Computer Viruses

Theory and Experiments

Fred Cohen
Dept. of Computer Science and Electric Engineering, Lehigh University, Bethlehem, PA 18215, USA, and The Foundation for Computer Integrity Research, Pittsburgh, PA 15217, USA.

This paper introduces "computer viruses" and examines their potential for causing widespread damage to computer systems. Basic theoretical results are presented, and the infeasibility of viral defense in large classes of systems is shown. Defensive schemes are presented and several experiments are described.

Keywords: Computer Viruses, System Integrity, Data Integrity.

Fred Cohen received a B.S. in Electrical Engineering from Carnegie-Mellon University in 1977, an MS in Information Science from the University of Pittsburgh in 1981 and a Ph.D. in Electrical Engineering from the University of Southern California in 1986.

He has worked as a freelance consultant since 1977, and has designed and implemented numerous devices and systems. He is currently a professor of Computer Science and Electrical Engineering at Lehigh University, Chairman and Director of Engineering at the Foundation for Computer Integrity Research, and President of Legal Software Incorporated.

He is a member of the ACM, IEEE, and IACR. His current research interests include computer viruses, information flow model, adaptive systems theory, genetic models of computing, and evolutionary systems.

0167-4048/87/$3.50 © 1987, Elsevier Science Publishers B.V. (North-Holland)

1. Introduction

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. Given the widespread use of sharing in current computer systems, the threat of a virus carrying a Trojan horse [1,20] is significant. Although a considerable amount of work has been done in implementing policies to protect against the illicit dissemination of information [4,7], and many systems have been implemented to provide protection from this sort of attack [12,19,21,22], little work has been done in the area of keeping information entering an area from causing damage [5,18]. There are many types of information paths possible in systems, some legitimate and authorized, and others that may be covert [18], 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 [9], 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' [14]. Protection from denial of services requires the detection of halting programs which is well known to be undecidable [11]. The problem of precisely marking information flow within a system [10] has been shown to be NP-complete. The use of guards for the passing of untrustworthy information [25] 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 [23] 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' [8] 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 nighttime 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 [13].

The potential threat of a widespread security problem has been examined [15] 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 [16]. Current military protection systems depend to a large degree on isolationism [3]; however, new systems are being developed to allow 'multilevel' usage [17]. 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.

2. 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.

This example virus (V) (Fig. 1) 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

```
program virus :=
{1234567;

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

subroutine do-damage :=
   {whatever damage is desired}

subroutine trigger-pulled :=
   {return true on desired conditions}

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

next:}
```

Fig. 1. Simple virus 'V'.

triggering condition is true, and does damage. Finally, V executes the rest of the program it was prepended [1] 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. We note that viruses need not prepend themselves nor must they be restricted to a single infection per use.

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 in Fig. 2.

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

subroutine infect-executable :=
  {loop: file = random-executable;
  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;
  }
}
```

Fig. 2. 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, decompressing 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 in Fig. 3.

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 through the computers of the world, it would likely stop most computer use for a significant period of time, and wreak havoc on modern government, financial, business, and academic institutions.

```
...

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

subroutine trigger-pulled :=
  {if year > 1984 then return(true)
        otherwise return(false);

...
```

Fig. 3. A denial of services virus.

[1] The term 'prepend' is used in a technical sense in this paper to mean 'attach at the beginning'.

3. 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.

3.1 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 use and we are ignoring the covert channel through the user. Assuming a Turing machine model for computation, we can prove 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 its 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.

3.2 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 do not. 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 [5] 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 nine cases corresponding to all pairings of lower boundaries with upper boundaries, but the three shown graphically in Fig. 4 are sufficient for understanding.

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.

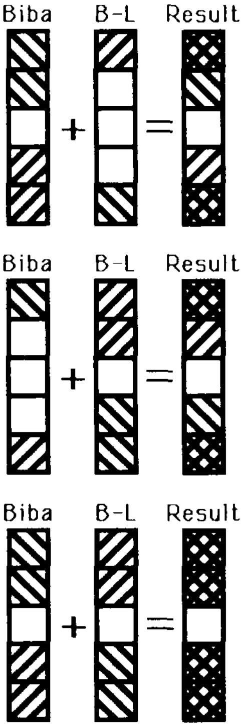

Fig. 4. Pairings of lower boundaries with upper boundaries. Top: Biba within B-L; middle: B-L within Biba; bottom: same divisions. \\ cannot write; // cannot read; × × no access; \ + / = ×.

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 does not 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 that partitions systems into closed subsets is the compartment policy used in typical military applications. This policy partitions users into compartments, with each user only able to access information required for their duties. If every user has access to only one compartment at a time, the system is secure from viral attack across compartment boundaries because they are isolated. Unfortunately, in current systems, users may have simultaneous access to multiple compartments. In this case, infection can spread across these boundaries to the transitive closure of information flow.

3.3 Flow Models

In policies that do not 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 least that distance.

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.

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:

$$\text{OR}(\text{users} \leqslant \text{distance } 1)$$

$$\text{AND NOT}(\text{OR}(\text{users} > \text{distance } 1))$$

A further generalization of flow lists to flow sequences is possible, and appears to be the most general scheme possible for implementing a flow control policy.

In a system with unlimited information paths, limited transitivity may have an effect if users do not 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.

3.4 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 do not 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 them.

In limited interpretation systems, infection 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.

3.5 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

	General Interpretation		Limited Interpretation	
sharing \ transitivity	limited	general	limited	general
general	unlimited	unlimited	unknown	unknown
limited	arbitrary	closure	arbitrary	closure

Fig. 5. Limits of viral infection.

space [7]. 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.

3.6 Summary and Conclusions

Fig. 5 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 general theory has been shown to predict limitations in these categories.

4. Cure of Computer Viruses

Since prevention of computer viruses may be infeasible if 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.

4.1 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 any proposed 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* (Fig. 6), we use the hypothetical decision procedure *D* which returns "true" iff its argument is a virus, to exemplify the undecidability of viral detection.

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.

4.2 Evolutions of a Virus

In our experiments, some viruses took under 100 bytes to implement on a general purpose computer. Since we could interleave any program that doesn't halt, terminates in finite time, and does not 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 state-

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

Fig. 6. Contradiction of the decidability of a virus 'C'.

ments between any two necessary statements (Fig. 7).

In general, proof of the equivalence of two evolutions of a program 'P' ('P_1' and 'P_2') is undecidable because any decision procedure 'D' capable of finding their equivalence could be invoked by P_1 and P_2. If found equivalent they perform different operations, and if found different they act the same, and are thus equivalent. This is exemplified by the modification in Fig. 8 to program EV in which the decision procedure D returns "true" iff two input programs are equivalent.

The program UEV evolves into one of two types of programs, P_1 or P_2. If the program type is P_1 the statement labeled "*zzz*" will become:

if $D(P_1, P_2)$ then print 1;

while if the program type is P_2, the statement labeled "*zzz*" will become:

if $D(P_1, P_2)$ 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 P_1 will print a 1 while P_2 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.

```
program evolutionary-virus :=
{...
subroutine print-random-statement :=
  {print (random-variable-name, "=",
          random-variable-name);
  loop: if random-bit = 1 then
        {print (random-operator,
                random-variable-name);
        goto loop;}
  print (semicolon);
  }

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

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

next:}
```

Fig. 7. Evolutionary virus 'EV'.

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

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

next:}
```

Fig. 8. Undecidable equivalence of evolutions of a virus 'UEV'.

Since both P_1 and P_2 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.

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 leads to 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.

4.3 Limited Viral Protection

A limited form of virus has been designed [24] 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.

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. Once a virus free compiler is generated, the old (and presumably more efficient) version can be recompiled for further use.

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 program in Fig. 9 is used in place of the normal run command, and refuses to execute programs infected by virus *V*.

Similarly, any particular detection scheme can be 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 that no infection can exist that cannot be detected, and no detection mechanism can exist that cannot be infected.

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

Fig. 9. Protection from virus V 'PV'.

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 do not evolve, this would likely lead to some st of fixed survivors, but program (or virus) that evolves into a diffcult to attack program (or virus) is more likely to survive. 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 [6], 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 [2].

Since we cannot precisely detect a virus, we are left with the problem of defining potentially illigitimate 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 affected 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 can 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 infection can 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 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. This may be quite painful if viruses are designed to have long waiting periods before doing damage, since even backups would have to be discarded to fully cleanse the system.

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.

5. Experiments with Computer Viruses

To demonstrate the feasibility of viral attack and the degree to which it is a threat, several experiments were performed. In each case, experiments were performed with the knowledge and consent of systems administrators. In the process of performing experiments, implementation flaws were meticulously avoided. It was critical that these experiments not be based on implementation lapses but only on fundamental flaws in security policies.

5.1 The First Virus

On November 3, 1983, the first virus was conceived of as an experiment to be presented at a weekly seminar on computer security. The concept was first introduced in this seminar by the author, and the name 'virus' was thought of by Len Adleman. After eight hours of expert work on a heavily loaded VAX 11/750 system running Unix, the first virus was completed and ready for demonstration. Within a week, permission was obtained to perform experiments, and five experiments were performed. On November 10, the virus was demonstrated to the security seminar.

The initial infection was implanted in 'vd', a program that displays Unix structures graphically, and introduced to users via the system bulletin board. Since vd was a new program on the system, no performance characteristics or other details of its operation were known. The virus was implanted at the beginning of the program so that it was performed before any other processing.

Several precautions were taken in order to keep the attack under control. All infections were performed manually by the attacker and no damage was done, only reporting. Traces were included to assure that the virus would not spread without detection, access controls were used for the infection process, and the code required for the attack was kept in segments, each encrypted and protected to prevent illicit use.

In each of five attacks, all system rights were granted to the attacker in under an hour. The shortest time was under five minutes, and the average under 30 minutes. Even those who knew the attack was taking place were infected. In each case, files were 'disinfected' after experimentation. It was expected that the attack would be successful, but the very short takeover times were quite surprising. In addition, the virus was fast enough (under 1/2 second) that the delay to infected programs went unnoticed.

Once the results of the experiments were announced, administrators decided that no further computer security experiments would be permitted on their system. This ban included the planned addition of traces which could track potential viruses and password augmentation experiments which could potentially have improved security to a great extent. This apparent fear reaction is typical, rather than try to solve technical problems technically inappropriate and inadequate policy solutions are often chosen.

After successful experiments had been performed on a Unix system, it was quite apparent that the same techniques would work on many other systems. In particular, experiments were planned for a Tops-20 system, a VMS system, a VM/370 system, and a network containing several of these systems. In the process of negotiating with administrators, feasibility was demonstrated by developing and testing prototypes. Prototypes attacks for the Tops-20 system were developed by an experienced Tops-20 user in six hours, a novice VM/370 user with the help of an experienced programmer in 30 hours, and a novice VMS user without assistance in 20 hours. These programs demonstrated the ability to find files to be infected, infect them, and cross user boundaries.

After several months of negotiation and administrative changes, it was decided that the experiments would not be permitted. The security officer at the facility was in constant opposition to security experiments, and would not even read any proposals. This is particularly interesting in light of the fact that it was offered to allow systems programmers and security officers to observe and oversee all aspects of all experiments. In addition, systems administrators were unwilling to allow sanitized versions of log tapes to be used to perform offline analysis of the potential threat of viruses, and were unwilling to have additional traces added to their systems by their programmers to help detect viral attacks. Although there is no apparent threat posed by these activities, and they require little time, money, and effort, administrators were unwilling to allow investigations. It appears that their reaction was the same as the fear reaction of the Unix administrators.

5.2 A Bell-LaPadula Based System

In March of 1984, negotiations began over the performance of experiments on a Bell-LaPadula [4] based system implemented on a Univac 1108. The experiment was agreed upon in principal in a matter of hours, but took several months to become solidified. In July of 1984, a two week period was arranged for experimentation. The purpose of this experiment was merely to demonstrate the feasibility of a virus on a Bell-LaPadula based system by implementing a prototype.

Because of the extremely limited time allowed for development (26 hours of computer usage by a user who had never used an 1108, with the assistance of a programmer who had not used an 1108 in five years), many issues were ignored in the implementation. In particular, performance and generality of the attack were completely ignored. As a result, each infection took about 20 seconds, even though they could easily have been done in under a second. Traces of the virus were left on the system although they could have been eliminated to a large degree with little effort. Rather than infecting many files at once, only one file at a time was infected. This allowed the progress of a virus to be demonstrated very clearly without involving a large number of users or programs. As a security precaution, the system was used in a dedicated mode with only a system disk, one terminal, one printer, and accounts dedicated to the experiment.

After 18 hours of connect time, the 1108 virus performed its first infection. After 26 hours of use, the virus was demonstrated to a group of about 10 people including administrators, programmers, and security officers. The virus demonstrated the ability to cross user boundaries and move from a given security level to a higher security level. Again it should be emphasized that no system flaws were involved in this activity, but rather that the Bell-LaPadula model allows this sort of activity to legitimately take place.

The attack was not difficult to perform. The code for the virus consisted of five lines of assembly code, about 200 lines of Fortran code, and about 50 lines of command files. It is estimated that a competent systems programmer could write a much better virus for this system in under two weeks. In addition, once the nature of a viral attack is understood, developing a specific attack is not difficult. Each of the programmers present was convinced that they could have built a better virus in the same amount of time. (This is believable since this attacker had no previous 1108 experience.)

5.3 Instrumentation

In early August of 1984, permission was granted to instrument a VAX Unix system to measure sharing and analyze viral spreading. Data at this time is quite limited, but several trends have appeared. The degree of sharing appears to vary greatly between systems, and many systems may have to be instrumented before these deviations are well understood. A small number of users appear to account for the vast majority of sharing, and a virus could be greatly slowed by protecting them. The protection of a few 'social' individuals might also slow biological diseases. The instrumentation was conservative in the sense that infection could happen without the instrumentation picking it up, so estimated attack times are unrealistically slow.

As a result of the instrumentation of these systems, a set of 'social' users were identified. Several of these surprised the main systems administrator. The number of systems administrators was quite high, and if any of them were infected, the entire system would likely fall within an hour. Some simple procedural changes were suggested to slow this attack by several orders of magnitude without reducing functionality.

Two systems are shown in Fig. 10, with three classes of users (*S* for system, *A* for system administrator, and *U* for normal user). '##' indicates the number of users in each category, 'spread' is the average number of users a virus would spread to, and 'time' is the average time taken to spread to them once they logged in, rounded up to the nearest minute. Average times are misleading because once an infection has reached the 'root' account on Unix, all access is granted. Taking this into account leads to takeover times on the order of one minute which is so fast that infection time becomes a limiting factor in how quickly infections can spread. This coincides with previous experimental results using an actual virus.

System 1

class	#	spread	time
S	3	22	0
A	1	1	0
U	4	5	18

System 2

class	#	spread	time
S	5	160	1
A	7	78	120
U	7	24	600

Fig. 10. Summary of spreading.

Users who were not shared with are ignored in these calculations, but other experiments indicate that any user can get shared with by offering a program on the system bulletin board. Detailed analysis demonstrated that systems administrators tend to try these programs as soon as they are announced. This allows normal users to infect system files within minutes. Administrators used their accounts for running other users' programs and storing commonly executed system files, and several normal users owned very commonly used files. These conditions make viral attack very quick. The use of separate accounts for systems administrators during normal use was immediately suggested, and the systematic movement (after verification) of commonly used programs into the system domain was also considered.

5.4 Summary and Conclusions

The Fig. 11 summarizes the results of these and several other experiments. The systems are across the horizontal axis (Unix, Bell-LaPadula, ...), while the vertical axis indicates the measure of performance (time to program, infection time, number of lines of code, number of experiments performed, minimum time to takeover, average time to takeover, and maximum time to takeover) where time to takeover indicates that all privileges would be granted to the attacker within that delay after introducing the virus.

Viral attacks appear to be easy to develop in a very short time, can be designed to leave few if any traces in most current systems, are effective

	unixC	B-L	Instr	Shell	VMS	Basic	DOS
time	8hrs	18hrs	N/A	15min	30min	2hrs	1hrs
inf t	.5sec	20sec	N/A	2sec	2sec	15sec	10sec
code	200L	260L	N/A	7L	9L	30L	20L
trials	5	N/A	N/A	N/A	N/A	N/A	N/A
min t	5min	N/A	30sec	N/A	N/A	N/A	N/A
avg t	30min	N/A	30min	N/A	N/A	N/A	N/A
max t	60min	N/A	48hrs	N/A	N/A	N/A	N/A

Fig. 11. Experimental results.

against modern security policies for multilevel usage, and require only minimal expertise to implement. Their potential threat is severe, and they can spread very quickly through a computer system. It appears that they can spread through computer networks in the same way as they spread through computers, and thus present a widespread and fairly immediate threat to many current systems.

The problems with policies that prevent controlled security experiments are clear; denying users the ability to continue their work promotes illicit attacks; and if one user can launch an attack without using system bugs or special knowledge, other users will also be able to. By simply telling users not to launch attacks, little is accomplished. Users who can be trusted will not launch attacks but users who would do damage cannot be trusted, so only legitimate work is blocked. The perspective that every attack allowed to take place reduces security is, in the author's opinion, a fallacy. The idea of using attacks to learn of problems is even required by government policies for trusted systems [16,17]. It would be more rational to use open and controlled experiments as a resource to improve security.

6. 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 through any general purpose 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. 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 criteria 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.

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 would likely 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.

References

[1] J.P. Anderson: *Computer Security Technology Planning Study*. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Oct, 1972. Cited in Denning.

[2] Norman T.J. Baily: *The Mathematical Theory of Epidemics*. Hafner Publishing Co., N.Y., 1957.

[3] D.B. Baker: *Department of Defense Trusted Computer System Evaluation Criteria (Final Draft)*. Private communication, The Aerospace Corporation, 1983.

[4] D.E. Bell and L.J. LaPadula: *Secure Computer Systems: Mathematical Foundations and Model*. The Mitre Corporation, 1973. Cited in many papers.

[5] K.J. Biba: *Integrity Considerations for Secure Computer Systems*. USAF Electronic Systems Division, 1977. Cited in Denning.

[6] Richard Dawkins: *The Selfish Gene*. Oxford Press, N.Y., N.Y., 1978.

[7] D.E. Denning: *Cryptography and Data Security*. Addison Wesley, 1982.

[8] A.D. Dewdney: Computer Recreations. *Scientific American* 250(5): 14–22, May, 1984.

[9] R.J. Feiertag and P.G. Neumann: The Foundations of a Provable Secure Operating System (PSOS). In *National Computer Conference*, pages 329–334. AIFIPS, 1979.

[10] J.S. Fenton: *Information Protection Systems*. PhD thesis, U. of Cambridge, 1973. Cited in Denning.

[11] M.R. Garey and D.S. Johnson: *Computers and Intractability*. Freeman, 1979.

[12] B.D. Gold, R.R. Linde, R.J. Peeler, M. Schaefer, J.F. Scheid, and P.D. Ward: A Security Retrofit of VM/370. In *National Computer Conference*, pages 335–344. AIFIPS, 1979.

[13] Gunn, ACM: *Use of Virus Functions to Provide a Virtual APL Interpreter Under User Control*, 1974.

[14] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman: Protection in Operating Systems. In *Proceedings*. ACM, 1976.

[15] L.J. Hoffman: Impacts of information system vulnerabilities on society. In *National Computer Conference*, pages 461–467. AIFIPS, 1982.

[16] U.S. Dept. of Justice, Bureau of Justice Statistics: *Computer Crime – Computer Security Techniques*. U.S. Government Printing Office, Washington, DC, 1982.

[17] M.H. Klein: *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, Fort Meade, Md. 20755, 1983.

[18] B.W. Lampson: A note on the Confinement Problem. In *Communications*. ACM, Oct, 1973.

[19] C.E. Landwehr: The Best Available Technologies for Computer Security. *Computer* 16(7), July, 1983.

[20] R.R. Linde: Operating System Penetration. In *National Computer Conference*, pages 361–368. AIFIPS, 1975.

[21] E.J. McCauley and P.J. Drongowski: KSOS – The Design of a Secure Operating System. In *National Computer Conference*, pages 345–353. AIFIPS, 1979.

[22] G.J. Popek, M. Kampe, C.S. Kline, A. Stoughton, M. Urban, and E.J. Walton: UCLA Secure Unix. In *National Computer Conference*. AIFIPS, 1979.

[23] Schochaud, Hupp, ACM: *The 'Worm' Programs – Early Experience with a Distributed Computation*, 1982.

[24] K. Thompson, ACM: *Reflections on Trusting Trust*, 1984.

[25] J.P.L. Woodward: Applications for Multilevel Secure Operating Systems. In *National Computer Conference*, pages 319–328. AIFIPS, 1979.