as `\@copytoend`, would be most beneficial.

Table 3 details infection times for a fixed size virus (237 lines) to infect a number of different size targets. The intent here is to highlight the degree to which the size of the file being infected affects the execution time of the virus, and it is clear that these times are fairly proportional to the length of the file being infected.

Target File	File Size	Full Virus (Seconds)	Infect Only (Seconds)	No Findbrace (Seconds)
small.tex	39	8.50	7.59	8.52
sample.tex	179	9.73	8.28	9.62
thesis.tex	720	15.68	10.98	15.15
btxdoc.tex	1117	17.76	12.25	17.84

Table 3: Average Processing Times For Infecting Various Sized Files

Chapter 6

Conclusions

Several conclusions can be drawn from this research. While writing TEX macros is a somewhat arcane art, for an experienced TEX programmer the virus would not be difficult to write. Its size makes it somewhat easier to spot than viruses written in assembly language, but TEX was never designed to be a general purpose programming language.

The virus is slow enough to draw attention to itself, particularly if it attempts to infect multiple files. In an interactive environment, it would surely draw attention to itself before long. Once it was detected, the virus would easily yield its secrets to an investigator, as it makes no attempt to encrypt itself or conceal its function by obfuscation of its code.

6.1 Suggested Modifications to TEX

While the inability to get a directory of files from within TEX presents a significant inconvenience for the virus, other changes to TEX could make the virus as presently implemented impotent. One such change, suggested by Dr. Yair Frankel, is restricting TEX to opening only files of the form `FILE.*` for output, where `FILE` represents the prefix of the input TEX file. This would prevent the virus from being able to open the Emacs initialization file in order to insert the code to dump the directory of `.tex` files. Also, this would prevent the virus from being able to open for output the file it was seeking to infect. It is unclear what, if any, impact this restriction would place on users of the TEX system.

Another modification to TEX would draw attention to the actions of the virus without the possibility of overly restricting users of the system. As currently implemented, TeX outputs to the user the names of files that it opens as part of the `\documentstyle` macro expansion, but not the names of files it opens for reading or writing via the `\openin` or `\openout` macros. A trivial modification to TeX would output the names of these files to the user, who would then have the opportunity to notice something amiss in the files that TeX opens.

6.2 Other Platform Independent Viruses

TeX and LaTeX are used here as a virtual machine on which we have built a virus that runs on several hardware platforms. This thesis clearly proves that platform independent viruses can exist and propagate. Additional viruses of the same type also suggest themselves.

PostScript is a language used for describing the appearance of documents to a printer, but which probably has sufficient complexity to host a virus. In the past, the question was not whether or not such a virus was possible, but rather how would it spread. Since PostScript is interpreted by the printer, which has no access to the secondary storage systems of a computer, the PostScript virus would appear to have no mechanism of infecting other files. The advent of X-Windows based PostScript viewers, such as GNU GhostScript, may present virus writers with a PostScript interpreter which *does* have access to the secondary storage systems of a computer. This is left as further research.

A virus written in Emacs LISP would also have the possibility of executing unchanged on a number of different platforms, but again there is no clear mechanism of spreading from one account or one machine to another. In this case, the LaTeX virus could be used as a vector for a virus that would reside principally in Emacs. This virus has the potential to spread with greater ease than the virus described in this thesis, but would also be prevented by the countermeasures described in the preceding section.

A virus written in PERL [WS91], the Practical Extraction and Reporting Language presents us with an environment where plain text files are interpreted by a software package. PERL is is available on a number of hardware platforms, due to the availability of the source code free of charge. PERL, being a general purpose programming language, does not suffer from the difficulty gaining access to secondary storage cited above.

All of the preceding examples share a common characteristic with the LaTeX virus: they interpret plain text files as instructions. The author believes that any system that so interprets text, and which is in common use, is a potential host environment for a viral program of the sort described in this thesis. Careful consideration needs to be given to the actions that these interpreters carry out on behalf of their input "programs" to prevent them from becoming a vector for a truly damaging virus.

Appendix A

Selected TeX Commands

This chapter presents definitions of several TeX macros used by the virus. These definitions are drawn from *The TeXbook* [Knu84], and are presented in alphabetical order.

`\begingroup` This primitive defines the start of a block of text. Definitions, catcodes, and other features changed inside of a block are only in force inside of the block unless they are declared *global*.

`\catcode` This is a TeX primitive that allows us to retrieve or set the category code for a character. This feature is of critical importance, as it allows us to coerce TeX into treating as letters characters it would normally treat as special.

`\closein` This primitive, the counterpart to the `\openin` primitive, closes a file associated with an input stream.

`\closeout` This primitive, the counterpart of the `\openout` primitive, closes a file associated with an output stream.

`\def` TeX primitive that defines a new macro. The syntax for defining a new macro is

```
\def<control sequence><parameter text>{<replacement text>}
```

The parameter text is not allowed to contain braces, and occurrences of braces in the replacement text must be properly nested. TeX allows us to delimit parameters in a fairly generic way: a hash mark ("#") followed by a numeral 1 through 9 in the parameters represents an argument to the macro. These arguments, if used, must appear only once, and in order in the parameter text, but may occur more than once and in any order in the replacement text. Additionally, characters other than these in the parameter text represent actual tokens in the input to the macro that must be matched. For instance, if the definition of a macro is

```
\def\foo#1 + #2{Hi Mom!}
```

then TEX will expect to find two parameters for this macro, either of which may be empty, separated by a space, a plus symbol, and another space. This mechanism allows us to define a parameter to a macro that consists of multiple tokens, delimited by a symbol of our choosing. This feature is extensively used in the `\@findbrace` and `\@findstyle` macros. Parameters which are not separated by any delimiter are matched by the next single token from the input stream.

`\endgroup` This primitive marks the end of a block of text.

`\expandafter` This primitive causes the token following the next one to be expanded, and then the token immediately following the `\expandafter` to be expanded. This allows the latter to operate on the expansion of a macro following it instead of the macro name.

`\global` This primitive allows a definition to be known outside the current group.

`\if` This primitive is equivalent to a standard programming language if statement, complete with an `\else` clause. It tests whether the next two tokens (after expansion) are identical. The `\if` primitive is usually terminated with a `\fi`. Of particular note is the use of some type of `\if` primitive (there are several) to test for the exit condition of a `\loop` primitive, which does *not* require a terminating `\fi`.

`\ifeof` This primitive, which is similar to the `\if` macro, tests for an end-of-file condition on the input or output stream specified as a parameter.

`\ifx` This primitive is similar to the `\if` primitive, save that it only compares the top level expansions of the next two tokens, instead of their final expansion. This distinction is of particular importance when comparing macros which may have empty expansions.

`\let` This primitive assigns its first parameter (a control sequence, or macro name) the same significance as the second parameter. If the latter is a macro, the former will have the same expansion, and changes to the original macro will be reflected by the expansion of the new macro.

`\long` This primitive allows a macro parameter to include a `\par` token, which symbolizes end-of-paragraph. TEX usually flags as an error any macro parameter which contains `\par` unless the macro is defined with a `\long` modifier.

`\loop` This macro declares a loop which is delimited by a `\repeat` statement. The format of the loop is

```
\loop<statements><if statement><statements>\repeat
```

Either block of statements may be empty. The first block of statements is executed, the `\if` macro is evaluated, and if it is true, the second block of statements is executed. This process is repeated until the `\if` macro evaluates false, at which point the loop terminates.

`\makeatletter` This macro changes the catcode of the @ character from other to letter. This allows us to declare macros with @ as part of the name, which reduces the risk of redefining a macro accidentally.

`\message` This primitive outputs its parameter to the user. While a real virus would desire to be as stealthy as possible, our virus makes use of `\message` to notify the user of its activities.

`\newif` This macro allows us to define a new macro which can be used in the same was as the builtin `\if` macros, along with macros to set its value to true or false.

`\newread` This macro allows us to acquire an input stream number without risk of collision with another macro by allowing TEX to tell us the next available input stream number.

`\newwrite` This macro is analogous to `\newread` for allocating a write stream number.

`\openin` This primitive opens a file for input and associates it with an input stream number.

`\openout` This primitive opens a file for output and associates it with an output stream number.

`\read` This primitive reads the next line from an input stream and defines a macro, the name of which is a parameter to `\read`. The expansion of this new macro is the contents of the line read.

`\relax` This primitive does nothing, but is sometimes necessary to convince TEX of the end of an argument. This is particularly notable in the virus, as this primitive is used whenever the catcode of the space is changed to letter, as `\relax` is needed to make TEX take the space as the argument to the `\@makechar` macro.

`\write` This primitive outputs tokens to the output stream specified as a parameter to the primitive.

Bibliography

[Bru75] John Brunner. *The Shockwave Rider*. Harper and Row, New York, New York, 1975.

[Coh84] Fred Cohen. *Computer Viruses — Theory and Experiments*. PhD thesis, University of Southern California, 1984.

[Duf89] Tom Duff. Viral attacks on unix system security. In *Proceedings of the Winter USENIX Conference*, pages 165–172. USENIX Association, 1989.

[Fer92] David J. Ferbrache. *A Pathology of Computer Viruses*. Springer-Verlag, London, 1992.

[Knu84] Donald E. Knuth. *The TEXbook*. Addison Wesley, Reading, Mass., 1984.

[Lam86] Leslie Lamport. LATEX*: A Document Preparation System*. Addison Wesley, Reading, Mass., 1986.

[RE89] Jon A. Rochlis and Mark W. Eichin. With microscope and tweezers: The worm from MIT's perspective. *Communications of the ACM*, 32(6):689–698, June 1989.

[SH82] John Shoch and Jon Hupp. The worm programs — early experience with distributed computation. *Communications of the ACM*, 25(3), March 1982.

[SH90] Brad Stubbs and Lance J. Hoffman. Mapping the virus battlefield: An overview of personal computer vulnerabilities to virus attack. In Lance J. Hoffman, editor, *Rogue Programs: Viruses, Worms and Trojan Horses*, chapter 12, pages 143–158. Van Nostrand Reinhold, New York, New York, 1990.

[SHF90] Eugene H. Spafford, Kathleen A. Heaphey, and David J. Ferbrache. What is a computer virus? In Lance J. Hoffman, editor, *Rogue Programs: Viruses, Worms and Trojan Horses*, chapter 2, pages 29–42. Van Nostrand Reinhold, New York, New York, 1990.

[Sol92] Alan Solomon. Mechanisms of stealth. In *Proceedings: Fifth International Computer Virus and Security Conference*, pages 374–383. Data Processing Management Association Financial Industries Chapter, 1992.

[Ste90] Suzanne Stefanac. Mad macs. In Lance J. Hoffman, editor, *Rogue Programs: Viruses, Worms and Trojan Horses*, chapter 16, pages 180–193. Van Nostrand Reinhold, New York, New York, 1990.

[WS91] Larry Wall and Randall L. Schwartz. *Programming PERL*. O'Reilly and Associates, Sebastopol, CA, 1991.