Computer Viruses - Theory and Experiments

Introduction and Abstract

This paper defines a major computer security problem called a virus. The virus is interesting because of its ability to attach itself to other programs and cause them to become viruses as well. *There are two spellings for the plural of virus; 'virusses', and 'viruses'. We use the one found in Webster's 3rd International Unabridged Dictionary* Given the wide spread use of sharing in current computer systems, the threat of a virus carrying a Trojan horse [Anderson72] [Linde75] is significant. Although a considerable amount of work has been done in implementing policies to protect from the illicit dissemination of information [Bell73] [Denning82], and many systems have been implemented to provide protection from this sort of attack [McCauley79] [Popek79] [Gold79] [Landwehr83], little work has been done in the area of keeping information entering an area from causing damage [Lampson73] [Biba77]. There are many types of information paths possible in systems, some legitimate and authorized, and others that may be covert [Lampson73], the most commonly ignored one being through the user. We will ignore covert information paths throughout this paper.

The general facilities exist for providing provably correct protection schemes [Feiertag79], but they depend on a security policy that is effective against the types of attacks being carried out. Even some quite simple protection systems cannot be proven 'safe' [Harrison76]. Protection from denial of services requires the detection of halting programs which is well known to be undecidable [Garey79]. The problem of precisely marking information flow within a system [Fenton73] has been shown to be NP-complete. The use of guards for the passing of untrustworthy information [Woodward79] between users has been examined, but in general depends on the ability to prove program correctness which is well known to be NP-complete.

The Xerox worm program [Shoch82] has demonstrated the ability to propagate through a network, and has even accidentally caused denial of services. In a later variation, the game of 'core wars' [Dewdney84] was invented to allow two programs to do battle with one another. Other variations on this theme have been reported by many unpublished authors, mostly in the context of night time games played between programmers. The term virus has also been used in conjunction with an augmentation to APL in which the author places a generic call at the beginning of each function which in turn invokes a preprocessor to augment the default APL interpreter [Gunn74].

The potential threat of a widespread security problem has been examined [Hoffman82] and the potential damage to government, financial, business, and academic institutions is extreme. In addition, these institutions tend to use ad hoc protection mechanisms in response to specific threats rather than sound theoretical techniques [Kaplan82]. Current military protection systems depend to a large degree on isolationism, however new systems are being developed to allow 'multilevel' usage [Klein83]. None of the published proposed systems defines or implements a policy which could stop a virus.

In this paper, we open the new problem of protection from computer viruses. First we examine the infection property of a virus and show that the transitive closure of shared information could potentially become infected. When used in conjunction with a Trojan horse, it is clear that this could cause widespread denial of services and/or unauthorized manipulation of data. The results of several experiments with computer viruses are used to demonstrate that viruses are a formidable threat in both normal and high security operating systems. The paths of sharing, transitivity of information flow, and generality of information interpretation are identified as the key properties in the protection from computer viruses, and a case by case analysis of these properties is shown. Analysis shows that the only systems with potential for protection from a viral attack are systems with limited transitivity and limited sharing, systems with no sharing, and systems without general interpretation of information (Turing capability). Only the first case appears to be of practical interest to current society. In general, detection of a virus is shown to be undecidable both by a-priori and runtime analysis, and without detection, cure is likely to be difficult or impossible.

Several proposed countermeasures are examined and shown to correspond to special cases of the case by case analysis of viral properties. Limited transitivity systems are considered hopeful, but it is shown that precise implementation is intractable, and imprecise policies are shown in general to lead to less and less usable systems with time. The use of system wide viral antibodies is examined, and shown to depend in general on the solutions to intractable problems.

It is concluded that the the study of computer viruses is an important research area with potential applications to other fields, that current systems offer little or no protection from viral attack, and that the only provably 'safe' policy as of this time is isolationism.

A Computer Virus

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

The following pseudo-program shows how a virus might be written in a pseudo-computer language. The ":= symbol is used for definition, the ":" symbol labels a statement, the ";" separates statements, the "=" symbol is used for assignment or comparison, the "~" symbol stands for not, the "{" and "}" symbols group sequences of statements together, and the "..." symbol is used to indicate that an irrelevant portion of code has been left implicit.

```
program virus:=
{1234567;

subroutine infect-executable:=
 {loop:file = get-random-executable-file;
 if first-line-of-file = 1234567 then goto loop;
 prepend virus to file;
 }

subroutine do-damage:=
 {whatever damage is to be done}

subroutine trigger-pulled:=
 {return true if some condition holds}

main-program:=
 {infect-executable;
 if trigger-pulled then do-damage;
 goto next;}

next:}
```

A Simple Virus "V"

This example virus (V) searches for an uninfected executable file (E) by looking for executable files without the "1234567" in the beginning, and prepends V to E, turning it into an infected file (I). V then checks to see if some triggering condition is true, and does damage. Finally, V executes the rest of the program it was prepended to. When the user attempts to execute E, I is executed in its place; it infects another file and then executes as if it were E. With the exception of a slight delay for infection, I appears to be E until the triggering condition causes damage.

A common misconception of a virus relates it to programs that simply propagate through networks. The worm program, 'core wars', and other similar programs have done this, but none of them actually involve infection. The key property of a virus is its ability to infect other programs, thus reaching the transitive closure of sharing between users. As an example, if V infected one of user A's executables (E), and user B then ran E, V could spread to user B's files as well.

It should be pointed out that a virus need not be used for evil purposes or be a Trojan horse. As an example, a compression virus could be written to find uninfected executables, compress them upon the user's permission, and prepend itself to them. Upon execution, the infected program decompresses itself and executes normally. Since it always asks permission before performing services, it is not a Trojan horse, but since it has the infection property, it is still a virus. Studies indicate that such a virus could save over 50% of the space taken up by executable files in an average system. The performance of infected programs would decrease slightly as they are decompressed, and thus the compression virus implements a particular time space tradeoff. A sample compression virus could be written as follows:

```
program compression-virus:=
{01234567;

subroutine infect-executable:=
 {loop:file = get-random-executable-file;
 if first-line-of-file = 01234567 then goto loop;
 compress file;
 prepend compression-virus to file;
 }

main-program:=
 {if ask-permission then infect-executable;
 uncompress the-rest-of-this-file into tmpfile;
 run tmpfile;}
}
```

A Compression Virus "C"

This program (C) finds an uninfected executable (E), compresses it, and prepends C to form an infected executable (I). It then uncompresses the rest of itself into a temporary file and executes normally. When I is run, it will seek out and compress another executable before decompressing E into a temporary file and executing it. The effect is to spread through the system compressing executable files, and decompress them as they are to be executed. Users will experience significant delays as their executables are decompressed before being run.

As a more threatening example, let us suppose that we modify the program V by specifying trigger-pulled as true after a given date and time, and specifying do-damage as an infinite loop. With the level of sharing in most modern systems, the entire system would likely become unusable as of the specified date and time. A great deal of work might be required to undo the damage of such a virus. This modification is shown here:

```
...
subroutine do-damage:=
 {loop: goto loop;}

subroutine trigger-pulled:=
 {if year>1984 then return true otherwise return false;}
...
```

A Denial of Services Virus

As an analogy to a computer virus, consider a biological disease that is 100% infectious, spreads whenever animals communicate, kills all infected animals instantly at a given moment, and has no detectable side effects until that moment. If a delay of even one week were used between the introduction of the disease and its effect, it would be very likely to leave only a few remote villages alive, and would certainly wipe out the vast majority of modern society. If a computer virus of this type could spread throughout the computers of the world, it would likely stop most computer usage for a significant period of time, and wreak havoc on modern government, financial, business, and academic institutions.

Prevention of Computer Viruses

We have introduced the concept of viruses to the reader, and actual viruses to systems. Having planted the seeds of a potentially devastating attack, it is appropriate to examine protection mechanisms which might help defend against it. We examine here prevention of computer viruses.

Basic Limitations

In order for users of a system to be able to share information, there must be a path through which information can flow from one user to another. We make no differentiation between a user and a program acting as a surrogate for that user since a program always acts as a surrogate for a user in any computer usage, and we are ignoring the covert channel through the user. In order to use a Turing machine model for computation, we must consider that if information can be read by a user with Turing capability, then it can be copied, and the copy can then be treated as data on a Turing machine tape.

Given a general purpose system in which users are capable of using information in their possession as they wish and passing such information as they see fit to others, it should be clear that the ability to share information is transitive. That is, if there is a path from user A to user B, and there is a path from user B to user C, then there is a path from user A to user C with the witting or unwitting cooperation of user B.

Finally, there is no fundamental distinction between information that can be used as data, and information that can be used as program. This can be clearly seen in the case of an interpreter that takes information edited as data, and interprets it as a program. In effect, information only has meaning in that it is subject to interpretation.

In a system where information can be interpreted as a program by its recipient, that interpretation can result in infection as shown above. If there is sharing, infection can spread through the interpretation of shared information. If there is no restriction on the transitivity of information flow, then the information can reach the transitive closure of information flow starting at any source. Sharing, transitivity of information flow, and generality of interpretation thus allow a virus to spread to the transitive closure of information flow starting at any given source.

Clearly, if there is no sharing, there can be no dissemination of information across information boundaries, and thus no external information can be interpreted, and a virus cannot spread outside a single partition. This is called 'isolationism'. Just as clearly, a system in which no program can be altered and information cannot be used to make decisions cannot be infected since infection requires the modification of interpreted information. We call this a 'fixed first order functionality' system. We should note that virtually any system with real usefulness in a scientific or development environment will require generality of interpretation, and that isolationism is unacceptable if we wish to benefit from the work of others. Nevertheless, these are solutions to the problem of viruses which may be applicable in limited situations.

Partition Models

Two limits on the paths of information flow can be distinguished, those that partition users into closed proper subsets under transitivity, and those that don't. Flow restrictions that result in closed subsets can be viewed as partitions of a system into isolated subsystems. These limit each infection to one partition. This is a viable means of preventing complete viral takeover at the expense of limited isolationism, and is equivalent to giving each partition its own computer.

The integrity model [Biba77] is an example of a policy that can be used to partition systems into closed subsets under transitivity. In the Biba model, an integrity level is associated with all information. The strict integrity properties are the dual of the Bell-LaPadula properties; no user at a given integrity level can read an object of lower integrity or write an object of higher integrity. In Biba's original model, a distinction was made between read and execute access, but this cannot be enforced without restricting the generality of information interpretation since a high integrity program can write a low integrity object, make low integrity copies of itself, and then read low integrity input and produce low integrity output.

If the integrity model and the Bell-LaPadula model coexist, a form of limited isolationism results which divides the space into closed subsets under transitivity. If the same divisions are used for both mechanisms (higher integrity corresponds to higher security), isolationism results since information moving up security levels also moves up integrity levels, and this is not permitted. When the Biba model has boundaries within the Bell-LaPadula boundaries, infection can only spread from the higher integrity levels to lower ones within a given security level. Finally, when the Bell-LaPadula boundaries are within the Biba boundaries, infection can only spread from lower security levels to higher security levels within a given integrity level. There are actually 9 cases corresponding to all pairings of lower boundaries with upper boundaries, but the three shown graphically below are sufficient for understanding.

```
 Same Divisions       Biba within B-L      B-L within Biba
  ---------------------  --------------------  ---------------------
Biba   B-L   Result    Biba   B-L   Result    Biba   B-L   Result
------ ----  --------- ----   ----  --------- ----   ----  ---------
|\\|   |//|  |XX|      |\\|   |//|  |XX|      |\\|   |//|  |XX|
|\\|   |//|  |XX|      |\\|   | |   |\\|      | |    |//|  |//|
| |  + | |  =|   |     | |  + | |  =|   |     | |  + | |  =| |
|//|   |\\|  |XX|      |//|   | |   |//|      | |    |\\|  |\\|
|//|   |\\|  |XX|      |//|   |\\|  |XX|      |//|   |\\|  |XX|
------ ----  --------- ----   ----  --------- ----   ----  ---------
\\ = can't write   // = can't read    XX = no access    \ + / = X
```

Biba's work also included two other integrity policies, the 'low water mark' policy which makes output the lowest integrity of any input, and the 'ring' policy in which users cannot invoke everything they can read. The former policy tends to move all information towards lower integrity levels, while the latter attempts to make a distinction that cannot be made with generalized information interpretation.

Just as systems based on the Bell-LaPadula model tend to cause all information to move towards higher levels of security by always increasing the level to meet the highest level user, the Biba model tends to move all information towards lower integrity levels by always reducing the integrity of results to that of the lowest incoming integrity. We also know that a precise system for integrity is NP-complete (just as its dual is NP-complete).

The most trusted programmer is (by definition) the programmer that can write programs executable by the most users. In order to maintain the Bell-LaPadula policy, high level users cannot write programs used by lower level users. This means that the most trusted programmers must be those at the lowest security level. This seems contradictory. When we mix the Biba and Bell-LaPadula models, we find that the resulting isolationism secures us from viruses, but doesn't permit any user to write programs that can be used throughout the system. Somehow, just as we allow encryption or declassification of data to move it from higher security levels to lower ones, we should be able to use program testing and verification to move information from lower integrity levels to higher ones.

Another commonly used policy used to partition systems into closed subsets is the category policy used in typical military applications. This policy partitions users into categories, with each user only able to access information required for their duties. If every user in a strict category system has access to only one category at a time, the system is secure from viral attack across category boundaries because they are isolated. Unfortunately, in current systems, users may have simultaneous access to multiple categories. In this case, infection can spread across category boundaries to the transitive closure of information flow.

Flow Models

In policies that don't partition systems into closed proper subsets under transitivity, it is possible to limit the extent over which a virus can spread. The 'flow distance' policy implements a distance metric by keeping track of the distance (number of sharings) over which data has flowed. The rules are; the distance of output information is the maximum of the distances of input information, and the distance of shared information is one more than the distance of the same information before sharing. Protection is provided by enforcing a threshold above which information becomes unusable. Thus a file with distance 8 shared into a process with distance 2 increases the process to distance 9, and any further output will be at at least that distance.

As an example, we show the flow allowed to information in a distance metric system with the threshold set at 1 and each user (A-E) able to communicate with only the 2 nearest neighbors. Notice that information starting at C can only flow to user B or user D, but cannot transit to A or E even with the cooperation of B and D.

```
Rules:
  D(output) = max(D(input))
  D(shared input)=1+D(unshared input)
  Information is accessible iff D < const

   A     B     C     D     E
  +-+   +-+   +-+   +-+   +-+
  |X|---|1|---|0|---|1|---|X|
  +-+   +-+   +-+   +-+   +-+
```

A Distance Metric with a Threshold of 1

The 'flow list' policy maintains a list of all users who have had an effect on each object. The rule for maintaining this list is; the flow list of output is the union of the flow lists of all inputs (including the user who causes the action). Protection takes the form of an arbitrary boolean expression on flow lists which determines accessibility. This is a very general policy, and can be used to represent any of the above policies by selecting proper boolean expressions.

The following figure shows an example of a flow list system implementing different restrictions (indicated by A and B) for different users (at row,col 1,3 and 2,5). Notice that although information is allowed to get to 1,5, it can't actually get there because there is no path from its source at 1,3. As in the distance metric system, transitivity of information flow does not hold, so that even if information indicated by B were able to reach 2,3, it could not transit any further.

Rules:

```
  F(output)=Union(F(inputs))
  Information is accessible iff B(F)=1
   1    2    3    4    5    6
  +-+  +-+  +-+  +-+  +-+  +-+
1 |A|---|A|---|A|---| |---|A|---|B|
  +-+  +-+  +-+  +-+  +-+  +-+
   |    |    |    |    |    |
  +-+  +-+  +-+  +-+  +-+  +-+
2 | |---| |---|A|---| |---|B|---|B|
  +-+  +-+  +-+  +-+  +-+  +-+
   |    |    |    |    |    |
  +-+  +-+  +-+  +-+  +-+  +-+
3 |B|---|B|---|B|---|B|---|B|---|B|
  +-+  +-+  +-+  +-+  +-+  +-+
```

A Sample Flow List System

The above example uses a fairly simple boolean function, but in general, very complex conditionals can be used to determine accessibility. As an example, user A could only be allowed to access information written by users (B and C) or (B and D), but not information written by B, C, or D alone. This can be used to enforce certification of information by B before C or D can pass it to A. The flow list system can also be used to implement the Biba and the distance models. As an example, the distance model can be realized as follows: @center[OR(users <= distance 1) AND NOT(OR(users > distance 1))]

In a system with unlimited information paths, limited transitivity may have an effect if users don't use all available paths, but since there is always a direct path between any two users, there is always the possibility of infection. As an example, in a system with transitivity limited to a distance of 1 it is 'safe' to share information with any user you 'trust' without having to worry about whether that user has incorrectly trusted another user.

Limited Interpretation

With limits on the generality of interpretation less restrictive than fixed first order interpretation, the ability to infect is an open question because infection depends on the functions permitted. Certain functions are required for infection. The ability to write is required, but any useful program must have output. It is possible to design a set of operations that don't allow infection in even the most general case of sharing and transitivity, but it is not known whether any such set includes non fixed first order functions.

As an example, a system with only the function 'display-file' can only display the contents of a file to a user, and cannot possibly modify any file. In fixed database or mail systems, this may have practical applications, but certainly not in a development environment. In many cases, computer mail is a sufficient means of communications, and so long as the computer mail system is partitioned from other applications so that no information can flow between them except in the covert channel through the user, this may be used to prevent infection.

Although no fixed interpretation scheme can itself be infected, a high order fixed interpretation scheme can be used to infect programs written to be interpreted by it. As an example, the microcode of a computer may be fixed, but code in the machine language it interprets can still be infected. LISP, APL, and Basic are all examples of fixed interpretation schemes that can interpret information in general ways. Since their ability to interpret is general, it is possible to write a program in any of these languages that infects programs in any or all of these languages.

In limited interpretation systems, infections cannot spread any further than in general interpretation systems, because every function in a limited system must also be able to be performed in a general system. The previous results therefore provide upper bounds on the spread of a virus in systems with limited interpretation.

Precision Problems

Although isolationism and limited transitivity offer solutions to the infection problem, they are not ideal in the sense that widespread sharing is generally considered a valuable tool in computing. Of these policies, only isolationism can be precisely implemented in practice because tracing exact information flow requires NP-complete time, and maintaining markings requires large amounts of space [Denning82]. This leaves us with imprecise techniques. The problem with imprecise techniques is that they tend to move systems towards isolationism. This is because they use conservative estimates of effects in order to prevent potential damage. The philosophy behind this is that it is better to be safe than sorry.

The problem is that when information has been unjustly deemed unreadable by a given user, the system becomes less usable for that user. This is a form of denial of services in that access to information that should be accessible is denied. Such a system always tends to make itself less and less usable for sharing until it either becomes completely isolationist or reaches a stability point where all estimates are precise. If such a stability point existed, we would have a precise system for that stability point. Since we know that any precise stability point besides isolationism requires the solution to an NP-complete problem, we know that any non NP-complete solution must tend towards isolationism.

Summary and Conclusions

The following table summarizes the limits placed on viral spreading by the preventative protection just examined. Unknown is used to indicate that the specifics of specific systems are known, but that no theory has been shown to predict limitations in these categories.

Limits of viral infection

	general interpretation		limited interpretation	
\Transitivity Sharing\	limited	general	limited	general
general	unlimited	unlimited	unknown	unknown
limited	arbitrary	closure	arbitrary	closure

Cure of Computer Viruses

Since prevention of computer viruses may be infeasible if widespread sharing is desired, the biological analogy leads us to the possibility of cure as a means of protection. Cure in biological systems depends on the ability to detect a virus and find a way to overcome it. A similar possibility exists for computer viruses. We now examine the potential for detection and removal of a computer virus.

Detection of Viruses

In order to determine that a given program 'P' is a virus, it must be determined that P infects other programs. This is undecidable since P could invoke the decision procedure 'D' and infect other programs if and only if D determines that P is not a virus. We conclude that a program that precisely discerns a virus from any other program by examining its appearance is infeasible. In the following modification to program V, we use the hypothetical decision procedure D which returns "true" iff its argument is a virus, to exemplify the undecidability of D.

```
program contradictory-virus:=
{...
main-program:=
 {if ~D(contradictory-virus) then
    {infect-executable;
    if trigger-pulled then do-damage;
    }
 goto next;
 }
}
```

Contradiction of the Decidability of a Virus "CV"

By modifying the main-program of V, we have assured that if the decision procedure D determines CV to be a virus, CV will not infect other programs, and thus will not act as a virus. If D determines that CV is not a virus, CV will infect other programs, and thus be a virus. Therefore, the hypothetical decision procedure D is self contradictory, and precise determination of a virus by its appearance is undecidable.

Evolutions of a Virus

In our experiments, some viruses took less than 4000 bytes to implement on a general purpose computer. Since we could interleave any program that doesn't halt, terminates in finite time, and doesn't overwrite the virus or any of its state variables, and still have a virus, the number of possible variations on a single virus is clearly very large. In this example of an evolutionary virus EV, we augment V by allowing it to add random statements between any two necessary statements.

```
program evolutionary-virus:=
{...
subroutine print-random-statement:=
 {print random-variable-name, " = ", random-variable-name;
 loop:if random-bit = 0 then
    {print random-operator, random-variable-name;
```

```
   goto loop;}
 print semicolon;
 }

subroutine copy-virus-with-random-insertions:=
 {loop: copy evolutionary-virus to virus till semicolon-found;
 if random-bit = 1 then print-random-statement;
 if ~end-of-input-file goto loop;
 }

main-program:=
 {copy-virus-with-random-insertions;
 infect-executable;
 if trigger-pulled do-damage;
 goto next;}

next:}
```

An Evolutionary Virus "EV"

In general, proof of the equivalence of two evolutions of a program 'P' ('P1' and 'P2') is undecidable because any decision procedure 'D' capable of finding their equivalence could be invoked by P1 and P2. If found equivalent they perform different operations, and if found different they act the same, and are thus equivalent. This is exemplified by the following modification to program EV in which the decision procedure D returns "true" iff two input programs are equivalent.

```
program undecidable-evolutionary-virus:=
{...
subroutine copy-with-undecidable-assertion:=
 {copy undecidable-evolutionary-virus to file till line-starts-with-zzz;
 if file = P1 then print "if D(P1,P2) then print 1;";
 if file = P2 then print "if D(P1,P2) then print 0;";
 copy undecidable-evolutionary-virus to file till end-of-input-file;
 }

main-program:=
 {if random-bit = 0 then file = P1 otherwise file = P2;
 copy-with-undecidable-assertion;
 zzz:
 infect-executable;
 if trigger-pulled do-damage;
 goto next;}

next:}
```

Undecidable Equivalence of Evolutions of a Virus "UEV"

The program UEV evolves into one of two types of programs P1 or P2. If the program type is P1, the statement labeled "zzz" will become:

```
        if D(P1,P2) then print 1;
```

while if the program type is P2, the statement labeled "zzz" will become:

```
        if D(P1,P2) then print 0;
```

The two evolutions each call decision procedure D to decide whether they are equivalent. If D indicates that they are equivalent, then P1 will print a 1 while P2 will print a 0, and D will be contradicted. If D indicates that they are different, neither prints anything. Since they are otherwise equal, D is again contradicted. Therefore, the hypothetical decision procedure D is self contradictory, and the precise determination of the equivalence of these two programs by their appearance is undecidable.

Since both P1 and P2 are evolutions of the same program, the equivalence of evolutions of a program is undecidable, and since they are both viruses, the equivalence of evolutions of a virus is undecidable. Program UEV also demonstrates that two unequivalent evolutions can both be viruses. The evolutions are equivalent in terms of their viral effects, but may have slightly different side effects.

An alternative to detection by appearance, is detection by behavior. A virus, just as any other program, acts as a surrogate for the user in requesting services, and the services used by a virus are legitimate in legitimate uses. The behavioral detection question then becomes one of defining what is and is not a legitimate use of a system service, and finding a means of detecting the difference.

As an example of a legitimate virus, a compiler that compiles a new version of itself is in fact a virus by the definition given here. It is a program that 'infects' another program by modifying it to include an evolved version of itself. Since the viral capability is in most compilers, every use of a compiler is a potential viral attack. The viral activity of a compiler is only triggered by particular inputs, and thus in order to detect triggering, one must be able to detect a virus by its appearance. Since precise detection by behavior in this case depends on precise detection by the appearance of the inputs, and since we have already shown that precise detection by appearance is undecidable, it follows that precise detection by behavior is also undecidable.

Limited Viral Protection

A limited form of virus has been designed [Thompson84] in the form of a special version of the C compiler that can detect the compilation of the login program and add a Trojan horse that lets the author login. Thus the author could access any Unix system with this compiler. In addition, the compiler can detect compilations of new versions of itself and infect them with the same Trojan horse. Whether or not this has actually been implemented is unknown (although many say the NSA has a working version of it).

As a countermeasure, we can devise a new login program (and C compiler) sufficiently different from the original as to make its equivalence very difficult to determine. If the 'best AI program of the day' would be incapable of detecting their equivalence in a given amount of time, and the compiler performed its task in less than that much time, it could be reasonably assumed that the virus could not have detected the equivalence, and therefore would not have propagated itself. If the exact nature of the detection were known, it would likely be quite simple to work around it.

Although we have shown that in general it is impossible to detect viruses, any particular virus can be detected by a particular detection scheme. For example, virus V could easily be detected by looking for 1234567 as the first line of an executable. If the executable were found to be infected, it would not be run, and would therefore not be able to spread. The following program is used in place of the normal run command, and refuses to execute programs infected by virus V:

```
program new-run-command:=
{file = name-of-program-to-be-executed;
if first-line-of-file = 1234567 then
 {print "the program has a virus";
 exit;}
otherwise run file;
}
```

Protection from Virus V "PV"

Similarly, any particular detection scheme can circumvented by a particular virus. As an example, if an attacker knew that a user was using the program PV as protection from viral attack, the virus V could easily be substituted with a virus V' where the first line was 123456 instead of 1234567. Much more complex defense schemes and viruses can be examined. What becomes quite evident is analogous to the old western saying: "ain't a horse that can't be rode, ain't a man that can't be throwed". No infection can exist that cannot be detected, and no detection mechanism can exist that can't be infected.

This result leads to the idea that a balance of coexistent viruses and defenses could exist, such that a given virus could only do damage to a given portion of the system, while a given protection scheme could only protect against a given set of viruses. If each user and attacker used identical defenses and viruses, there could be an ultimate virus or defense. It makes sense from both the attacker's point of view and the defender's point of view to have a set of (perhaps incompatible) viruses and defenses.

In the case where viruses and protection schemes didn't evolve, this would likely lead to some set of fixed survivors, but since programs can be written to evolve, the program that evolved into a difficult to attack program would more likely survive as would a virus that was more difficult to detect. As evolution takes place, balances tend to change, with the eventual result being unclear in all but the simplest circumstances. This has very strong analogies to biological theories of evolution [Dawkins78], and might relate well to genetic theories of diseases. Similarly, the spread of viruses through systems might well be analyzed by using mathematical models used in the study of infectious diseases [Baily57].

Since we cannot precisely detect a virus, we are left with the problem of defining potentially illegitimate use in a decidable and easily computable way. We might be willing to detect many programs that are not viruses and even not detect some viruses in order to detect a large number of viruses. If an event is relatively rare in 'normal' use, it has high information content when it occurs, and we can define a threshold at which reporting is done. If sufficient instrumentation is available, flow lists can be kept which track all users who have effected any given file. Users that appear in many incoming flow lists could be considered suspicious. The rate at which users enter incoming flow lists might also be a good indicator of a virus.

This type of measure could be of value if the services used by viruses are rarely used by other programs, but presents several problems. If the threshold is known to the attacker, the virus can be made to work within it. An intelligent thresholding scheme could adapt so the threshold could not be easily determined by the attacker. Although this 'game' can clearly be played back and forth, the frequency of dangerous requests might be kept low enough to slow the undetected virus without interfering significantly with legitimate use.

Several systems were examined for their abilities to detect viral attacks. Surprisingly, none of these systems even include traces of the owner of a program run by other users. Marking of this sort must almost certainly be used if even the simplest of viral attacks are to be detected.

Once a virus is implanted, it may not be easy to fully remove. If the system is kept running during removal, a disinfected program could be reinfected. This presents the potential for infinite tail chasing. Without some denial of services, removal is likely to be impossible unless the program performing removal is faster at spreading than the virus being removed. Even in cases where the removal is slower than the virus, it may be possible to allow most activities to continue during removal without having the removal process be very fast. For example, one could isolate a user or subset of users and cure them without denying services to other users.

In general, precise removal depends on precise detection, because without precise detection, it is impossible to know precisely whether or not to remove a given object. In special cases, it may be possible to perform removal with an inexact algorithm. As an example, every file written after a given date could be removed in order to remove any virus started after that date.

One concern that has been expressed and is easily laid to rest is the chance that a virus could be spontaneously generated. This is strongly related to the question of how long it will take N monkeys at N keyboards to create a virus, and is laid to rest with similar dispatch.

Summary, Conclusions, and Further Work

To quickly summarize, absolute protection can be easily attained by absolute isolationism, but that is usually an unacceptable solution. Other forms of protection all seem to depend on the use of extremely complex and/or resource intensive analytical techniques, or imprecise solutions that tend to make systems less usable with time.

Prevention appears to involve restricting legitimate activities, while cure may be arbitrarily difficult without some denial of services. Precise detection is undecidable, however statistical methods may be used to limit undetected spreading either in time or in extent. Behavior of typical usage must be well understood in order to use statistical methods, and this behavior is liable to vary from system to system. Limited forms of detection and prevention could be used in order to offer limited protection from viruses.

It has been demonstrated that a virus has the potential to spread throughout any system which allows sharing. Every general purpose system currently in use is open to at least limited viral attack. In many current 'secure' systems, viruses tend to spread further when created by less trusted users. Experiments show the viability of viral attack, and indicate that viruses spread quickly and are easily created on a variety of operating systems. Further experimentation is still underway.

The results presented are not operating system or implementation specific, but are based on the fundamental properties of systems. More importantly, they reflect realistic assumptions about systems currently in use. Further, nearly every 'secure' system currently under development is based on the Bell-LaPadula or lattice policy alone, and this work has clearly demonstrated that these models are insufficient to prevent viral attack. The virus essentially proves that integrity control must be considered an essential part of any secure operating system.

Several undecidable problems have been identified with respect to viruses and countermeasures. The are summarized here:

Undecidable Detection Problems

- Detection of a virus by its appearance
- Detection of a virus by its behavior
- Detection of an evolution of a known virus
- Detection of a triggering mechanism by its appearance
- Detection of a triggering mechanism by its behavior

- Detection of an evolution of a known triggering mechanism
- Detection of a virus detector by its appearance
- Detection of a viral detector by its behavior
- Detection of an evolution of a known viral detector

Several potential countermeasures were examined in some depth, and none appear to offer ideal solutions. Several of the techniques suggested in this paper which could offer limited viral protection are in limited use at this time. To be perfectly secure against viral attacks, a system must protect against incoming information flow, while to be secure against leakage of information a system must protect against outgoing information flow. In order for systems to allow sharing, there must be some information flow. It is therefore the major conclusion of this paper that the goals of sharing in a general purpose multilevel security system may be in such direct opposition to the goals of viral security as to make their reconciliation and coexistence impossible.

The most important ongoing research involves the effect of viruses on computer networks. Of primary interest is determining how quickly a virus could spread to a large percentage of the computers in the world. This is being done through simplified mathematical models and studies of viral spreading in 'typical' computer networks. The implications of a virus in a secure network are also of great interest. Since the virus leads us to believe that both integrity and security must be maintained in a system in order to prevent viral attack, a network must also maintain both criterion in order to allow multilevel sharing between computers. This introduces significant constraints on these networks.

Significant examples of evolutionary programs have been developed at the source level for producing many evolutions of a given program. A simple evolving virus has been developed, and a simple evolving antibody is also under development. A flow list mechanism for Unix will be implemented when the necessary hardware is available, and the instrumentation of networks is expected to continue as long as facilities and funding permit. Statistical detection techniques based on the results of instrumentation are also in the planning stages, and a set of guidelines for reducing the viral threat have been developed.

Acknowledgements

Because of the sensitive nature of much of this research and the experiments performed in its course, many of the people to whom I am greatly indebted cannot be explicitly thanked. Rather than ignoring anyone's help, I have decided to give only first names. Len and David have provided a lot of good advice in both the research and writing of this paper, and without them I likely would never have gotten it to this point. John, Frank, Connie, Chris, Peter, Terry, Dick, Jerome, Mike, Marv, Steve, Lou, Steve, Andy, and Loraine all put their noses on the line more than just a little bit in their efforts to help perform experiments, publicize results, and lend covert support to the work. Martin, John, Magdy, Xi-an, Satish, Chris, Steve, JR, Jay, Bill, Fadi, Irv, Saul, and Frank all listened and suggested, and their patience and friendship were invaluable. Alice, John, Mel, Ann, and Ed provided better blocking than the USC front 4 ever has.