COMPUTER VIRUSES

by
Fred Cohen

Table of Contents

Dedication

This work is dedicated to the loving memory of my grandfather Sam Cohen. I hope that when my life ends, I will have lived it so well.

Acknowledgements

It is somewhat strange that the day I first arrived at USC, there was only one professor available to countersign for my courses, and that the same professor, by a very complicated sequence of events, ended up the chairman of my dissertation committee. Dr. Reed has been exceptionally helpful to me in many ways. His advice over the past several years has always proven fruitful, and he never ceases to amaze me with his interest and insights. I am also greatful to Dr. Golomb for his suggestions and assistance in the latter phases of my dissertation. Some of the directions he suggested were quite interesting, and their effect on this work is significant.

I must make special mention of Dr. Adleman's numerous contributions to this work. It was as a direct result of his security class that I stumbled across the rudimentary ideas that inspired this work. He suggested the name "virus" for the class of phenomena which we explore herein, and arranged for permission to perform the first experiments. Another result of his efforts was the initial publication of this work in the popular press, the transitive result of which is still spreading and evolving. His demands for rigor and detail have significantly improved the quality and long term import of this work, and for this I am sincerely greatful.

When you've been a graduate student as long as I have, you end up with a large number of people to thank. Year by year, their names and faces have graced my life, their humor and insanity has filled me with joy and laughter. To those I do not explicitly mention, I sincerely apologize. It is more likely out of forgetfulness than a lack of gratitude. If you don't see your name mentioned, remember my proclivity towards misspelling, and consider that I may have changed some of the names to protect the innocent.

Before thanking those who have helped me, I would like to thank all those who stood in my way. I just want them to know that I haven't forgotten them.

The folks at Harris Plaza have long since given my keys to the rats, but hopefully my fond memories of them will last a lifetime. My tea shirts are already fading from overuse, but my last last tango in Harris is yet to be danced.

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 their help, I have decided to give only first names. Len and David provided a lot of good advice, 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, Howard, 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, but then there are 5 of them.

I would be remiss if I did not express my special thanks to Dr. Irwin Marin. Irwin has been a good friend and intellectual sounding board for me over the last several years, and in many ways was personally responsible for my progress towards the PhD.

Both my immediate and not so immediate family have provided me with support, love, friendship, advice, and untold other types of assistance to my progress in this matter, and without naming them each, I want to thank them. My parents deserve a great deal of thanks for their restraint in not getting involved in the whole affair of getting a PhD. They have managed to stay at an arms length despite the uncontrollable urge that parents have to stay involved with their childrens' lives.

A special debt of gratitude is due to my brother Don and his wife Eve who not only put up with my eccentric habits, but read countless drafts, commented on various aspects of the work, housed and fed me for a significant amount of my time as a gradual student, and blessed my life with another niece, Elisabeth.

Finally, I would like to thank my wife Susan and daughter Carolyn. To quote from an old grade school song: "For understanding and inspiration, to you we sing our praise."

1. Introduction

A "virus" may be loosely defined as a sequence of symbols which, upon interpretation in a given environment, causes other sequences of symbols in that environment to be modified so as to contain (possibly evolved) viruses. If we consider programs as sequences of symbols and computer systems as environments, viruses are programs that may attach themselves to other programs and cause them to become viruses as well. If we consider strands of proteins as sequences of symbols and the biochemistry of cell nuclei as environments, viruses are protein strands that may attach themselves to other protein strands and cause them to become viruses as well. If we consider thought patterns as sequences of symbols and brains as environments, viruses are thought patterns that may attach themselves to other thought patterns and cause them to become viruses as well.

Consider the case where two similar information areas (call them cells), are able to communicate sequences of symbols. If one cell (A) contains a virus (V), and if communication results in the transmission of V to the other cell (B), and if B then interprets V, sequences of symbols stored in B may be modified. If appropriate communication paths are available, a virus may spread from cell to cell. Consider the case where two similar groups of cells (call them organisms), are able to communicate sequences of symbols. If one organism (A) contains a virus (V), and if communication results in the transmission of V to the other organism (B), and if B then interprets V, sequences of symbols stored in cells of B may be modified. If appropriate communication paths are available, a virus may spread from organism to organism. We can extend this sequence of analogous events indefinitely, and thus form a hierarchy of organisms and an associated hierarchy of viral communication paths.

There are many properties of viruses that are interesting at many different levels within many different domains. We will extend our discussion in the domain of computer viruses; viruses within computer systems. In our discussion, we use as general a model of environments and symbol sequences as we reasonably can in the hopes that the extensions to other domains and levels will be straight forward and obvious. The reader who is so inclined, may consider our discussion of computer viruses as merely a vehicle for expressing our understanding in the more general sense.

1.1 Extended Abstract

In this thesis, we open the new topics of viruses and protection from viruses in computer systems. We define a class of computing mechanisms called "viruses",[1] and explore many of their properties, particularly in regard to the threat they pose to the integrity of information in information systems.

The present work concentrates, at the surface level, on integrity problems in computer systems, but strong analogies may be drawn to biological systems and other systems with the information characteristics necessary to support viruses. Where possible, analogies to other systems will be drawn at a philosophical level, but no attempt will be made to demonstrate these analogies with mathematical rigor.

We begin our discussion by briefly reviewing the relevant literature in "computer security", and conclude that no serious previous work has been found in the open literature on the problem of computer viruses. It thus appears that the concept of computer viruses is a novelty in scientific literature at this point, and that little effective protection against viruses is currently available.

We begin the discussion of viruses with an informal discussion based on an English language definition. We give "pseudo-program" examples of viruses as they might appear in modern computer systems, and use these examples to demonstrate some of the potential damage that could result from their use in attacking systems. It is because of this potential damage that we give our examples in pseudo-code rather than an actual computer language for an actual computer system.

---

[1] There are two spellings for the plural of virus; 'virusses', and 'viruses'. We use the one found in Webster's 3rd International Unabridged Dictionary.

We formally define viruses for "Turing machines", and explore some of their properties. We define a Turing machine and a set of (machine,tape-set) pairs which comprise "viral sets" (VS). We show that the union of VSs is also a VS, and that therefor a "largest" VS (LVS) exists for any machine with a viral set. We define a "smallest" VS (SVS), as a VS of which no subset is a VS, and show that for any finite integer "i", there is an SVS with exactly i elements.

We show that any self replicating tape sequence is a one element SVS, that there are countably infinite VSs and non VSs, that machines exist for which all tape sequence are viruses and for which no tape sequences are viruses, and that any finite sequence of tape symbols is a virus with respect to some machine.

We show that determining whether a given (machine,tape-set) pair is a VS is undecidable (by reduction from the halting problem), that it is undecidable whether or not a given "virus" evolves into another virus, that any number that can be "computed" by a TM can be "evolved" by a virus, and that therefor, viruses are at least as powerful as Turing machines as a means for computation.

We then move into a discussion of the relevance of viruses to modern computer protection techniques. We modify the "subject object" protection model [32] to allow computation to be modeled along with protection, by defining a new class of protection machines called "Universal Protection Machines" (UPMs). We show several examples of UPM viruses, and prove that a virus can spread to the transitive closure of information paths from any given source.

The paths of sharing, transitivity of information flow, and generality of information interpretation are identified as the key properties in the spread of computer viruses, and a case by case analysis of these properties is shown. We show that the only systems with potential for limiting viral spreading 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 computer systems. Several protection techniques are explored for their effect on limiting viral spread in computer systems, and some previously unexposed properties of the combination of the "security" and "integrity" models are shown. Difficulties with "imprecise" protection schemes are presented, the most injurious being their tendency to move towards isolationism.

These results are extended to the design of secure computer networks which implement distributed isolationism, and which allow the connection of trusted and untrusted computers to form trusted computer networks. Simple design rules are derived which allow the configuration of secure networks from pictures. Two classes of attacks against these types of computer networks are examined, and an example network is shown under various attack assumptions.

We examine the generalization and combination of security and integrity lattices to partial orderings, and show that a partial ordering is as general a classification scheme as is necessary to model protection in a transitive information network. We extend the previous results to include the effects of modifications of a protection system over time, show techniques for generalized evaluation of the effects of collusions, and demonstrate a method by which a provably correct information management system for automating administration of protection in information networks may be implemented.

We explore viral detection and removal methods which don't depend on the prevention of sharing, limitations on transitivity of information flow, or restricted functionality. Undecidability issues presented earlier are presented in a different form to demonstrate the potential difficulties with detection and cure of computer viruses. Although certain classes of viruses, predominantly those with trivial or simplistic evolutionary characteristics, appear to be defensible through detection and removal, more complex or highly evolutionary viruses appear to present unscalable barriers. The biological analogy to rapidly mutating viruses such as those which comprise the common cold appears to be very strong here.

We examine a complexity based integrity maintenance method with the possibility of detecting corruption through built in self test. A method is shown whereby copyright notices and other aspects of programs and data may be maintained even in a system with no built in defenses. Integrity corruption in such a system is show to be extremely complex, and the technique appears to present a costly but viable defense.

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. Detailed descriptions of experiments are given for three examples, an example of a very short virus for an actual operating system is given, and summary tables are presented.

We explore the use of the results in computer viruses in biological and other domains, and consider the use of the fundamental viral definition as a definition of life. Living systems are considered as a combination of an environment and information within that environment which reproduces and evolves, and several philosophical questions are explored.

It is concluded that 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 perfectly 'safe' policy as of this time is isolationism. Extensions of this work are suggested, and several conjectures are presented.

1.2 Related Work

Given the wide spread use of sharing in current computer systems, the threat of a virus carrying a Trojan horse [1] [41] is significant. Although a considerable amount of work has been done in implementing policies to protect from undesirable dissemination of information [3] [19], and many systems have been implemented to provide protection from this sort of effect [42] [45] [30] [40], little work has been done in the area of keeping information entering an area from causing integrity corruption. [39] [5]

There are many types of information paths possible in computer systems, some legitimate and authorized, and others that may be covert [39], the most commonly ignored one being through the user. We will ignore covert information paths throughout this work, and concentrate only on the effects of viruses as transmitted through the normal authorized information paths available in computer systems.

The general facilities exist for providing provably correct protection schemes [24], but they depend on a consistent and complete security policy that is effective against the types of attacks being carried out. Even some quite simple protection systems cannot be proven safe. [32] Protection from denial of services requires the solution to the halting problem which is well known to be undecidable. [53] The problem of precisely marking information flow within a system has been shown NP-complete. [27] The use of guards for passing untrustworthy information between users has been examined [56], but in general depends on the ability to prove program correctness which is well known to be NP-complete. [28]

The Xerox worm program [50] 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' [21] 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. [31]

The potential threat of a widespread security problem has been examined [33] 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 theoretically sound techniques. [36] Current military protection systems depend to a large degree on isolationism, however new systems are being developed to allow 'multilevel' usage. [37] None of the published proposed systems defines or implements a policy which could completely prevent viral attack.

More detailed literature reviews on particular areas of interest are presented throughout the text as required.

2. Computational Aspects of Computer Viruses

We begin our presentation of the computational aspects of viruses with an informal discussion of viruses within modern computer systems. We then move into more formal definitions using Turing machines [53], and formally show mathematical properties of viruses.

2.1 Informal Discussion

We informally 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 spreads.

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:}
```

Figure 2.1 - 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 here, 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 destructive 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 decreases 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;
        decompress the-rest-of-this-file into tmpfile;
        run tmpfile;}
}
```

Figure 2.2 - A Compression Virus "CV"

This program (C) finds an uninfected executable (E), compresses it, and prepends C to form an infected executable (I). It then decompresses 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. An implementation of this virus has been tested under the UNIX operating system, and is quite slow, predominantly because of the time required for decompression.

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 computer 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 true otherwise false;}
...
```

Figure 2.3 - A Denial of Services Virus

As an analogy to this 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 the people in 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.

A better understanding of the events which might comprise an actual viral attack may be facilitated with the following time line, which shows a simplified scenario of a viral attack on a computer system.

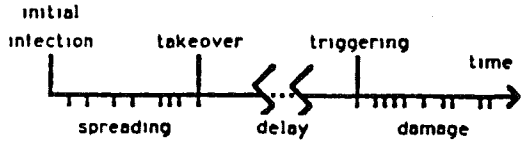


Figure 2.4 - A Scenario of a Viral Attack

A viral attack on a computer system begins with an initial infection. This infection may be created internally or communicated to the system from outside, perhaps as the result of importing infected vendor software.

Once implanted, every time a virus is interpreted, other programs may become infected. Each replication of a virus is

called an infection, and the period over which infection takes place is called the spread time. A typical virus spreads from program to program, and from user to user, eventually embedding its replicants in every program in the system.

Once a virus spreads to the transitive closure of information flow within the system, the infectious period is ended. In most current operating systems, the resulting infection can spread to all programs, so we call the end of the infectious period the takeover time. In systems with special users that have all rights, we consider the system taken over when a special user's program becomes infected.

At this point, an attacker wishing to do severe damage might choose to simply wait. By delaying the damage in a viral attack, an attacker can cause backup tapes to store infected copies of programs, and thus to become of little value once damage is done. A particularly nasty attacker might even infect the backup program and encrypt all information on backup tapes, decrypting information upon retrieval until such time as desired. The period over which an attacking virus waits before performing damage is called the delay time.

The condition used to cause the damaging effects of a virus to begin is called the triggering condition, and the time at which triggering takes place is called the triggering time. Once triggering occurs, every time an infected program is executed, damage is done.

In the case of the encrypting virus mentioned above, the damage might be for each program to enter an infinite loop. Even if we were to restore the backup tapes using a different system, we would find only encrypted information, and thus a great deal of work might be lost.

2.2 Symbols Used in Computability Proofs

Throughout the remainder of this thesis, we will be using logical symbols to define and prove theorems about "viruses" and "machines". We begin by detailing these symbols and their intended interpretation.

We denote sets by enclosing them in curly brackets "{" and "}" [e.g. {a,b}]. We normally use lower case letters [e.g. a,b,...] to denote elements of sets, and upper case letters [e.g. A,B,...] to denote sets themselves. The exception to this rule is the case where sets are elements of other sets, in which case we use the form most convenient for the situation.

The set theory symbols $\in$, $\subset$, $\cup$, and, or, $\forall$, iff, and $\exists$ will be used in their normal manner, and the symbol $\mathbb{N}$ will be used to denote the set of the natural numbers [e.g.{0,1,...}]. The notation {x s.t. P(x)} where P is a predicate will be used to indicate all x s.t. P(x) is true. Square brackets "[" and "]" will be used to group together statements where their grouping is not entirely obvious, and will take the place of normal language parens. The "(" and ")" parens will be used to denote sequences [e.g. (1,2,...)]. The "..." notation will be used to indicate an indefinite number of elements of a set, members of a sequence, or states of a machine wherein the indicated elements are too numerous to fill in or can be generated by some given procedure.

When speaking of sets, we may use the symbol "+" to indicate the union of two sets [e.g. {a}+{b}={a,b}], the symbol $\cup$ to indicate the union of any number of sets, and the symbol "-" to indicate the set which contains all elements of the first set not in the second set [e.g. {a,b}-{a}={b}]. We may also use the "=" sign to indicate set equality. In all other cases, we use these operators in their normal arithmetic sense. The |...| operator will be used to indicate the cardinality of a set or the number of elements in a sequence as appropriate to the situation at hand [e.g. |{a,b,c}|=3, |(a,b,...,f)|=6], and the symbol | when standing alone will indicate the "mod" function [e.g. 12|10=2].

2.3 Computing Machines

We begin our formal discussion with a definition of a computing machine [53] which will serve as our basic computational model for the duration of the discussion. We will be discussing the class of machines which consist of a finite state machine (FSM) with a "tape head" and a semi-infinite tape [see figure below]. The tape head is pointing at one

tape "cell" at any given instant of time, and is capable of reading or writing any of a finite number of symbols from or to the tape, and of moving the tape one cell to the left (-1) or right (+1) on any given "move". The FSM takes input from the tape, sets its next state, and produces output on the tape as functions of its internal state and maps.

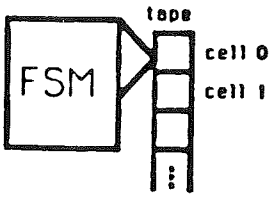


Figure 2.5 - A Computing Machine

A set of Computing Machines "TM" is defined as follows:

$\forall$ M [M $\in$ TM] iff

$$M: (S_M, I_M, O_M: S_M \times I_M \rightarrow I_M, N_M: S_M \times I_M \rightarrow S_M, D_M: S_M \times I_M \rightarrow d)$$

where the state of the FSM is one of n+1 possible states,

$$S_M = \{s_0, \ldots, s_n\} \qquad n \in \mathbb{N}$$

the set of tape symbols is one of j+1 possible symbols, and

$$I_M = \{i_0, \ldots, i_j\} \qquad j \in \mathbb{N}$$

the set of tape motions is one of three possibilities

$$d = \{-1, 0, +1\}.$$

We now define three functions of "time" which describe the behavior of TM programs. Time in our discussion expresses the number of times the TM has performed its basic operation (called a "move" by Turing).

The "state(time)" function is a map from the move number to the state of the machine after that move,

$$\mathcal{S}_M : \mathbb{N} \rightarrow S_M \qquad \text{;state(time)}$$

the "tape-contents(time,cell#)" function is a map from the move number and the cell number on the semi-infinite tape, to the tape symbol on that cell after that move,

$$\square_M : \mathbb{N} \times \mathbb{N} \rightarrow I_M \qquad \text{;tape-contents(time,cell\#)}$$

and the "cell(time)" function is a map from the move number to the number of the cell in front of the tape head after that move.

$$P_M: \mathbb{N} \rightarrow \mathbb{N} \qquad \text{;cell(time)}$$

We call the 3-tuple ($\mathcal{S}_M$, $\square_M$, P_M), the "history" (H_M) of the machine, and the H_M for a particular move number (or instant in time if you prefer) the "situation" at that time. We describe the operation of the machine as a series of "moves" that go from a given situation to the next situation. The initial situation of the machine is described by:

$$(\mathcal{S}_M(0) = \mathcal{S}_{M0},\ \square_M(0,i) = \square_{M0,i},\ P_M(0) = P_0) \qquad i \in \mathbb{N}$$

All subsequent situations of the machine can be determined from the initial situation and the functions "N", "O", and "D" which map the current state of the machine and the symbol in front of the tape head before a move to the "next state", "output", and "tape position" after that move. We show the situation here as a function of time:

$\forall t \in \mathbb{N}$

$$[\mathcal{S}_M(t+1) = N(\mathcal{S}_M(t), \square_M(t, P_M(t)))] \text{ and}$$

$$[\square_M(t+1, P_M(t)) = O(\mathcal{S}_M(t), \square_M(t, P_M(t)))] \text{ and}$$

$$[\forall j \neq P_M(t),\ \square_M(t+1, j) = \square_M(t, j)] \text{ and}$$

$$[P_M(t+1) = \mathrm{Sup}(0, P_M(t) + D(\mathcal{S}_M(t), \square_M(t, P_M(t))))]$$

These machines have no explicit "halt" state which guarantees that from the time such a state is entered, the situation of the machine will never change. We thus define what we mean by "halt" as any situation which does not change with

We will say that "M Halts at time t" iff
[$\forall$ t' > t

[$\mathcal{S}_M(t)=\mathcal{S}_M(t')$] and

[$\forall$ i $\in$ $\mathbb{N}$ [$\square_M(t,i)=\square_M(t',i)$]] and

[$P_M(t)=P_M(t')$]]

and that "M Halts" iff

[$\exists$t $\in$ $\mathbb{N}$ [M Halts at time t]]

We say that "x runs at time t" iff

[[x $\in$ I_M^1 where i $\in$ [$\mathbb{N}$+1]] and

[P(t) = j] and [$\mathcal{S}(t)=\mathcal{S}_0$] and

[($\square$(t,P(t)),....,$\square$(t,P(t)+|x|))=x]]

and that "x runs" iff

[$\exists$ t $\in$ $\mathbb{N}$ [x runs at time t]]

As a matter of convenience, we define two structures which will occur often throughout the rest of the discussion. The first structure "TP" is intended to describe a "Turing machine Program". We may think of such a program as a finite sequence of symbols such that each symbol is a member of the legal tape symbols for the machine under consideration. We define TP as follows:

[$\forall$ M $\in$ TM [$\forall$ v [$\forall$ i $\in$ [$\mathbb{N}$+1]

[v $\in$ TP_M] iff [v $\in$ I_M^1]]]]

The second structure "TS" is intended to describe a non-empty set of Turing machine programs (Turing machine program Set) and is defined as:

[$\forall$ M $\in$ TM [$\forall$ V [V $\in$ TS] iff

i) [$\exists$ v $\in$ V] and

ii) [$\forall$ v $\in$ V [v $\in$ TP_M]]]]

The use of the subscript M (e.g. TP_M) is unnecessary in those cases where only a single machine is under consideration and no ambiguity is present. We will therefor abbreviate throughout this paper by removing the subscript when it is unnecessary.

2.4 Formal Definition of Viruses

We now define the central concept under study, the "viral set". In earlier statements, we informally defined a "virus" as a "program" that modifies other "programs" so as to include a (possibly "evolved") version of itself. In the mathematical embodiment of this definition for TMs, given below, we attempt to maintain the generality of this definition. We note that in the sense of a TM, there is no fundamental difference between data and program. We thus speak only of sequences in our TM discussion.

Several previous attempts at definition have failed because the idea of a singleton "virus" makes the understanding of "evolution" of viruses very difficult, and as we will hopefully make clear, this is a central theme in the results presented herein. The "viral set" embodies evolution by allowing elements of such a set to produce other elements of that set as a result of computation. So long as each "virus" in a "viral set" produces some element of that "viral set" on some part of the tape outside of the original "virus", the set is considered "viral". Thus "evolution" may be described as the production of one element of a "viral set" from another element of that set.

The sequence of tape symbols we call "viruses" is a function of the machine on which they are to be interpreted. In particular, we may expect that a given sequence of symbols may be a "virus" when interpreted by one TM and not a "virus" when interpreted by another TM. Thus, we define the following pair "VS" as follows:

```
[1] ∀ M ∀ V
[2] (M,V) ∈ VS iff
[3]     [V ∈ TS] and [M ∈ TM] and
[4]     [∀v ∈ V [∀H_M
[5]     [∀t ∀j
[6]         [   1) P_M(t)=j and
[7]             2) S_M(t)=S_M0 and
[8]             3) (□_M(t,j),...,□_M(t,j+|v|-1))=v
[9]         ] ⇒
[10]        [   ∃ v' ∈ V [∃t'>t [∃j'
[11]                [   1) [(j'+|v'|)≤j] or [(j+|v|)≤j'] and
[12]                    2) (□_M(t',j'),...,□_M(t',j'+|v'|-1))=v' and
[13]                    3) [∃t'' s.t. [t<t''<t'] and
[14]                            [P_M(t'')∈{j',...,j'+|v'|-1}]]
[15]    ]]] ]       ]
```

We will now review this definition line by line:

```
[1] for all "M" and "V",
[2] the pair (M,V) is a "viral set" if and only if:
[3] V is a non-empty set of TM sequences and M is a TM and
[4] for each virus "v" in V, for all histories of machine M,
[5]     For all times t and cells j
[6]     if      1) the tape head is in front of cell j at time t and
[7]             2) TM is in its initial state at time t and
[8]             3) the tape cells starting at j hold the virus v
[9]     then
[10]          there is a virus v' in V, a time t'>t, and place j'
[11]                    1) at place j' far enough away from v
[12]                    2) the tape cells starting at j' hold virus v'
[13]                    3) and at some time t'' between t and t'
[14]                            v' is written by M
```

For convenience of space, we will use the expression

$$a \overset{B}{\Rightarrow} C$$

to abbreviate part of the previous definition starting at line [4] where a, B, and C are specific instances of v, M, and V respectively as follows:

```
[∀ B [∀ C
      [(M,C) ∈ VS] iff
            [[C ∈ TS] and [M ∈ TM] and
            [∀ a ∈ C [a ⇒(B) C]]]]]
```

Before continuing, we should note some of the features of this definition and their motivation. We define the predicate VS over all Turing Machines. We have also stated our definition so that a given element of a viral set may generate any number of other elements of that set depending on the rest of the tape. This affords additional generality without undue complexity or restriction. Finally, we have no so called "conditional viruses" in that EVERY element of a viral set must ALWAYS generate another element of that set. If a "conditional virus" is desired, we may add conditionals that either cause or prevent a virus from being executed as a function of the rest of the tape, without modifying this definition.

We may also say that V is a "viral set" w.r.t. M

$$\text{iff } [(M,V) \in VS]$$

and define the term "virus" w.r.t. M as

$$\{[v \in V] \text{ s.t. } [(M,V) \in VS]\}$$

We say that "v evolves into v' for M" iff

$$[(M,V) \in VS$$
$$[[v \in V] \text{ and } [v' \in V] \text{ and } [v \overset{M}{\Rightarrow} \{v'\}]]$$

that "v' is evolved from v for M" iff

"v evolves into v' for M"

and that "v' is an evolution of v for M" iff

$$[(M,V) \in VS$$
$$[\exists\, i \in \mathbb{N}\ [\exists\, v' \in V^i$$
$$[v \in V] \text{ and } [v' \in V] \text{ and}$$
$$[\forall\, vk \in V'\ [vk \overset{M}{\Rightarrow} vk+1]] \text{ and}$$
$$[\exists\, 1 \in \mathbb{N}$$
$$[\exists\, m \in \mathbb{N}$$
$$[[1 < m] \text{ and } [v1=v] \text{ and } [vm=v']]]]]]]$$

In other words, the transitive closure of $\overset{M}{\Rightarrow}$ staring from v, contains v'.

2.5 Basic Theorems

At this point, we are ready to begin proving various properties of viral sets. Our most basic theorem states that any union of viral sets is also a viral set:

Theorem 1:

$$\forall\, M\ \forall\, U^*$$
$$[\forall\, V \in U^*\ (M,V) \in VS] \Rightarrow$$
$$[(M, \cup\, U^*) \in VS]$$

Proof:

Define $U = \cup\, U^*$

by definition of $\cup$

1) $[\forall v \in U\ [\exists\, V \in U^* \text{ s.t. } v \in V]]$
2) $[\forall\, V \in U^*\ [\forall\, v \in V\ [v \in U]]]$

Also by definition,

$$[(M,U) \in VS] \text{ iff}$$
$$[[V \in TS] \text{ and } [M \in TM] \text{ and}$$
$$[\forall\, v \in U\ [v \overset{M}{\Rightarrow} U]]]$$

```
by assumption,
        [∀ V ∈ U*
                [∀ v ∈ V [v ⇒M V]]]
thus since
        [∀ v ∈ U [∃ V ∈ U* [v ⇒M V]]]
and     [∀V ∈ U* [V ⊂ U]]

        [∀v ∈U [∃ V ⊂ U [v ⇒M V]]]

hence   [∀ v ∈ U [v ⇒M U]]
thus by definition, (M,U) ∈ VS
Q.E.D.
```

Knowing this, we prove that there is a "largest" viral set with respect to any machine, that set being the union of all viral sets w.r.t. that machine.

```
Lemma 1.1:
[∀ M ∈ TM
        [[∃ V [(M,V) ∈ VS]] ⇒
        [∃ U
                i) [(M,U) ∈ VS] and
                ii) [∀ V [[(M,V) ∈ VS] ⇒
                        [∀ v ∈ V [v ∈ U]]]]]]]

We call U the "largest viral set" (LVS) w.r.t. M, and define
        (M,U) ∈ LVS iff [i and ii]


Proof:
assume  [∃ V [(M,V) ∈ VS]]
choose  U = ∪ {V s.t. [(M,V) ∈ VS]}
now prove i and ii


Proof of i:     (by Theorem 1)
        (M,[∪{V s.t. [(M,V)∈VS]}) ∈VS
thus    (M,U) ∈ VS
```

```
Proof of ii by contradiction:
assume ii) is false:
thus    [∃ V s.t.
            1) [(M,V) ∈ VS] and
            2) [∃ v ∈ V s.t. [v ∉ U]]]

but     [∀ V s.t. (M,V) ∈ VS
            [∀ v ∈ V [v ∈ U]]]      (definition of union)
thus    [v ∉ U] and [v ∈ U]         (contradiction)
thus ii) is true
Q.E.D.
```

Having defined the largest viral set w.r.t. a machine, we would now like to define a "smallest viral set" as a viral set of which no proper subset is a viral set w.r.t. the given machine. There may be many such sets for a given machine.

```
We define SVS as follows:
[∀ M [∀ V
        [(M,V) ∈ SVS] iff
            1) [(M,V) ∈ VS] and
            2) [∄ U s.t
                    [U ⊂ V] (proper subset) and
                    [(M,U) ∈ VS]]]]
```

We now prove that there is a machine for which the SVS is a singleton set, and that the minimal viral set is therefore singleton.

```
Theorem 2:
        [∃ M [∃ V
            i) [(M,V) ∈ SVS] and
            ii) [|V|=1]]]


Proof:  by demonstration
        M: S={s0,s1}, I={0,1},
```

SxI	N	O	D
s0,0	s0	0	0
s0,1	s1	1	+1
s1,0	s0	1	0
s1,1	s1	1	+1

```
        |{(1)}|=1 (by definition of the operator)

        [(M,{(1)}) ∈ SVS] iff
            1) [(M,{(1)}) ∈ VS] and
            2) [(M,{}) ∉ VS]

        (M,{}) ∉ VS (by definition since {} ∉ TS)
```

as can be verified by the reader:

$(1) \overset{M}{\Rightarrow} \{(1)\}$ $(t'=t+2,\ t''=t+1,\ j'=j+1)$

thus $(M,\{(1)\}) \in VS$

Q.E.D.

A simulation of this TM is provided in the appendices to demonstrate that its operation is as claimed.

With the knowledge that the above sequence is a singleton viral set and that it duplicates itself, we suspect that any sequence which duplicates itself is a virus w.r.t. the machine on which it is self duplicating.

Lemma 2.1:

$[\forall\ M \in TM\ [\forall\ u \in TP$

$[[u \overset{M}{\Rightarrow} \{u\}] \Rightarrow [(M,\{u\}) \in VS]]]]$

Proof:

by substitution into the definition of viruses:

$[\forall\ M \in TM\ [\forall\ \{u\}$

$[[(M,\{u\}) \in VS]$ iff

$[[\{u\} \in TS]$ and $[u \overset{M}{\Rightarrow} \{u\}]]]]$

since $[[u \in TP] \Rightarrow [\{u\} \in TS]]$ (definition of TS)

and by assumption,

$[u \overset{M}{\Rightarrow} \{u\}]$

$[(M,\{u\}) \in VS]$

Q.E.D.

The existence of a singleton SVS spurns interest in whether or not there are other sizes of SVSs. We show that for any finite integer i, there is a machine such that there is a viral set with i elements. Thus, SVSs come in all sizes. We prove this fact by demonstrating a machine that generates the "(x mod i) + 1"th element of a viral set from the xth element of that set. In order to guarantee that it is an SVS, we force the machine to halt as soon as the next "evolution" is generated so that no other element of the viral set is generated in the interim. Removing any subset of the viral set guarantees that some element of the resulting set cannot be generated by another element of the set. If we remove all the elements from the set, we have an empty set, which by definition is not a viral set.

Theorem 3:

$[\forall\ i \in [\mathbb{N}+1]$

$[\exists\ M \in TM\ [\exists\ V$

1) $[(M,V) \in SVS]$ and

2) $[|V|=i]]]]$

Proof: By demonstration

M: $S=\{s0,s1,\ldots,si\}$, $I=\{0,1,\ldots,i\}$, $\forall\ x \in \{1,\ldots,i\}$

SxI	N	O	D	
s0,0	s0	0	0	; if I=0, halt
s0,x	sx	x	+1	; if I=x, goto state x, move +1
...				; other states generalized as:
sx,*	sx	[x|i]+1	0	; write [x|i]+1, halt

proof of i)
define $V=\{(1),(2),\ldots,(i)\}$
$|V|=i$ (by definition of operator)

proof of ii)
$[(M,V) \in SVS]$ iff
1) $[(M,V) \in VS]$ and
2) $[\exists U\ [[U \subset V]$ and $[(M,U) \in VS]]]$

proof of "1) $(M,V) \in VS$"
(1) $\overset{M}{\Rightarrow} \{(2)\}$ $(t'=t+2,\ t''=t+1,\ j'=j+1)$
...
$([i-1]) \overset{M}{\Rightarrow} \{(i)\}$ $(t'=t+2,\ t''=t+1,\ j'=j+1)$
(i) $\overset{M}{\Rightarrow} \{(1)\}$ $(t'=t+2,\ t''=t+1,\ j'=j+1)$
and $(1) \in V$, ..., and $(i) \in V$
as can be verified by simulation

thus, $[\forall\ v \in V\ [v \overset{M}{\Rightarrow} V]]$
so $(M,V) \in VS$

proof of "2) $[\exists U\ [[U \subset V]$ and $[(M,U) \in VS]]$"

given $[\exists t,j \in \mathbb{N}\ [\exists\ v \in V$
$[[\square(t,j) = v]$ and
$[\mathcal{S}(t)=\mathcal{S}_0]$ and
$[P(t)=j]]$
$\Rightarrow$
[[M halts at time t+2] and
[v|i]+1 is written at j+1 at t+1]]]
(as may be verified by simulation)

and $[\forall x \in\{1,\ldots,i\}\ [(x) \in V]]$ (by definition of V)
and $[\forall x \in\{1,\ldots,i\}\ [x \overset{M}{\Rightarrow} \{[x|i]+1\}]]$
we conclude that:
[x|i]+1 is the ONLY symbol written outside of (x)

```
thus    [∄ x' ≠ [x|i]+1 [x ⇒M {x'}]]

now     [∀ (x) ∈ V
              [([x|i]+1) ∉ V ⇒ [(x) ∉ V]]]

assume  [∃ U ⊂ V [(M,U) ∈ VS]]
        [U={}] ⇒ [(M,U) ∉ VS] thus U ≠ {}
by definition of proper subset
        [U ⊂ V] ⇒ [∃ v ∈ V [v ∉ U]]


but     [∃ v ∈ V [v ∉ U]]
        ⇒ [∃ v' ∈ U [[v'|i]+1=v]
              and [v ∉ U]
              and [∄ v'' ∈ V [v' ⇒M v'']]]
thus    [∄ v ∈ U [v' ⇒ V]]
        and [v' ∈ U]
thus    [(M,U) ∉ VS] which is a contradiction
Q.E.D.
```

Again, a demonstration of this TM is provided in the appendices for independent verification of its operation.

2.6 Abbreviated Table Theorems

We will now move into a series of proofs that demonstrate the existence of various types of viruses. In order to simplify the presentation, we have adopted the technique of writing "abbreviated tables" in place of complete state tables. The basic principal of the abbreviated table (or macro) is to allow a large set of states, inputs, outputs, next states, and tape movements to be abbreviated in a single statement. We do not wish to give the impression that these macros are anything but abbreviations, and thus we display the means by which our abbreviations can be expanded into state tables. This technique is essentially the same as that used in [53], and we refer the reader to that manuscript for further details on the use of abbreviated tables.

In order to make effective use of macros, we will use a convenient notation for describing large state tables with a small number of symbols. When we define states in these state tables, we will often refer to a state as S_n or S_{n+k} to indicate that the actual state number is not of import, but rather that the given macro can be used at any point in a larger table by simply substituting the actual state numbers for the variable state numbers used in the definition of the macro. For inputs and outputs, where we do not wish to enumerate all possible input and output combinations, we will use variables as well. In many cases, we may describe entire ranges of values with a single variable. We will attempt to make these substitutions clear as we describe the following set of macros.

The "halt" macro allows us to halt the machine in any given state Sn. We use the "*" to indicate that for any input the machine will do the rest of the specified function. The next state entry (N) is Sn so that the next state will always be Sn. The output (O) is * which is intended to indicate that this state will output to the tape whatever was input from the tape. The tape movement (D) is 0 to indicate the tape cell in front of the tape head will not change. The reader may verify that this meets the conditions of a "halt" state as defined earlier.

name	S,I	N	O	D	
halt	Sn,*	Sn	*	0	(halt the machine)

The "right till x" macro describes a machine which increments the tape position (P(t)) until such position is reached that the symbol x is in front of the tape head. At this point, it will cause the next state to be the state after Sn so that it may be followed by other state table entries. Notice the use of "else" to indicate that for all inputs other than x, the machine will output whatever was input (thus leaving the tape unchanged) and move to the right one square.

```
name     S,I      N        O        D
-----------------------------------
R(x)     Sn,x     Sn+1     x        0       (right till x)
         Sn,else  Sn       else     +1
```

The "left till x" macro is just like the R(x) macro except that the tape is moved left (-1) rather than right (+1).

```
name     S,I      N        O        D
-----------------------------------
L(x)     Sn,x     Sn+1     x        0       (left till x)
         Sn,else  Sn       else     -1
```

The "change x to y until z" macro moves from left to right over the tape until the symbol z is in front of the tape head, replacing every occurrence of x with y, and leaving all other tape symbols as they were.

```
name     S,I      N        O        D
-----------------------------------
C(x,y,z)Sn,z      Sn+1     z        0       (change X to Y till Z)
         Sn,x     Sn       y        +1
         Sn,else  Sn       else     +1
```

The above macros are demonstrated in the appendices in a sample program to demonstrate that they do indeed perform as described.

The "copy from x till y to after z" macro is a bit more complex than the previous macros because its size depends on the number of input symbols for the machine under consideration. The basic principal is to define a set of states for each symbol of interest so that that set of states replaces the symbol of interest with the "left of tape marker", moves right until the "current right of tape marker", replaces that marker with the desired symbol, moves right one more, places the marker at the "new right of tape", and then moves left till the "left of tape marker", replaces it with the original symbol, moves right one tape square, and continues from there. The loop just described requires some initialization to arrange for the "right of tape marker" and a test to detect the y on the tape and thus determine when to complete its operation. At completion, the macro goes onto the state following the last state taken up by the macro, and it can thus be used as the above macros.

```
name     S,I      N        O        D
-----------------------------------
CPY(X,Y,Z)                (copy from X till Y to after Z)
         Sn       R(X)                       ;right till X
         Sn+1     Sn+2     "N"      0        ;write "N"
         Sn+2     R(Y)                       ;right till Y
         Sn+3     R(Z)                       ;right till Z
         Sn+4     Sn+5     Z        +1       ;right one more
         Sn+5     Sn+6     "M"      0        ;write "M"
         Sn+6     L("N")                     ;left till "N"
         Sn+7     Sn+8     X        0        ;replace the initial X
         Sn+8,Y   Sn+9     Y        +1       ;if Y, done
         Sn+8,*   Sk+5*    "N"      +1       ;else write "N" an
                                             ;goto sn+5 times input
                                             ;symbol number
         Sn+9     R(M)                       ;right till "M"
         Sn+10    Sn+11    Y        0        ;copy completed
         Sk+5*    R("M")                     ;goto the "M"
         Sk+5*+1  Sk+5*+2  *        +1       ;write the copied symbol
         Sk+5*+2  Sk+5*+3  "M"      0        ;write the trailing "M"
         Sk+5*+3  L("N")                     ;left till "N"
         Sk+5*+4  Sn+8     *        +1       ;rewrite * and go on
```

As a note, we should observe that for each of the above macros (except "halt"), the "arguments" must be specified ahead of time, and if the tape is not in such a configuration that all of the required symbols are present in their proper order, the macros may cause the machine to loop indefinitely in the macro rather than leaving upon completion.

We now show that there is a viral set which is the size of the natural numbers (countably infinite), by demonstrating a viral set of which each element generates an element with one additional symbol. Since, given any element of the set, a new element is generated with every execution, and no previously generated element is ever regenerated, we have a set generated in the same inductive manner as the natural numbers, and there is thus a one to one mapping to the natural numbers from the generated set.

Theorem 4:

[∃ M ∈ TM ∃ V ∈ TS s.t.
1) [(M,V) ∈ VS] and
2) [|V|=|ℕ|]

Proof by demonstration:

```
                S,I     N       O       D
                ----------------------------
     M:         S0,L    S1      L       +1      ;start with L
                S0,else S0      X       0       ;or halt
                S1,0    C(0,X,R)                ;change 0s to Xs till R
                S2,R    S3      R       +1      ;write R
                S3      S4      L       +1      ;write L
                S4      S5      X       0       ;write X
                S5      L(R)                    ;move left till R
                S6      L(X or L)               ;move left till X or L
                S7,L    S11     L       0       ;if L goto s11
                S7,X    S8      0       +1      ;if X replace with 0
                S8      R(X)                    ;move right till X
                S9,X    S10     0       +1      ;change to 0, move right
                S10     S6      X       0       ;write X and goto S6
                S11     R(X)                    ;move right till X
                S12     S13     0       +1      ;add one 0
                S13     S13     R       0       ;halt with R on tape
```

V={(L0R),(L00R),...,(L0...0R),...}

proof of 1) (M,V) ∈ VS

definition:

[∀ M ∈ TM [∀ V
[(M,V) ∈ VS] iff
[[V ∈ TS] and [∀ v ∈ V [v $\overset{M}{\Rightarrow}$ V]]]]]]

by inspection,
[V ∈ TS]

now [∀ (L0...0R) [∃ (L0...00R) ∈ V
[(L0...0R) $\overset{M}{\Rightarrow}$ {(L0...00R)}]]]
(may be verified by simulation)

thus [(M,V) ∈ VS]

proof of 2) $|V|=|\mathbb{N}|$

$[\forall\ v_n \in V\ [\exists\ v_{n+1} \in V$
$\quad [\forall\ k \leq n$
$\quad\quad [\exists\ v_k \in V\ [v_k=v_{n+1}]]]]$

this is the same form as the definition of $\mathbb{N}$, hence $|V|=|\mathbb{N}|$
Q.E.D

This program is also demonstrated in the appendices to demonstrate its operation and correctness.

As a side issue, we show the same machine has a countably infinite number of sequences that are not viral sequences, thus proving that no finite state machine can be given to determine whether or not a given (M,V) pair is "viral" by simply enumerating all viruses (from Thm 4) or by simply enumerating all non viruses (by Lem 4.1).

Lemma 4.1:

$[\exists\ M \in TM\ [\exists\ W \in TS$
$\quad 1)\ [|W|=|\mathbb{N}|]$ and
$\quad 2)\ [\forall\ w \in W\ [\exists\ W' \subset W$
$\quad\quad [w \overset{M}{\Rightarrow} W']]]]]$

Proof:
using M from Theorem 4, we choose
$W=\{(X),(XX),\ldots,(X\ldots X),\ldots\}$
clearly $[M \in TM]$ and $[W \in TS]$ and $[|W|=|\mathbb{N}|]$
since (from the state table)
$[\forall\ w \in W\ [w$ runs at time $t] \Rightarrow [w$ halts at time $t]]$
$[\exists\ t'>t\ [P_M(t') \neq P_M(t)]]$
thus $[\forall\ w \in W\ [\exists\ W' \subset W\ [w \overset{M}{\Rightarrow} W']]]$
Q.E.D.

It turns out that the above case is an example of a viral set that has no SVS. This is because no matter how many elements of V are removed from the front of V, the set can always have another element removed without making it nonviral.

We also wish to show that there are machines for which no sequences are viruses, and do this trivially below by defining a machine which always halts without moving the tape head.

Lemma 4.2:

$[\exists\ M \in TM\ [\exists\ V \in TS\ [(M,V) \in VS]]]$

Proof by demonstration:

	S,I	M	0	D
M:	s0,all	s0	0	0

(trivially verified that $[\forall\ t\ [P_M(t)=P_0]]$)
Q.E.D.

We now show that for ANY finite sequence of tape symbols "v", it is possible to construct a machine for which that

sequence is a virus. As a side issue, this particular machine is such that LVS=SVS, and thus no sequence other than "v" is a virus w.r.t. this machine. We form this machine by generating a finite "recognizer" that examines successive cells of the tape, and halts unless each cell in order is the appropriate element of v. If each cell is appropriate we replicate v and subsequently halt.

Theorem 5:

$[\forall\ v \in TP\ [\exists\ M \in TM\ [(M,\{v\}) \in VS]]]$

Proof by demonstration:

$v=\{v0,v2,\ldots,vk\}$ where $[k \in \mathbb{N}]$ and $[v \in I^i]$

(definition of TP)

	S,I	N	O	D	
M:	s0,v0	s1	v0	+1	(recognize 1st element of v)
	s0,else	s0	0	0	(or halt)
	...				(etc till)
	sk,vk	sk+1	vk	+1	(recognize kth element of v)
	sk,else	s0	0	0	(or halt)
	sk+1	sk+2	v0	+1	(output 1st element of v)
	...				(etc till)
	sk+k	sk+k	vk	+0	(output kth element of v)

it is trivially verified that $[v \overset{M}{\Rightarrow} \{v\}]$
and hence (by Lemma 2.1) $[(M,\{v\}) \in VS]$
Q.E.D.

With this knowledge, we can easily generate a machine which recognizes any of a finite number of finite sequences and generates either a copy of that sequence (if we wish each to be an SVS), another element of that set (if we wish to have a complex dependency between subsequent viruses), a given sequence in that set (if we wish to have only one SVS), or each of the elements of that set in sequence (if we wish to have LVS=SVS).

We will again define a set of macros to simplify our task. This time, our macros will be the "recognize" macro, the "generate" macro, the "if-then-else" macro, and the "pair" macro.

The "recognize" macro simply recognizes a finite sequence and leaves the machine in one of two states depending on the result of recognition. It leaves the tape at its initial point if the sequence is not recognized so that successive recognize macros may be used to recognize any of a set of sequences starting at a given place on the tape without additional difficulties. It leaves the tape at the cell one past the end of the sequence if recognition succeeds, so that another sequence can be added outside of the recognized sequence without additional difficulty.

S,I	N	O	D	
recognize(v) for v of size z				
sn,v0	sn+1	v0	+1	(recognize 0th element)
sn,*	sn+z+z-1	*	0	(or rewind 0)
...			(etc till)	
sn+k,vk	sn+k+1	vk	+1	(recognize kth element)
sn+k,*	sn+z+z-k	*	-1	(or rewind tape)
...			(etc till)	
Sn+z-1,vz	Sn+z+z	vz	+1	(recognize the last one)
Sn+z-1,*	Sn+z	vz	+1	(or rewind tape)
Sn+z,*	Sn+z+1	*	-1	(rewind tape one square)
...				(for each of k states)
Sn+z+z-1		("didn't recognize" state)		
Sn+z+z		("did recognize" state)		

The "generate" macro simply generates a given sequence starting at the current tape location:

```
        S,I     N       O       D
        ------------------------
generate(v) where v is of length k
        Sn      Sn+1    v0      +1
        ...
        Sn+k    Sn+k+1  vk      +0
```

The "if-then-else" macro consists of a "recognize" macro on a given sequence, and goes to a next state corresponding to the initial state of the "then" result if the recognize macro succeeds, and to the next state corresponding to the initial state of the "else" result if the recognize macro fails:

```
        S,I             N               O       D
        ------------------------
if (v) (then-state) else (else-state)
        Sn      recognize(v)
        Sn+2|v|-1,*     else-state      *       0
        Sn+2|v|,*       then-state      *       0
```

The "pair" macro simply appends one sequence of states to another, and thus forms a combination of two sequences into a single sequence. The resulting state table is just the concatenation of the state tables:

```
        S,I     N       O       D
        ------------------------
pair(a,b)
        Sn      a
        Sm      b
```

We may now write the previous machine "M" as:

```
if (v) (pair(generate(v),halt)) else (halt)
```

We can also form a machine which recognizes any of a finite number of sequences and generates copies,

```
if (v0) (pair(generate(v0),halt)) else
        if (v1) (pair(generate(v1),halt)) else
                ...
                if (vk) (pair(generate(vk),halt)) else (halt)
```

a machine which generates the "next" virus in a finite "ring" of viruses from the "previous" virus,

```
if (v0) (pair(generate(v1),halt)) else
        if (v1) (pair(generate(v2),halt)) else
                ...
                if (vk) (pair(generate(v0),halt)) else (halt)
```

and a machine which generates any desired dependency.

```
if (v0) (pair(generate(vx),halt)) else
        if (v1) (pair(generate(vy),halt)) else
                ...
                if (vk) (pair(generate(vz),halt)) else (halt)
where vx, vy, ...,vz ∈ {v1,...,vk}
```

We provide a demonstration of a simple "recognize generate" virus of the above sort in the appendices.

We now show a machine for which every sequence is a virus, as is shown in the following simple lemma.

Lemma 6.1:

```
    [∃ M ∈ TM
            [∀ v ∈ TP [∃ V
                    [[v ∈ V] and [(M,V) ∈ LVS]]]]]
```

Proof by demonstration:

```
        I={X}, S={s0}
        S,I     N       O       D
M:      s0,X    s0      X       +1
```

trivially seen from state table:
[∀ time t [∀ S [∀P [not M halts]]]]
and [∀ n ∈ ℕ [∀ v ∈ I^n
[[v $\overset{M}{\Rightarrow}$ {(X)}] and [(M,{(X),v}) ∈ LVS]]]]
hence [∀ v ∈ TP [(M,{v,(X)}) ∈ VS]]
and by Theorem 1, [∃ V [[v ∈ V] and [(M,V) ∈ LVS]]]
Q.E.D.

2.7 Computability Aspects of Viruses and Viral Detection

We can clearly generate a wide variety of viral sets, and the use of macros is quite helpful in pointing this out. Rather than follow this line through the enumeration of any number of other examples of viral sets, we would like to determine the power of viruses in a more general manner. In particular, we will explore three issues.

The "decidability" issue addresses the question of whether or not we can write a TM program capable of determining, in a finite time, whether or not a given sequence for a given TM is a virus. The "evolution" issue addresses the question of whether we can write a TM program capable of determining, in a finite time, whether or not a given sequence for a given TM "generates" another given sequence for that machine. The "computability" issue addresses the question of determining the class of sequences that can be "evolved" by viruses.

We now show that it is undecidable whether or not a given (M,V) pair is a viral set. This is done by reduction from the halting problem in the following manner. We take an arbitrary machine M' and tape sequence V', and generate a machine M and tape sequence V such that M copies V' from inside of V, simulates the execution of M' on V', and if V' halts on M', replicates V. Thus, V replicates itself if and only if V' would halt on machine M'. We know that the "halting problem" is undecidable [53], that any program that replicates itself is a virus [Lemma 2.1], and thus that [(M,V) ∈ VS] is undecidable.

Theorem 6:
[∄ D ∈ TM [∃ s1 ∈ S_D
[∀ M ∈ TM [∀ V ∈ TS
1) [D halts] and
2) [$S_D(t)$ = s1] iff [(M,V) ∈ VS]]]]]

Proof by reduction from the Halting Problem:
[∀ M ∈ TM [∃ M' ∈ TM
["L" ∉ $I_{M'}$] and ["R" ∉ $I_{M'}$] and
["l" ∉ $I_{M'}$] and ["r" ∉ $I_{M'}$] and
[∀ $S_{M'}$ [$I_{M'}$ = "r"] ⇒
[[$N_{M'}$=$S_{M'}$] and [$O_{M'}$="r"] and [$D_{M'}$=+1]]]
and [∀S_M
[[N_M=S_M] and [O_M=I_M] and [D_M=0]]
⇒ [[$N_{M'}$=S_x] and [$O_{M'}$=I_M] and $D_{M'}$=0]]]
]]

We must take some care in defining the machine M' to assure that it CANNOT write a viral sequence, and that it CANNOT overwrite the critical portion of V which will cause V to replicate if M' halts. Thus, we restrict the "simulated" (M',V') pair by requiring that the symbols L,R,l,r not be used by them. This restriction is without loss of generality, since we can systematically replace any occurrences of these symbols in M' without changing the computation performed or its halting characteristics. We have again taken special care to assure that (M',V') cannot interfere with the sequence V by restricting M' so that in ANY state, if the symbol "l" is encountered, the state remains unchanged, and the tape moves

right by one square. This effectively simulates the "semi-infinite" end of the tape, and forces M' to remain in an area outside of V. Finally, we have restricted M' such that for all states such that "M halts", M' goes to state S_x.

now by [63]
[∃ D ∈ TM
[∀ M' ∈ TM [∀ V' ∈ TS
1) [D halts] and
2) [$S_D(t)$ = s1] iff [(M',V') halts]]]]

We now construct (M,V) s.t.
[(M,V) ∈ VS] iff [(M',V') Halts]
as follows:

```
            S,I     N       O       D
            ----------------------------
     M:     s0,L    S1      L       0       ;if "L" then continue
            s0,else S0      X       0       ;else halt
            s1      CPY("1","r","R")        ;Copy from 1 till r after R
            s2      L("L")                  ;left till "L"
            s3      R("R")                  ;right till "R"
            s4      s5      1       +1      ;move to start of (M',V')
            s5      M'                      ;the program M' goes here
            sx      L("L")                  ;move left till "L"
            sx+1    CPY("L","R","R")        ;Copy from L till R after R
```

V={(L,1,v',r,R)}

Since the machine M requires the symbol "L" to be under the tape head in state s0 in order for any program to not halt immediately upon execution, and since we have restricted the simulation of M' to not allow the symbol "L" to be written or contained in v', M' CANNOT generate a virus.

∀ t ∈ ℕ [∀ $S_M \leq$ sx
[∃ $P_M(t)$ [[I ≠ "L"] and [O="L"]]]]]

This restricts the ability to generate members of VS such that V only produces symbols containing the symbol "L" in state s0 and sx+1, and thus these are the ONLY states in which replication can take place. Since s0 can only write 'L' if it is already present, it cannot be used to write a virus that was not previously present.

[∀ t ∈ ℕ [∀s (s5 ≤ s ≤ sx)
[not [M' halts at time t]] and [P_M(t+1) not within V]]]

If the execution of M' on V' never halts, then sx+1 is never reached, and thus (M,V) can not be a virus.

[∀ Z ∈ TP s.t. Z_0 ≠ "L"]
[M run on Z at time t] ⇒ [M halts at time t+1]
[(M',V') Halts] iff
[∃ t ∈ ℕ s.t. S_t=sx+1]
thus [not (M',V') Halts] ⇒ [(M,V) ∉ VS]

Since sx+1 replicates v after the final "R" in v, M' halts ⇒ that V is a viral set w.r.t. M

[∃ t ∈ ℕ s.t. S_t=sx+1] ⇒
[∀v ∈ V s.t. [v $\overset{M}{\Rightarrow}$ {V}]]
and from Lemma 2.1
[∀ v ∈ V v $\overset{M}{\Rightarrow}$ V] ⇒ [(M,V) ∈ VS]
thus [(M,V) ∈ VS] iff [(M',V') Halts]

```
and by [53]
[∄ D ∈ TM
        [∀ M' ∈ TM [∀ V' ∈ TS
        1) [D halts] and
        2) [S_D(t) = s1] iff [(M',V') halts]]]]
```

```
thus
[∄ D ∈ TM
        [∀ M ∈ TM [∀ V ∈ TS
        1) [D halts] and
        2) [S_D(t) = s1] iff [(M,V) ∈ VS]]]]
Q.E.D.
```

We now answer the question of viral "evolution" quite easily by changing the above example so that it replicates (state 0') before running V' on M', and generates v' iff (M',V') halts. The initial self replication forces [(M,V)∈VS], while the generation of v' iff (M',V') halts, makes the question of whether v' can be "evolved" from v undecidable. v' can be any desired sequence a, and if it is a virus and not v, it is an evolution of v iff (M',V') halts. As an example, v' could be v with a slightly different sequence V" in place of V'.

```
Lemma 6.1:
[∄ D ∈ TM
        [∀ (M,V) ∈ VS
                [∀v ∈ V [∀v'
                        1) [D halts] and
                        2) [S(t) = s1] iff [v ⇒(M) {v'}]]]]]
```

```
sketch of proof by demonstration:
modify machine M above s.t.:
```

```
M:      s0,L    S0'     L       0       ;if "L" then continue
        s0,else S0      X       0       ;else halt
        s0'     CPY("L","R","R")        ;replicate initial virus
        s0''    L("L")                  ;return to replicated "L"
        s1      CPY("l","r","R")        ;Copy from l till r after R
        s2      L("L")                  ;left till "L"
        s3      R("r")                  ;right till "R"
        s4      s5      r       +1      ;move to start of (M',V')
        s5      M'                      ;the program M' goes here
        sx      L("L")                  ;move left till "L"
        sx+1    R("R")                  ;move right till "R"
        sx+2    s0+k    "R"     +1      ;get into available space
        sx+3    generate(v')            ;and generate v'
```

```
assume  [v' is a virus w.r.t. M]
since   [sx+3 is reached] iff [(M',V') halts]
thus    [v' is generated] iff [(M',V') halts]
Q.E.D.
```

We are now ready to determine just how powerful viral evolution is as a means of computation. Since we have shown that an arbitrary machine can be embedded within a virus (Theorem 6), we will now choose a particular class of machines to embed to get a class of viruses with the property that the successive members of the viral set generated from any

particular member of the set, contain subsequences which are (in Turing's notation) the of successive iterations of the "Universal Computing Machine." [53] The successive members are called "evolutions" of the previous members, and thus any number that can be "computed" by a TM, can be "evolved" by a virus. We therefore conclude that "viruses" are at least as powerful a class of computing machines as TMs, and that there is a "Universal Viral Machine" which can evolve any "computable" number.

Theorem 7:

$$[\forall M' \in TM\ [\exists (M,V) \in VS\ [\forall i \in \mathbb{N}\ [\forall x \in \{0,1\}^i\ [x \in H_{M'}]\ [\exists v \in V\ [\exists v' \in V\ [[v \text{ "evolves" into } v'] \text{ and } [x \subset v']]$$

Proof by demonstration:

by [53]:

$$[\forall M' \in TM\ [\exists UTM \in TM\ [\exists \text{"D.N"} \in TS\ [\forall i \in \mathbb{N}\ [\forall x \in \{0,1\}^i\ [x \in H_{M'}]$$

Using the original description of the "Universal Computing Machine" [53], we modify the UTM so that each successive iteration of the UTM interpretation of an "D.N" is done with a new copy of the "D.N" which is created by replicating the modified version resulting from the previous iteration into an area of the tape beyond that used by the previous iteration. We will not write down the entire description of the UTM, but rather just the relevant portions.

```
SxI     N       0       D
-----------------------
b:      f(b1,b1,"::")           ;initial states of UTM print out
b1:     R,R,P:,R,R,PD,R,R,PA anf;:DA on the f-squares after ::
anf:                            ;this is where UTM loops
...                             ;the interpretation states follow
ov:     anf                     ;and tha machine loops to anf
```

We modify the machine as in the case of Theorem 6 except that:

```
we replace:
        ov:     anf             ;goto "anf"
with:   ov:     g(ov',"r")      ;write an "r"
        ov':    L("L")          ;go left till "L"
        ov'':   CPY("L","R","R");replicate virus
        ov''':  L("L")          ;left till start of the evolution
        ov'''': R("r")          ;right till marked "r"
        ov''''':anf             ;goto "anf"
```

and $[\forall S_{UTM}\ [I_{UTM}\text{"R"}] \Rightarrow$

<move right 1, write "R", move left 1, continue as before>

The modification of the "anf" state breaks the normal interpretation loop of the UTM, and replaces it with a replication into which we then position the tape head so that upon return to "anf" the machine will operate as before over a different portion of the tape. The second modification assures that from any state that reaches the right end of the virus "R", the R will be moved right one tape square, the tape will be repositioned as it was before this movement, and the operation will proceed as before. Thus, tape expansion does not eliminate the right side marker of the virus. We now specify a class of viruses as:

("L","D.N","R")

and M as:

```
SxI      N       O       D
-------------------------
s0,L     s1      L       +1      ;start with "L"
s0,else  s0      else    0       ;or halt
s1 ...                           ;states from modified UTM
```

3. The Modified Subject Object Model

We now examine computer viruses in terms of the subject object protection model. [32] We define a "universal protection machine" (UPM) which generalizes the subject object model by combining it with the Turing machine definition. [53] The resultant structure appears to be a good model of a computer with an operating system. We then show that a virus can infect an object *e* if some subject can both read an infected object *i* and write *e*. We show that the transitivity property holds for infection, and that a virus can therefor spread to the transitive closure of information paths from an initial source. We discuss an extension of the UPM to model computer networks, and comment further on the model.

3.1 A Protection Model

A *protection system* is defined in terms of the rights of subjects to objects. [32] We are primarily concerned here with the "read" and "write" rights *r* and *w*, in a static *configuration* of a protection system. A *protection system* is defined by a triple (S, O, P) where; S is a set of *subjects*; O is a set of *objects*; and P is an *access matrix*, with a row for every subject in S, and a column for every object in O.

It is common in modern computer systems to have a set of "users" with access to a set of "files", and the subjects and objects in this model may be thought of as corresponding respectively to users and files, with access rights being "read" and "write". In general, the model is not limited to this view. Another perspective might be that each "subject" is a robot, and each "object" is a physical world object, with access rights being the ability of robots to touch, move, tool, and restrict access to objects.

	o0	o1	o2	o3
s0	rw	r-	--	-w
s1	r-	rw	rw	r-

Figure 3.1 - An Access Matrix

The above example of an access matrix shows a protection system with two subjects (s0 and s1), and four objects (o0, o1, o2, o3). Each element of the access matrix contains an 'r' if the corresponding subject can read the corresponding object, and a 'w' if the corresponding subject can write the corresponding object. Thus, subject s0 can read objects o0, and o1, and can write o0 and o3; while s1 can read o0, o1, o2, and o3, and write o1 and o2.

In our analysis, we will assume that all objects are finite sequences of symbols representing either the D.N of a UTM program [53], or data for interpretation by such a program, and that two rights are of primary interest; the generic *read* right which enables a subject to examine the symbol sequence of an object; and the generic *write* right which enables a subject to set the symbol sequence of an object.

Although we will be primarily discussing the case where the access matrix is in a static configuration, dynamic configurations are also of considerable interest. We note that in Harrison, Ruzzo and Ullman [32], it has been proven that "It is undecidable whether a given configuration of a given protection system is 'safe' for a generic right", where safety implies that no right to an object can be "leaked" to a subject without the permission of the "owner" of that object.

3.2 A Universal Protection Machine

In order to model the mutual effects of computation and protection, we specify a model which allows the features of the Turing machine to be combined with the features of a protection system. We specify a "Universal Protection Machine" (UPM) wherein any finite number of subjects and objects may coexist. The UPM simulates the interpretation of objects by subjects and uses some decidable scheduling algorithm to determine which subject is simulated on each successive

The UPM maintains a subject object matrix, the current sequences representing all objects, the sequence of objects remaining to be interpreted by each user, current tape sequences, states, and tape positions of each sequence under interpretation; and mediates the rights of subjects to objects, the scheduling process which determines after each subject's move which subject is allocated the next move, and the effects of subjects and objects on each other.

We show here the manner in which information may be stored in such a machine so that an appropriate TM would be able to perform all necessary operations using finite time and space. We then describe procedures which a UPM might use in performing the required operations. We note that in order to strictly prove that such a machine is possible, we would have to construct a state table which would actually carry out these operations, or prove that such a state table exists. Although this would likely be of some interest, the space that a formal proof would require would be quite more than we wish to dedicate to this problem. We will instead, make an informal but accurate case for the existence of such a state table, and move on to the ramifications of the existence of such a machine.

We begin by specifying the sequence stored on the semi-infinite tape of the UPM. The UPM maintains information in much the same manner as a Universal Computing Machine [53], wherein a finite set of special purpose symbols are used to preface each type of information. We first give a generic description of a UPM tape contents, and then detail the symbols used in the description.

The tape consists of eight distinct sections, all but the last consisting of a finite number of symbols, and each representing a different aspect of the UPM. These sections are as follows:

```
The left of the tape
The Subject/Object Matrix
The remaining objects to be "run" by each subject
The sequences representing the current objects
The current tape sequences and markings under interpretation
The temporary use area
The right of tape
The rest of the tape
```

As in the Universal Computing Machine, we will use every other square for the storage of most of the information of use to us, and use the intervening squares for the operation of the machine itself. We now specify each of the above listed sections of the tape in further detail.

The left of tape is signified by the symbol "L":

```
L       left of tape
```

The Subject/Object matrix is bracketed by "S/O" and "O/S", with each row of the matrix representing a given subject initiated by "S" followed by the appropriate number of s's to indicate the subject number. Within each row, each column indicating a given object is indicated by an "O" followed by an appropriate number of o's to indicate the object number. Within each subject object pair, each generic right is indicated by an "R" followed by an appropriate number of r's indicating a given right number.

```
S/O     subject/object matrix
S       the start of a subject
ss...s  the subject number indicated by the number of s's
O       the start of an object
oo...o  the object number indicated by the number of o's
R       the generic right
rr...r  the right number indicated by the number of r's
...
R
rr..r   as many rights as needed
O
oo..o   the next object
...
O
oo.o    the last object for that subject
S
ss..s   the next subject
...
O/S     the end of the subject object matrix
```

The sequence of object numbers of objects awaiting interpretation for each subject are maintained in the "run list" which is bracketed on the left by "R/L" and on the right by "L/R". Each subject with objects awaiting interpretation is indicated by an entry "S" followed by an appropriate number of s's to indicate the subject number. Each object awaiting interpretation by that subject is indicated following the subject indicator by an "O" followed by an appropriate number of o's. We note that each subject may only have a finite sequence of objects in its run list.

```
R/L     The start of the run list
S       A new subject
ss...s  The subject number
O       The next object to be interpreted
oo...o  The object number
...
O       The last object to be interpreted for that subject
oo..o   Its object number
S       The next subject
ss..s   The subject number
...     etc.
L/R     The end of the run list
```

Each of the current objects is itself the D.N of a Universal Computing Machine tape, and as such is described in the same manner as tapes are described in Turing's original paper [53] and we will not describe them further here. Each D.N is denoted by the object number, and the set of objects are bracketed by "B/O" and "O/B":

```
B/O     Beginning of objects
O       Object start
oe...o  Object number
D.N     D.N of object
...
O       Last object start
oe..e   Object number
D.N     D.N of object
O/B     End of objects
```

Each sequence interpretable at any given instant (a "process" in descriptions of operating systems), has a representative tape sequence which is generated by the sequence of the object being interpreted at the initial invocation of interpretation, the moves which have been made in that interpretation by the UPM, and any effects of read or written sequences. The state of a process at any given instant is completely described by the D.N and markings of that process as it appears on the tape at the end of its last move. [53] The set of D.Ns currently being interpreted are bracketed by "C/P" and "P/C", and each sequence is prefaced by an "S" followed by an appropriate number of s's to indicate the subject number for which that D.N is operating. We note that since the D.N and marking include the marking of the current state of the program and the current position of the tape head within that program, these need not be stored independently.

```
C/P     Current sequences beginning
S       Start of a subject
ss...s  Subject number
D.N+M   D.N and Marking of a tape sequence
...
S       Start of a last subject
ss..s   Subject number
D.N+M   D.N and Marking of a tape sequence
P/C     End of current sequences
```

The temporary use area is used by the UPM to store the sequence being interpreted at any given instant, and for other temporary use as required, and may contain any required sequence. The right of tape is used to keep track of the right most place on the tape at any given moment, and is denoted by the symbol "R".

```
R       The right of the tape
```

We note that for finite subjects, objects, and other sequences, the tape contents are finite, and are representable in a finite number of symbols, and that we can thus place this information on the tape of a TM.

3.3 Operation of the Universal Protection Machine

We now briefly summarize the operation of the UPM by description without formally specifying its operation. Perhaps the most important aspect of our description is that all operations and information stored as a result of these operations are finite, and can thus be performed in a finite number of moves of a TM. If all of these operations are possible for a TM, and if they can all be performed in finite time, then we can be certain that a D.N of a TM exists for implementing the UPM, even if we cannot easily generate it herein. The existence of a D.N for this purpose is sufficient for almost any demonstrations that an actual description would be useful for, and thus we do not attempt to generate an actual description.

- **Initial State:** the UPM invokes a finite run time algorithm for determining the "next subject" (S) to be interpreted as a function of the contents of the tape between "left of tape" and "right of tape" without changing that contents. Goto **One Move.**

The **Initial State** of this machine is essentially a scheduler to determine the next subject to be granted a move. We have allowed the greatest possible flexibility in this scheduler, and only require that the next subject be determined in a finite amount of time without effecting the rest of the relevant UPM tape. In practice, we may only be interested in certain classes of schedulers (e.g. "fair schedulers") in any given application, and we note that in our later discussion, we may demonstrate the existence of particular schedules that allow a given activity to occur.

- **One Move:** Once S has been determined, the UPM moves to the "C/P" area of the tape and seeks out a "current program" sequence for S. If no such sequence exists, goto **Next Run**, otherwise goto **Run On.**

The **One Move** submachine arranges to make a single move for a given subject by locating the current program (C/P) for that subject or arranging to load a new program if none is current.

- **Run On**: Copy the subject number and "current program" sequence to the temporary area, and shift all information to the right of the copied area left so as to cover the copied area. Now move to the temporary area, and perform one move for the program stored there. If the program in the temporary area halts on this move, move to the beginning of the temporary area, enter "R", and goto **Initial State**. If the move causes a "special state" to be entered, goto **Special State**. Otherwise, append "P/C" and "R" to the temporary area, and shift the temporary area one square left, thus overwriting the previous P/C marker, and extending the C/P area to include the temporary area used by the "current program". Goto **Initial State.**

The **Run On** submachine actually makes a single move for the current subject by copying the C/P for that subject to the temporary area at the end of the tape, overwriting its old copy with the rest of the C/P area, simulating a single move, and if the program didn't halt, appending the resulting sequence to the C/P area. The particular manner in which this is done assures that the old state of the C/P is overwritten so that subsequent searches of the C/P area will only find the new C/P. We are also assured that the tape does not grow without cause by leaving no excess areas in the middle of the tape.

By moving the C/P to the end of the tape, we assure that if the current move extends the tape of the C/P, we do not have to move additional information (except the "R" marker) to the right to deal with this event. Finally we note that a simple "fair scheduler" could be generated by always appending the "next run" object of any user not in the C/P area to the C/P area, and always running the first entry in the C/P area. Since each program is moved to the end of the C/P area with every move, this implements a "round robin" scheduler which is fair. [8]

In the case that the sequence halts, the **Run On** submachine does not add the temporary area to the C/P area, and thus the program automatically leaves the C/P area upon termination. The only other possibility is that the move causes the C/P to enter a **Special State** which will be described a little later.

- **Next Run**: The UPM moves to the "run list" section of the tape, and seeks out an entry for S. If no such entry exists, goto **Initial State**, otherwise determine the object number (O) of the next object to be interpreted for subject S, and overwrite the marking for that object in the run list by shifting the remainder of the tape left. Goto **Load Object.**

The **Next Run** submachine is used in the case that there is no C/P sequence for the scheduled subject in the C/P area. In this case, the object number of the next object to be run for that subject is sought in the "run list". If no such object if found, the scheduler is again called upon to determine the next subject to be scheduled. Otherwise, the object to be scheduled next is loaded via the **Load Object** submachine. We note here that a scheduler that selects a subject which has no run list entry or C/P sequence for execution may result in an infinite loop with no further moves being interpreted. Finally, we note that the **Next Run** submachine overwrites the marking for each object to be run as soon as it is determined, so that subsequent run list searches will not find the marking again, and space is not wasted.

- **Load Object**: If the entry in S/O for (S,O) does not include the "read" right, or if no such object exists, goto **Initial State**. Append "S" and the proper number of s's to the C/P area to indicate the beginning of the current running program for subject S. Move to the B/O area and seek out the beginning of object O. Copy the sequence stored for the object O to the end of the C/P area so that it is appended to the marker for subject S, and append the P/C and R markers to properly end the tape. Goto **One Move.**

The **Load Object** submachine uses the result of the **Next Run** submachine to determine the object from the object list to be interpreted on behalf of the requesting subject. If there is no such object or if the object to be interpreted is not "readable" by the requesting subject, the object is treated as if it did not exist, and the requested run is simply ignored. If the object exists and is accessible by the subject, it is copied to the temporary area with the subject marker prepended to its description, and one move is made for the program in the normal fashion. We are thus guaranteed at least one move for each program loaded.

We note here that the stringency in this submachine is often not required of actual protection systems because the "run" right is often considered different from the "read" right, and strictly speaking we should base the running of a program on a generic "run" right. In fact, many would claim that allowing the "run" of a program has no effect on security or integrity of information as long as "read" and "write" checks are made on all information accessed by that

program. The above check is necessary if we consider that information about an object may be leaked if it produces any output that is readable by a subject that could not read the object itself. Even the knowledge that the given object exists leaks one bit of information about the object, and thus we must treat the object as if it doesn't exist unless the subject requesting its use has read access to the object.

- **Special State**: Perform the appropriate operations for a special state operation.

Finally, we come to the **Special State** submachine which is a generic submachine that invokes all operations not exclusively limited to the moves of a TM as described by the D.N of a single object. The **Special State** is like a "monitor call" in an operating system that allows an object acting as a surrogate for a subject to request services on behalf of that subject from the underlying UPM. A typical example of such a special state would be a state which is predefined by the UPM to request the reading of an object into tape squares of the current program. We will be discussing special cases of this **Special State** in later sections, and note here that since the **Special State** has access to the entire UPM tape, all **Special State** cases must maintain protection restrictions for the UPM to operate correctly.

At this point we argue that the above specifications, with the exception of the **Special State** submachine, specify TM programs which are implementable with finite time algorithms and which take finite space on the UPM tape for all finite initial states and finite numbers of moves. We thus conclude and postulate that such a machine exists, even if we have not explicitly specified it. We further postulate that as long as all **Special States** of such a machine fit the above criteria, the resulting machine exists.

3.4 A Model of Computers

Rather than work with this complex description of the UPM, we abstract out the details of UPM operation in favor of an operational model. We thus define a *computer* as:

(1) an *interpretation unit* **that:**

i) fetches initial process states for subjects from objects
ii) schedules processes for interpretation
iii) interprets moves for processes
iv) manages information on the computer's tape

(2) a set of subjects $(s_1,\ldots,s_m)$ **and objects** $(o_1,\ldots,o_n)$
and an "access matrix" which specifies a protection configuration:

r **in** (s_i,o_j) **for** $0<i<m+1,\ 0<j<n+1,$
w **in** (s_i,o_j) **for** $0<i<m+1,\ 0<j<n+1$

(3) a "run sequence" of objects to be interpreted for each subject.

In operation, the scheduling mechanism selects the subject whose move is interpreted at each interpretation step. When and if a process halts, the next move for that subject is interpreted from a process initialized by reading the next object in that subject's run list. If there exists no such object or if r is not in that object for that subject, the next object in that subject's run list is chosen, while if there are no further objects in that subject's run list, no process is invoked.

At least three Special State cases exist for the particular computer that we will be considering herein, the "read" state, the "write" state, and the "interpret" state. We describe here the events for these cases.

Upon entry into the "read" Special State, the symbol under the tape head must be one of {0,1,...m} where the integer corresponds to an object number in the access matrix, or the process will halt. If the object number corresponds to an invalid access matrix entry, or the S/O entry does not contain the "read" privilege for the (subject,object) pair under consideration, the process enters the "read failed" (RF) state. If the integer corresponds to a valid access matrix entry and

the user has the "read" privilege for that entry, then the sequence of tape symbols corresponding to that entry is placed on the tape starting from the current tape position with each subsequent symbol being placed on a subsequent tape square. When the "read" operation is completed, the normal next state of the process is entered, with the tape head over the left most cell of the sequence read in.

Upon entry into the "write" Special State, the symbol under the tape head must be "BO", and the symbol directly to its right must be one of {0,1,...m} where the integer corresponds to an object number in the access matrix, or the object will halt. If the integer corresponds to a valid access matrix entry, and that entry does not contain the "write" privilege for the subject under consideration, the object enters the "write failed" (WF) state. If the integer corresponds to a valid access matrix entry and the user has the "write" privilege for that entry, then the sequence of tape symbols on the tape up until the first "EO" symbol, starting from the current tape position with each subsequent symbol being taken from a subsequent tape square, replace the stored object corresponding to that integer. When the "write" operation is completed, the normal next state of the process is entered, with the tape head over the left most cell of the written sequence.

Each tape sequence stored or retrieved from the object memory must be in the following format, or the process may never halt, and the stored sequence will not be effected:

Tape square	Tape symbol
0	"BO" (Beginning of Object)
1	object number
2	1st symbol
...	...
n	last symbol
n+1	"EO" (End of Object)

The "interpret" Special State causes the UPM to begin interpretation of a sequence at the current tape square as the D.N of a UPM program. We note that this is not a necessary state in the sense that any program being interpreted could itself interpret the other program by simulating a UPM operating on that machine [53], but that it is a convenient state in that it saves a great deal of difficulty in further examples.

3.5 A Simple Virus

We now demonstrate a self replicating object o_c which, if interpreted by a subject s_u with *r* in (s_u,o_c) and *w* in some (s_u,o_z), can copy its own contents into o_z, and thus modify o_z to include a copy of itself. We note that any object that replicates itself outside of itself is a virus (Lemma 2.1), and that thus the following object is a virus.

```
SxI      N        O       D
-------------------------
s0,BO    s0'      BO      0       ;check for start of object
s0,else  s0       else    0       ;or halt
s0',*    CPY(BO,EO,EO)             ;copy object to after self
s0''     L(BO)                     ;get to beginning of object
s0''',*  s1       BO      +1      ;move over object number
s1,x     write    [x+1]|n -1      ;replace object number
s2,*     s1       BO      +1      ;loop to next object #
WF,*     s1       BO      +1      ;even if write failed
```

If we examine this program, we see that it simply copies itself, changes the object number, and writes the next object as a copy of itself with a different object number. We note that regardless of the length of the object required to indicate this machine to the UPM interpreting it, the write will duplicate the entire sequence, and that for any finite n, this constitutes an SVS of size n. If there exists some subject s_u with *r* in (s_u,o_c) and *w* in some (s_u,o_z) where $z \geq n$, then as o_u is interpreted, the object o_z will come to contain a virus.

Although state s0', s0'', and s0''' help fulfill the Turing machine definition of a virus given earlier, the storage system maintaining the objects of the UPM constitute sequences of symbols that may be subject to interpretation. In order for a sequence to be a virus, it must merely cause a (possibly evolved) version of itself to be created outside of itself in the storage system. Thus, we have the following simplified version of a virus called "OV" for the computer under consideration.

```
SxI      N       O       D
-------------------------
s0,BO    s1      BO      +1      ;check for start of object
s0,else  s0      else    0       ;or halt
s1,x     write   [x+1]|n -1      ;change object number and write
s2,BO    s1      BO      +1      ;loop to next object #
WF,*     s1      BO      +1      ;even if write failed

         UPM Virus "OV"
```

3.6 Viral Transitivity

We feel compelled here to discuss the "run list" and "scheduling algorithm" which we have purposely left nebulous until this point. In order to prove that a protection system is "safe", we generally wish to prove that a particular set of states or sequence of events CANNOT occur. We therefor wish to consider the "possibility" of the existence of a sequence of events which result in particular effects on the state of the UPM.

Our modeling problem is one of determining which aspects of machine operation should be fixed, and which should be allowed to vary. We justify our choice of arbitrary run lists and scheduling by explaining that in an actual computer system, the run list and sequence of object interpretation are not in fact determined a-priori, but rather result from the relatively unpredictable use of the system by users. In particular, we may rest assured that any specific sequence of interpretations of objects by subjects is possible.

As an example of the utility of the choice of arbitrary scheduling and run lists, let us suppose that there exist objects o_1, o_2, and o_3 and subjects s_a and s_b such that:

r in (s_a, o_1),
w in (s_a, o_2),
r in (s_b, o_2),
w in (s_b, o_3)

From the example above, we know that:
if o_1 starts with OV AND
o_1 is interpreted by s_a at time t
then o_2 contains OV at some time $t' < t$

We also know that:
if o_2 starts with OV AND
o_2 is interpreted by s_b at time $t'' > t'$
then o_3 contains OV at some time $t''' < t''$

We thus know that:
if o_1 is in s_a's run list and
if o_2 is in s_b's run list and
if the scheduler schedules:
o_1 for s_a at time t and
o_2 for s_b at time t''
and if o_1 completes OV at time t'<t''
then OV spreads transitively from o_1 to o_3.

We say that o_x can infect o_y iff
[∃ a set of run lists [∃ a scheduling of moves
[∃ v ∈ V
[(UPM,V) ∈ VS and
v $\overset{UPM}{\Rightarrow}$ V and
[∀ o_y at time t [∃ v' ∈ V
[∃ t' ∈ ℕ
[v ⊂ o_x at time t and t' > t
and v runs at time t
⇒ v' ⊂ o_y at time t']
]]]]]]]

In other words, an object X "can infect" another object Y if and only if there is a set of run lists, a scheduling of runs, and some virus v which, if it is in X and is interpreted at time t, causes some virus v' to appear in Y at some later time t'. We say that Y is "infectable" by X iff X can infect Y.

We may now easily show that if X can infect Y and Y can infect Z, then X can infect Z. In other words infectability is transitive. We show transitivity by noting that:

if X can infect Y then
there is a sequence of events S1
which causes infection of Y by X

and if Y can infect Z then
there is a sequence of events S2
which causes infection of Z by Y

We now note that if there exist sequences S1 and S2 then there exists a sequence S3 which consists of S2 appended to S1, which causes infection of Z by X. Thus infectability is transitive.

We note also that it is fairly straight forward to show that "sharing" is also transitive, although this is not of particular interest to our discussion at this point.

3.7 A More Advanced Virus

We now demonstrate a virus that is more advanced in that it is considerably harder to detect than the above examples. In particular, this virus modifies programs so as to leave their functionality unchanged. The basic principal is to prepend a virus to the program being modified so that upon completion of the infection of other programs, the infected program executes normally. Thus, the final configuration of the infected program should look something like this:

```
tape square     contents
-----------     --------
n               "BO"
n+1             object number
...             virus code
n+k             "BO"
n+k+1           object number
...             original object
n+m             "EO"
```

The virus is described as follows:

```
SxI     N       O       D
-------------------------
s0,BO   s1      BO      0       ;verify BO
s0,else halt                    ;or halt
s1,*    CPY("BO","EO","EO")     ;replicate
s2,*    L("BO")                 ;move left till original program
s3,BO   s4      BO      +1      ;move to object number
s4,x    read    [x+1]|n -1      ;read next object
s5,*    L("BO")                 ;get to virus copy BO
s6,BO   s7      BO      +1      ;move to object number
s7,x    write   [x+1]|n -1      ;write infected object
s8,*    L("BO")                 ;left till original program
s9,BO   interpret BO    0      ;run that program
```

The reader may verify that this machine generates the arrangement above, and we will not do this here. What is most worthy of note here is that the virus is able to infect another program and then execute its host as if there were no virus present. This example ignores issues such as the access rights to the [x+1]|n numbered object, but is intended only to demonstrate the concept, not to be the ultimate virus. We note for the more rigorous reader that even if infection of another program cannot be carried out, this program is a virus since it replicates itself on the tape before attempting to effect an object in the subject object memory.

Further extensions of this program would be the inclusion of a detection mechanism that would not infect other programs if they were previously infected, a pseudo-random number generator using the object number as a seed to overwrite the prepended virus prior to execution of the infected program so that it would be difficult to determine whether the program being executed was infected from within itself, additional evolutionary capabilities, more specific targets for infection, detection of the contents of an object to verify that it is the D.N of a TM program rather than another type of data, the ability to infect data formats intended for interpretation by specific TMs (such as language interpreters), and any number of other advances.

3.8 Model Extensions and Comments

In order to extend the UPM model to networks of computers, we may choose to simply add special states which transmit or receive sequences of symbols to or from other UPMs through a well defined communications protocol. Access rights to the network are determined by the access matrix, and some set of rights to access the network are encoded in access matrix entries.

A similar mechanism can be used to embody functions commonly associated with an operating system, by allowing special states to act as an inter-process communications method, and granting some special process access to relevant portions of the UPM tape. As examples of the power of this mechanism, we can implement the "fork" and "join" operation by simply introducing and removing multiple objects into and from the C/P area of the tape, we can provide inter-process communications by providing read and write access for each of a set of objects used for communication, and we can provide synchronization mechanisms by moving sequences in and out of the C/P area in much the same manner as swapping moves processes in and out of the main store in many operating systems. [8]

This special state mechanism is quite general, and the most general manner in which it can be used is by allowing some special process full access to the UPM tape. Since the UPM has Turing capability, and the special states allow an arbitrary computable function to be evaluated with the results left on the UPM tape, any more general mechanism would require a machine of greater computing power than a TM.

The problem with this sort of mechanism is that the special process may be too powerful. As an example, this mechanism is powerful enough to make the "safety" of the protection system undecidable since it is undecidable whether or not the special process modifies a given access matrix entry. [32] In essence, we must prove properties of the special process program in order to be able to prove the safety of the protection system. This is what we mean when we speak of a provably secure system. [37]

In the network analogy, we must prove that our system is "secure" given some set of constraints on the rest of the network. If we assume the most general case of the rest of the network, we must assume that no real protection is provided outside of our UPM, and we are left in a very restrictive case. As we shall see in later sections, the restrictions on UPMs and networks containing them may be quite severe, depending on our requirements.

4. Prevention of Computer Viruses

Having planted the seeds of a potentially devastating attack, it is appropriate to examine protection mechanisms that might help defend against it. We examine here "absolute" prevention of computer viruses, wherein viral spreading is made mathematically impossible.

4.1 Basic Limitations

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

Given a general purpose system in which subjects are capable of using information in their possession as they wish and passing such information, as they see fit, to others, we have established that the ability to share information is transitive. That is, if there is a path from subject A to subject B, and there is a path from subject B to subject C, then there is a path from subject A to subject C with the witting or unwitting cooperation of subject 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 previously. If there is sharing, infection can spread through the interpretation of shared information. If there is no restriction on transitivity or information flow, then 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.

Clearly, if there is no sharing, there can be no dissemination of information across subject boundaries, and thus no shared information can be interpreted, and a virus cannot spread outside a single subject. 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 interpretable 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.

4.2 Partition Models

Two limits on the paths of information flow can be distinguished, those that partition systems 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, and thus they 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 combination of the Bell-LaPadula security model [3] with the Biba integrity model [5] is an example of a policy that can partition systems into closed subsets under transitivity. Mathematically, the security model is defined over a set of "security levels". Each "user" of a system is assigned to a given security level, and all activity of that user occurs at that level. Sharing is limited by two properties; the "simple security property", and the "*-property". The "simple security property" states that a user at some level (x) may not read information from a security level exceeding x. This is often referred to as "no read up". The "*-property" states that a user at some level (x) may not write information to a security level lower than x. This is often referred to as "no write down". A simple generalization [19] has resulted in the

mathematical use of a lattice structure to describe these two properties. The integrity policy is just like the security policy except that the rules are reversed, and the word "integrity" substituted for "security". Thus, we have the dual of the security policy in the integrity policy. The integrity policy is often stated as consisting of the rules "no read down", and "no write up", but we must not forget that integrity levels might not be partitioned in the same manner as security levels. Examples of these two policies are shown graphically below.

If the integrity model and the security 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 as is demonstrated graphically below. When the integrity model has boundaries within the security model boundaries, infection can only spread from the higher integrity levels to lower ones within a given security level. Finally, when the security boundaries are within the integrity 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 cases shown graphically below are sufficient for understanding.

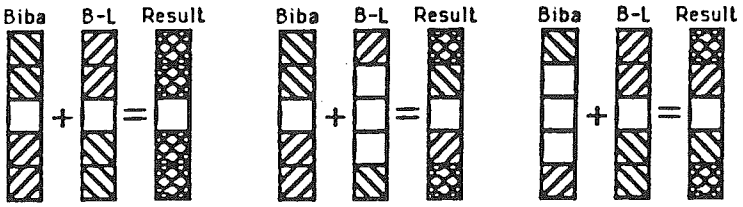


Figure 4.1 - Combining Security and Integrity

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, and these policies will not be considered further here.

Just as systems based on the security model tend to cause all information to move towards higher levels of security by always increasing the level to meet the highest level user [19], the integrity 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 (by duality). [19]

The most trusted user is (de-facto) the user that can write information accessible by the most users. In order to maintain the security policy, high level users cannot write programs used by lower level users. This means that the most trusted users must be those at the lowest security level. This seems contradictory. When we mix the security and integrity models, we find that the resulting isolationism secures us from viruses, but doesn't, of course, permit any user to write programs that can be used throughout the system.

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 compartments required for their duties. If every user in a strict compartment system has access to only one compartment at a time, the system is secure from viral attack across compartment boundaries because compartments are isolated. Unfortunately, in current systems, users may have simultaneous access to multiple compartments. In this case, infection can spread across compartment boundaries to the transitive closure of information flow.

4.3 Flow Models

In policies that don't partition systems into closed proper subsets under transitivity, it is also possible to limit the extent over which a virus can spread. The "flow distance" policy implements a distance metric by keeping track of the number of sharings over which data flows. 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 is at least distance 9.

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. Information starting at B can, however, transit to A, so long as it is not mixed with information from C

Rules:

```
D(output) = max(D(input))
D(shared input)=1+D(unshared input)
Information is accessible iff D < const (2 in this case)
```

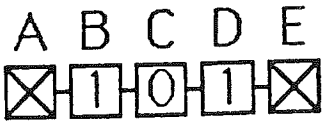


Figure 4.2 - A Distance Metric with a Threshold of 1

The 'flow list' policy maintains a list of all users who may 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 interpretation). 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.

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 Bell LaPadula, the Biba, and the distance models.

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 a note, 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.

4.4 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 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.

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. So long as the computer mail system is partitioned from other applications so that no information can flow between them, and is of sufficiently limited functionality as to make viral spreading in the mail partition impossible, we 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, COBOL, Fortran, 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 presumably 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 therefor provide upper bounds on the spread of a virus in systems with limited interpretation.

4.5 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 precise information flow is NP-complete, and maintaining precise markings requires large amounts of space. [19] As a simple example of the complexity of precisely maintaining the sources of information, consider the problem of determining the source of the result of an OR of two bits. Suppose bit A is from user U, and bit B is from user V. If we OR these bits together, we get either a 1 or a 0, but the question arises of which user's information we are getting. We summarize the answer to this question in the following table:

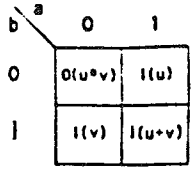


Figure 4.3 - Precision Problems

In two cases, we get only the information from one user, and know which one. In the 0 case, we have information from both users, while in the fourth case, we have information from either user or both. If many users are involved and information is manipulated to any large extent, the complexity of maintaining these markings will become very high. Imagine the case where user W may only access information from U. Only the case where A is 1 is available since all other cases reveal information from V.

A more general, and more severe caseis that where W may only access information from U or V, but not both. Clearly, if we tell W that the result of an OR is unavailable, it indicates that both A and B must be 0! Thus by telling W the information is unavailable, we give the information away. The OR operation in this case must be dissallowed, even though certain cases of the OR should legitimately be available. Thus, in the general case, precision is impossible.

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 source of the problem is that, when information has been unjustly deemed unreadable for some 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 until it either becomes completely isolationist or reaches a stability point where all estimates are precise. If such a stability point exists, we have a precise system for that stability point. Since we know that any precise stability point except isolationism requires the solution to an NP-complete problem, we know that any non NP-complete solution, must tend towards isolationism. In the most general case, we have shown that even NP-complete solutions may not be sufficient. We refer the interested reader to [19] for a more complete discussion.

4.6 Summary

The following table summarizes the limits placed on viral spreading by the preventive 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.

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

Figure 4.4 - Limits of Viral Infection

5. A Secure Network Based on Distributed Domains

Given the extreme openness and communications level of current computer networks, the threat of attack is severe. [33] In most current computer networks, sets of heterogeneous computer systems are connected through heterogeneous communications networks using a wide variety of communications devices, protocols, and programs. [25] [6] [9] [13] One fact that is not widely publicized is that these networks are not intended to be secure in any way. [25] Both the communications lines and intermediate computers used for data transfer are open to widespread observation and/or modification.

Legal protection is provided in most states against unauthorized wire tapping and wire fraud, but proof of the intruder's guilt is often difficult, and the damage done may not be cured simply by arresting an attacker. The most predominant networks have open memberships, allow computer mail and file transfer between nearly any pair of computers with arrival times ranging from seconds to hours after requests, and connect to major computer manufacturing and software houses.

5.1 Background and Overview

Protection Policies and Models

In order to make any system secure, we must first consider what we mean by the word secure. A "security policy" is a formalization of the desired security goals. Implementation of a policy is usually done with the use of a formal model of desired behavior. This section of the thesis examines a security policy in which both illicit dissemination and modification of information are impossible. The design of secure computer systems has been studied by many authors [39] [24] [32] [27] [40] [19], and as we saw earlier, for the protection of information from illicit disclosure and modification in a general purpose system, a design with both a security policy [3] [19] and an integrity policy [5] affords limited protection.

We will assume that the security and integrity models reviewed earlier are the basis for protection policies, that both are always in effect, and that they are identically partitioned. This combination leads to distributed isolationism, a policy wherein "subjects" [32] with a given access "level" [3] cannot communicate with subjects at any other access level. In essence, we are using a network to allow spatial distribution of isolated domains, so that the functionality of many different facilities in different physical locations may be treated as an isolated system. We use the term "distributed domains" to describe such a system.

Where sufficient, a (security, integrity) level pair will be referred to simply as a "level". The term "subject" in this text refers to a single "identity" as perceived from the point of view of the policy. In actual implementations, a person may be identified with many subjects, but in the formal model, we assume that subjects are independent of each other. We always assume that all communications of concern to our implementation are those that go through the computer systems and networks we are designing. We will also assume that all systems in the network are general purpose.

Implementation Problems

Once a desired policy has been specified, an implementation of it must be used in order to result in a secure computer system or network. In order to guarantee that an implementation correctly implements the policy, we must be able to prove it mathematically. Provably secure operating systems capable of enforcing an isolationist policy have been designed and implemented [4], but secure network design has only recently been investigated. [54] None of the proposed systems perfectly solve the "covert channel" problem [39], although identification and measurement of covert channels is possible.

The covert channel problem comes from the fact that when subjects share a resource, the manner in which one subject uses the resource may be detectable by another subject with access to that resource. By examining the statistical behavior

of programs which use shared resources, it is possible to extract information regardless of the degree of noise in this statistic. [48] The bandwidth of covert channels is limited by the amount of noise in the channel, and the quantity of information that can pass through a channel as a function of time can be determined and measured. A related problem is the problem of "traffic analysis" in which information is obtained by detecting the patterns of traffic in a network. The traffic analysis problem can be addressed in the same manner as the covert channel problem through the use of information theory. We will not discuss the covert channel problem further in this work, although it is both interesting and important to modern secure networks.

Two basic types of computer systems can be distinguished, systems based on a trusted computing base (TCB) in which operation is proven to meet a security policy [24] [32], and systems based on an untrusted computing base (UCB) in which there may be policy, design, and/or implementation flaws. [37] [41] As we will see, fundamental limitations must be placed on allowable information flows between these systems if there is to be any hope of controlling the dissemination and modification of information.

Communications Between Computers

Whenever computers are connected to form a computer network, there are some physical links over which communication between these computers takes place. Two basic types of communication links can be distinguished, links in which communication is physically secured from external intrusion and observation, and links in which illicit observation and/or modification of data is possible. In the case of trusted communication links, we assume that illicit modification or observation of information is impossible. With untrusted communication links, protection of communicated information from illicit dissemination requires that the information be transformed into a form which will not reveal its content, while protection from acceptance of illicit or illicitly modified information requires some form of authentication. These two goals can be accomplished through the use of cryptography. [7] [44]

Shannon's information theory [48] and work on secrecy systems [49] form the mathematical foundation for most modern analysis of cryptosystems, and are the basis for the designs of many modern "one key" systems like the DES. [16] [23] The introduction of "public key" cryptography [22] brought about drastic changes in the research perspective towards cryptography, with complexity based protection becoming a prevalent area of mathematical analysis. In public key systems there are two keys; the "public key" which may be revealed to the public and used either for encryption of messages sent to the key creator or for public authentication of messages signed by the key creator; and the "private key" which is kept confidential by its creator and may be used either for decrypting incoming messages or signing outgoing messages. It is not necessary that the "public key" be revealed to the public, and any public key system can be used as a private "two key" system. The RSA cryptosystem [46], a system based on the complexity of factoring very large primes [55], is the most well known and most studied of the public key cryptosystems, is currently thought to be very secure and practical, and has been implemented in several hardware and software systems.

The existence of a high quality cryptosystems alone, is insufficient to provide for secure use of a network; security depends on the proper use of encryption. The manner in which cryptosystems are used is specified by a "cryptographic protocol". A cryptographic protocol may be thought of as a well specified and systematic means for applying a cryptosystem to a specific problem. In the case of a provably secure network, protocols must be formally shown to meet the formal specifications of the security policy.

In conventional one key systems, protocols are fairly straight forward [26], but functionality is quite limited. The concept of public key cryptography has led to many papers on cryptographic protocols for increasing the utility of a cryptosystem. [15] [18] [43] Public key based network file servers have been investigated [29], and practical designs are emerging. Threshold based systems [47] can be combined with public key systems to allow a secure key distribution system [11] even in the presence of tappers and illicit distributors. Secure key exchange protocols have been developed [43] so that two subjects that have never met can obtain a secure communications path in an untrusted environment. Authentication protocols for allowing legal document signatures have been examined [46] [43], and usable systems have been proposed. Among the most advanced current uses of an RSA based cryptographic protocol, is the system used for verification of the nuclear test ban treaty. [51]

Overview of Results

We first examine networks in which communication lines are considered trusted paths and connections may be made at any security and integrity level. We show that bidirectional communication between UCBs is only acceptable when they have identical integrity and security levels, and that a UCB cannot safely send information to a TCB unless the UCB is at a single security and integrity level. This analysis is then expanded to untrusted communications networks where connections can only be made at the lowest level. We show that UCBs can only be linked directly to the network at the lowest integrity level, while TCBs can be used at all levels with the use of a "good enough" cryptosystem. These cases combine to form a set of easily applied design rules for the connection of computers to form secure computer networks.

Protocols that do not violate security or integrity conditions are shown, and a "good enough" cryptosystem [46] is shown to fulfill all of the network security and protocol requirements. Analysis of attacks based on the compromise of one subject or facility are then shown to be potentially devastating unless further protection is provided. The use of compartment based protection with each site accessing only a restricted subset of the totality of compartments is shown to limit the potential damage of such attacks, but may not be ample protection for many applications.

5.2 Network Communications

The fundamental goal of the network security policy considered here is that information not be able to move down security levels or up integrity levels. The assumption that integrity and security levels are aligned implies that information may only move about at its creation level. Unfortunately, in UCBs operating at multiple levels, strict alignment is unenforceable, and thus special provisions must be made. We first consider the formation of networks in environments with trusted communication paths and derive a set of easily followed design rules.

Networks with Secure Communications Paths

In a secure network with trusted communications paths, communications are allowed from place 1 (P_1) to place 2 (P_2) if and only if the security level of P_1 (S_1) doesn't exceed that of P_2 (S_2), and the integrity level of P_2 (I_2) doesn't exceed that of P_1 (I_1). This is because communication from P_1 to P_2 with $S_1 > S_2$ violates the simple security rule [3] and would allow illicit dissemination of information, and communication from P_1 to P_2 with $I_1 < I_2$ allows viral spreading up integrity levels, which allows illicit modification of information.

Connecting UCBs with UCBs

If we consider that a UCB is a computer that cannot be trusted to maintain security or integrity levels within itself, we can regard it from an external point of view as having the security level of the most secure information processed in it (system high security) and the integrity level of the lowest integrity information processed in it (system low integrity):

$$\text{in a UCB: } I = \min(I \text{ in UCB}),\ S = \max(S \text{ in UCB})$$

This is a direct result of the fact that any information at a high security level could be declassified by a UCB, and thus if we allow output from a UCB at lower than the highest level of information processed within it, information could be moved from a higher security level to a lower security level and thus be illicitly disseminated. Similarly, low integrity information within a UCB could be output at a higher integrity level because the UCB cannot be trusted to maintain integrity levels. This would allow a virus to spread to higher integrity levels and thus allow illicit modification of information. We then obtain the rules for safe information flow given in figure 1.[2]

---

[2] Unidirectional communication of information from system "1" to system "2" will be written as "1-->2" or as "2<--1", and bidirectional communications between systems "1" and "2" will be written as "1<-->2".

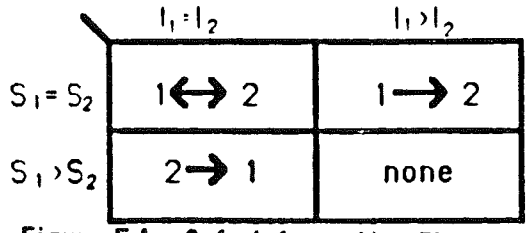


Figure 5.1 - Safe Information Flow Rules

By using a simple set of examples, we can display these equations in terms of pictures. In order to determine whether a connection can be made, a designer can then use these pictures to make decisions rather than having to solve equations. Figure 2 shows the equations from figure 1 in pictorial form. The 4 parts of figure 2 represent the four cases from figure 1. Each system is represented by a set of connected boxes and is labeled by the number of the system as used in the equations. The "high", "medium", and "low" designations indicate different levels in the system, and the arrows between systems show permissible connections and the allowable direction of information flow. An 'X' is used in the case where no communications between the systems is permitted. Notice that communication links are never allowed to cross level boundaries, and that bidirectional communication is only possible when $S_1 = S_2$ and $I_1 = I_2$.[3]

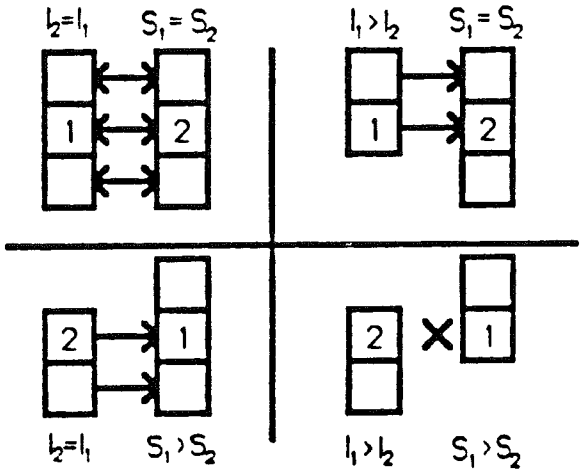


Figure 5.2 - Safe Communications Paths Between UCBs

Since the equations in figure 1 follow the rules that no information can ever flow from a higher security level to a lower security level or from a lower integrity level to a higher integrity level, and since the $<$, $>$, and $=$ relationships used in these equations are transitive (e.g. $A<B$ and $B<C \Rightarrow A<C$), these security relations hold over the transitive closure of information flow. We conclude that any network of UCBs in which the rules from figure 1 are followed locally for each connection between computers, will globally meet the network security and integrity requirements. In other words, if every connection looks like the pictures in figure 2, the network will meet the security requirements as stated. This "cookbook" approach to designing secure computer networks made up of UCBs with secure communications links will now be extended to networks with mixed UCBs and TCBs and networks with untrusted communications links.

Connecting UCBs with TCBs

---

[3] In fact, with UCBs communication links can cross level boundaries so long as all levels with communication exist in both systems because the UCB cannot be trusted to maintain these levels anyway.

In a network containing both UCBs and TCBs, we must consider that although a TCB can be trusted to maintain both security and integrity levels, a UCB can be trusted to do neither. Consider a network consisting of a single TCB (1) and a single UCB (2), both operating at two levels (high and low). Since the UCB cannot be trusted to maintain these levels, we must consider it externally as a computer with:

$$S_2 = \max(\text{high},\text{low}) = \text{high}$$
$$\text{and } I_2 = \min(\text{high},\text{low}) = \text{low}.$$

Under the Bell-LaPadula model (B-L), we conclude that no information can flow from the UCB to the TCB at any security level below S_2 (high) without violating the *-property and thus allowing illicit dissemination of information. Under the Biba model, we conclude that no information can flow from the UCB to the TCB at any integrity level above I_2 (low) without allowing illicit modification of information. We conclude that the only communication that can be allowed is unidirectional from the TCB to the UCB. This derivation is shown graphically in figure 3 below, and is trivially extended to systems with an arbitrary number of levels.

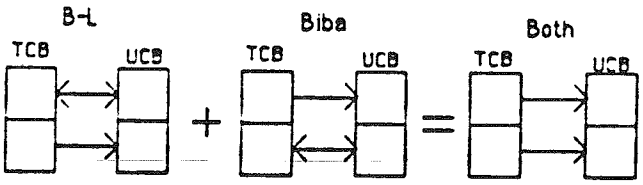


Figure 5.3 - Combining B-L and Biba Between UCB and TCB

The unidirectional communication problem seems to imply that reliable communication is impossible without leaking information through a covert channel formed by the UCBs responses to protocols. This is easily seen in the case where a subject in a UCB sends a bit to a subject in a TCB by; filling the UCB's disk so that a transfer cannot be successfully completed from TCB to UCB to indicate a 0; and freeing up this space so that a transfer from TCB to UCB can be successfully completed to indicate a 1. As an alternative to allowing this channel, it may be possible to design a portion of the TCB with limited functionality such that transfer protocols can be done reliably without end to end confirmation. This limited confirmation with the TCB will not reliably indicate the success or failure of the transmission to the transmitting subject, but it is secure from this covert channel, while allowing reliable communication after an unknown delay.

The only case where a UCB and TCB can communicate bidirectionally is the case where the UCB operates at a single level equal to that of the communicating TCB level. This type of connection doesn't violate security or integrity because $S_{UCB} = I_{UCB} = S_{TCB} = I_{TCB}$. Finally, we assert that two TCBs can communicate bidirectionally over a trusted communications link at any level at which both exist, since they can both be trusted to maintain security and integrity constraints on all information. The acceptable communications links between UCBs and TCBs and between pairs of TCBs are shown in figure 4.

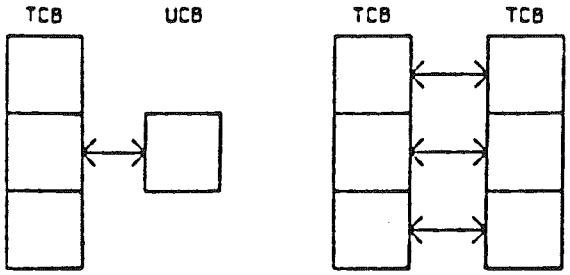


Figure 5.4 - Communications Between UCBs and TCBs

As with UCBs, the relations of security and integrity models hold over the transitive closure of information flow and thus networks can safely be formed using the rules for connections shown in figure 4. With the above results, we can straight forwardly connect UCBs and TCBs into trusted computer networks in any environment where communication links between systems are trusted, without fear of either security or integrity violations, so long as each system maintains

its specified properties. An example of such a network is shown in figure 5. Verification that it meets the above connection criteria can easily be done by observing that only connections of the forms shown in figure 4 are used. This network therefore meets the security requirements specified by the policy under consideration for trusted communication environments.

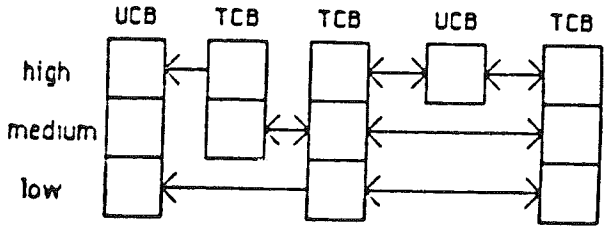


Figure 5.5 - A Secure Net w/ Trusted Communications

Networks with Untrusted Communications Paths

In spatially distributed networks or networks operating within untrusted environments, untrusted communications paths must be used. In general, an untrusted communications path can not be relied upon to either maintain the secrecy of information flowing through it, or to prevent an attacker from introducing false information to it. Both authentication and secrecy are clearly required if secure communication is to take place.

Network Level Communications

In an untrusted communication path, we must consider all data as being at the lowest integrity level since it could have been manufactured or modified by an attacker, and at the lowest security level since a tapper could observe information in transit. Thus:

$$S_{network} = \min(\text{security-levels})$$
$$\text{and } I_{network} = \min(\text{integrity-levels})$$

From the previous analysis, UCBs may output to a network iff

$$S_{UCB} \leq S_{network},$$

and it may input information from a network iff

$$I_{UCB} \leq I_{network}.$$

Since

$$S_{network} = \min(\text{security-levels})$$
$$\text{and } I_{network} = \min(\text{integrity-levels}),$$

bidirectional communication requires that

$$S_{UCB} = S_{network} \text{ and } I_{UCB} = I_{network}.$$

while reception of information from a network by a UCB requires only that

$$I_{UCB} = I_{network}.$$

Since TCBs enforce levels, communication with levels in TCBs where

$$S_{TCB} = S_{network} \text{ and } I_{TCB} = I_{network}$$

is safe. Thus we can connect any TCB with a level at $S_{network}$ to an insecure network, without violating the system or network security and integrity policies. These cases are shown pictorially in figure 6, and as before the results extend transitively so that these pictures can be used to design a secure computer network.

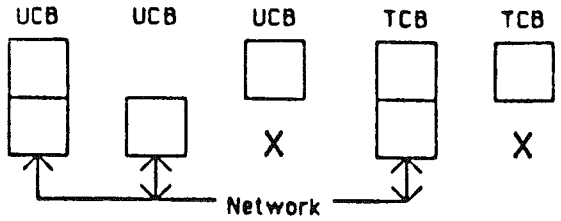


Figure 5.6 - Safe Communications with Untrusted Nets

High Level Communications

The problem remaining is that only data at $S_{network}$ and $I_{network}$ can be placed on the network, and it may be desirable to communicate higher level information. If typical network performance levels are desired, a means of automatically reducing and increasing the level of information at a reasonable speed on a demand basis seems necessary. This can be provided if we have a "good enough" cryptographic function "E" with built in authentication such that:

$$S_{E(data)} = S_{network} \text{ and } I_{E(data)} = I_{network}$$

and a "good enough" inverse function "D" such that:

$$S_{D(E(data))} = S_{data} \text{ and } I_{D(E(data))} = I_{data}.$$

Assuming that an appropriate cryptographic function is available, we can communicate any desired information over the network by transforming it to the network level. Since all information in the network is at the same level, the network meets the policy requirement. Since all computers in the network communicate at the same level, there is no covert channel due to bidirectional communication protocols between processes at different levels. A simple example of this type of system is shown in figure 7 where "E/D" is used to indicate an encryption/decryption link which allows information at one level to be sent to another level through appropriate encryption or decryption.

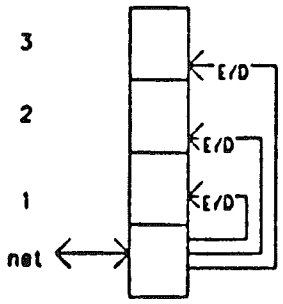


Figure 5.7 - Simple Encryption/Decryption

As before, transitivity of the "=" relation allows any desired connectivity between computers at the network level without violating policy requirements. We also note that the addition of UCBs to the network under the previous rules has no detrimental effect and maintains the transitivity property because the only UCBs that can pass information out are single level UCBs at the network level, and single level UCBs connected to appropriate TCB levels, and thus the rules given in figure 6 still apply.

End to end protocols can be implemented for data sent between identical levels since there is a means of transforming the data to and from the network level. Since encryption and decryption guarantees that no communication is permitted between nonidentical TCB levels, this is sufficient to assure maintenance of these levels. Note that the encryption and authentication functions E and D must be built into the TCB so that it can be proven that there is no possible manner in which levels can communicate except through the proper transformation of information. Also note that there may be covert channels available through the use of traffic analysis unless further precautions are taken. This will not be discussed further here.

A final problem that must be addressed in an untrusted network involves communication between computers where there is no direct path at the network level. This is illustrated in figure 8 in the case of communication from A to B.

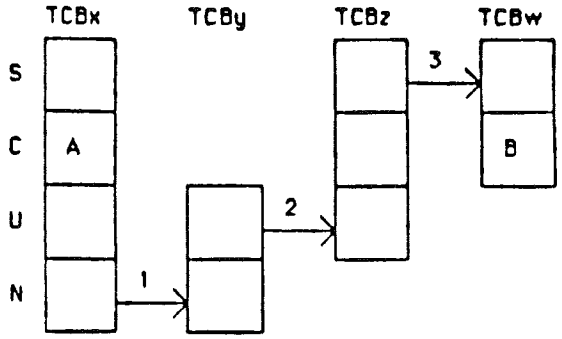


Figure 5.8 - A Multihop Communications Problem

Since data at A cannot be sent to TCB-Y except at level N, it must be transformed into E(data) for transmission. Once inside TCB-Y, it cannot be decrypted into D(E(data)) since this would leave the data at level U, a violation of the security condition. It also cannot be kept in the E(data) form since this is at too low an integrity level for transmission over 2. If decryption in the cryptosystem used were as secure as encryption, we could decrypt the information to level U with the hope of later encrypting back to level N and then decrypting back to level C. Unfortunately, there is no other place in this network where such a transition can be made. Sending the data over link 3 presents the same sort of problem because the integrity must be increased to level S in TCB-Z in order for it to be sent over 3, and then decreased to C in order to reach B. We are faced with a potential problem which we call the "level shifting" problem.

5.3 A Proposed Network Protocol

There are several potential solutions to the level shifting problem seen in figure 8. The simplest and perhaps most reasonable technique is to require that each level of declassification require independent encryption and authentication, and that each level of reclassification require independent decryption and authentication. In other words, we require a cryptosystem and communications protocol where:

$$S_{E(data)} = S_{data} - 1,\ I_{E(data)} = I_{data} - 1,$$
$$S_{D(data)} = S_{data} + 1,\ \text{and}\ I_{D(data)} = I_{data} + 1.$$

This type of system is shown in figure 9.

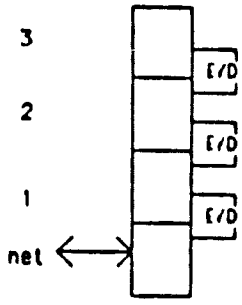


Figure 5.9 - Stepwise Encryption for Level Changing

With the technique in figure 9, the problem in figure 8 is easily solved. Data is encrypted twice in moving from A to 1, decrypted once for transmission over 2, decrypted twice more for transmission over 3, and encrypted one last time to reach B. A similar path is required in the reverse order for transmission from B to A. This stepwise encryption solution of figure 8 is shown in figure 10, where E and D label each information path by its function.

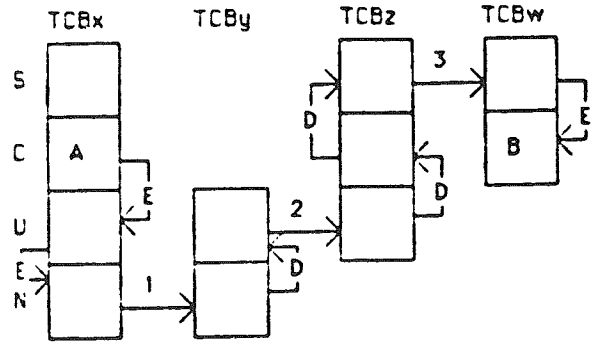


Figure 5.10 - A Multihop Communications Solution

This protocol has cases where information has been decrypted more times than it has been encrypted, and allows plaintext to be found in intermediate network locations. This is not a violation of the security or integrity policy because it is at the same level as the source data. The protocol requires the use of a cryptographic algorithm in which encryption inverts decryption and decryption is as cryptographically strong as encryption. In other words,

$$E(D(\text{data})) = \text{data and}$$

D(data) is "good enough".

If end to end security is also desired, the initial data can be encrypted with a key known only to A and B so that intermediate places in the network at the same level as A and B cannot access the plaintext of the message. Alternatively, intermediate places in the network can use limited functionality to pass information on without allowing it to be read even though it is in the plaintext form, as was noted earlier in our discussion. Limited functionality can only be assured in TCBs, and end to end encryption is still a good idea in cases where intermediate nodes may be taken over. This is examined in a later section, and will not be discussed further here.

This multiple encryption scheme has a potential benefit in that the more encryptions are performed, the more sure we might be of the security and integrity of the information. In some cryptosystems this is not necessarily the case. As an example, the DES cryptosystem has several keys that are self inverting or have an inverting dual, and even has at least one key that doesn't transform data at all. [16] This may not be bad since even the provably perfect "one time pad" [49] has such keys (with probability $1/2^n$ for an n bit message), but it's not encouraging either. A possibly desirable property of the cryptosystem for this application is that double encryption not reveal the data:

$$E(E(\text{data})) \neq \text{data},$$

and more generally, that n-ary encryption for n>0 not reveal the data:

$$E^n(\text{data}) \neq \text{data}.$$

In conjunction with the previous equations, this implies also that

$$D^n(\text{data}) \neq \text{data},$$

and in general can only be fulfilled in a cryptosystem in which

$$n \leq \text{number of unique ciphertext blocks}$$

since there can only be n unique representations when there are n unique ciphertext blocks. As a practical matter, the number of embedded encryptions required is unlikely to exceed 2^{32} for any contemporary or projected system, and the cryptosystem we will examine (the RSA [46]) can have sufficient numbers of unique ciphertext blocks ($\geq 2^{500}$ for a typical implementation) so that this is not a problem.

5.4 A "Good Enough" Cryptosystem

The major deficit of the stepwise encryption scheme is that it takes time for each cryptographic operation and may have severe key distribution and maintenance problems in some implementations. The major advantage is that it offers extremely good security even under fairly severe fault assumptions if a "good enough" cryptosystem can be found. Fortunately, there is at least one cryptosystem that fits enough of the requirements to make it usable in such a network.

Feasibility of the RSA

The RSA cryptosystem [46] encrypts and decrypts information by exponentiation in a modulus "M". Although there is no proof yet that it is, in general, difficult to determine plaintext from ciphertext, it is proven that determining either the enciphering or deciphering key from the other is as hard as factoring the product of two very large prime numbers. Even with (plaintext, ciphertext) pairs available to the cryptanalyst, determining keys is this difficult. Factoring primes has been studied for a very long time by many famous mathematicians, and no polynomial time algorithm has ever been found for it. This does not rule out the possibility that a fast enough factoring algorithm might be found in the future. The time taken for breaking the RSA system can be made arbitrarily long by using appropriately long keys. The use of longer keys doesn't change any aspect of protocols or other procedures except that it reduces the performance of the algorithms. Without going into mathematical details, we will outline the reasons that the RSA system meets all of the requirements for a "good enough" cryptosystem stated earlier.

Encryption and decryption under RSA are identical except in that they use different keys. The choice of which key is private and which is public is entirely arbitrary, and as such the RSA constitutes a "double" public key cryptosystem. Thus, if the RSA is "good enough", and every message is both encrypted with a public key and authenticated with a private key, then

$$S_{E(data)}=S_{data}\text{-}1,\ I_{E(data)}=I_{data}\text{-}1,$$
$$S_{D(data)}=S_{data}+1,\text{ and }I_{D(data)}=I_{data}+1$$

and if E(data) is "good enough", then D(data) is "good enough".

Because the product of the 2 keys used in RSA must be congruent to 1 in the modulus M in order to produce the plaintext from the ciphertext by double exponentiation, and since both must also be prime with respect to M, repeated exponentiation with either key must produce M-1 unique elements of the ciphertext space before repetition. This has been exploited in the generation of pseudorandom numbers [10] through repetitious exponentiation of an initial seed, but more importantly it shows that as long as $n<(M-1)$,

$$E^n(data)\neq data \text{ and } D^n(data)\neq data.$$

Since all of the protocols based on public key systems will work with any "good enough" public key system, and since RSA is a public key system, it can be used to implement any of the public key protocols. We conclude that RSA is "good enough" for the security requirements of a network if it is secure enough for the application under consideration.

Some Simple Network Protocols

There are also other advantages of public key systems that can be exploited in secure networks. A public key system requires only n key pairs for secure communications between n subjects (as opposed to n^2 keys for private key systems). This offers significant space savings over private key systems. Key pairs can easily be generated locally for spatial distribution of security. This limits the effectiveness of local attacks, and allows individuals to generate their own keys. Limited functionality systems that can not be infected or broken into without physical attack can be used for local key generation. In addition, the RSA can be used as a key distribution system for distributing keys of other cryptosystems with higher bandwidth or other advantages.

In order to obtain an end to end secure encryption channel between any two subjects (A and B) in a network where no previous secure channel existed, protocol 1 may be used:

```
Subject_A                    Subject_B
create an RSA key pair (E1,D1)
send E1 key to B
                                  create an RSA key pair (E2,D2)
                                  encrypt E2 with E1 => C1
                                  send C1 to A
decrypt C1 with D1 => E2
create an RSA key pair (E3,D3)
encrypt E3 with E2 => C2
send C2 to B
                                  decrypt C2 with D2 => E3
```

Protocol 1 - Secure Key Exchange in an Open Channel

After this exchange, only A and B can know E_2 because it was encoded with the public key to which only A has the private key. Similarly, only A and B can know E_3 because it was encoded with key E_2 to which only B has the private key. Therefore, no other subject can forge either A or B and no other subject can observe the plaintext data being sent between them. Thus we have both secrecy and authentication in both directions. The only problem is that the actual identities of A and B were never verified to each other. This problem may be solved with a sufficient authentication procedure, and will not be discussed further here.

This protocol needn't be used exclusively for end to end encryption, as it can be just as effective for exchanging keys of intermediate store and forward stations in the network without a centralized secure key distribution system. Indeed, the same concept can be used for introducing new sites and subjects into the system. Since each subject only needs to maintain the keys of the end to end subjects with which communication is desired, the space required for keys can be kept quite low. If a new subject is to be communicated with, the public key of that subject can be exchanged with all communicating subjects' public keys with only an addition of one key per subject. The number of keys maintained by each subject is thus linear in the number of subjects being communicated with.

The only problems with the RSA cryptosystem in this context are that it operates at a fairly low bandwidth (under 2000 bits/sec), and after a "long enough" time, any given key can be broken. The bandwidth problem is a fundamental limitation of the algorithm used to encipher and decipher information, and currently can only be improved upon through the parallel ciphering and deciphering of multiple blocks of data, and improved hardware technologies. This has limited application in centralized facilities, but is less likely to be useful for individual users. A realistic design could be implemented in a hand held device with 10 RSA chips that would allow communications at an effective baud rate of 20Kbaud with a .2 second delay between transmission and reception. Technological changes predicted for the next 10 years would allow such a system to be implemented using a single chip with a delay time under .01 seconds, and 20K baud bandwidth. This would seem adequate for a hand held or wristwatch mounted single user device.

The "eventual" breaking of the RSA appears to pose little or no threat to its practicality. The number of bits of key used for the RSA can be increased for a longer attack time, so if more security is desired, it can be attained at the cost of performance. Current estimates for attacking a 200 digit key using the best known algorithm on a special purpose computer are that, for the next 10 years, there will be no algorithm that will break a 200 digit RSA in under 10^{100} years. [10] 10^{100} years is much longer than the expected lifetime of the Universe, and appears to be an insignificant threat. In addition, new keys can be generated at frequent intervals to limit the damage of breaking a given key. With the use of a truly random number generator in each hand held device [10], a practically unbreakable key could be generated from a truly random seed as often as once every few minutes.

5.5 Fault Tolerant Network Security

The analysis to this point has been based on the assumption that every TCB within a secure network is perfectly trustworthy. Severe problems may arise when this assumption is dropped, and there is considerable reason to believe that

this assumption is not a reasonable one. As an example, if a single user were not trustworthy, if a single site in the network were secretly taken over by an attacker, or if a combination of errors or hardware failures were to occur, the security of the entire network might be compromised unless we considered the possibilities in our design. We examine the ramifications of such failures on the class of networks derived above, and explore techniques which could increase the fault tolerance of such a network and further secure it from attacks.

Fault Models

Our analysis of failures in a trusted computer network is based on two fault models. The first fault model assumes that some user in the network decides to launch an attack against the entire network and do as much damage as possible. A well placed traitor or terrorist might launch such an attack as might a disgruntled employee. We will see that without further restrictions on the network, such an attacker might cause fairly severe damage. This fault model will be called the "Lone Ranger Attack" (LR) throughout the remainder of this paper.

The second fault model considers the complete takeover of a computer or site in the network. We will use the word "node" from this point forward to designate a taken over portion of the network. This is a fairly severe type of fault since it allows all information including locally stored keys to cryptosystems to be attained and used by the attacker without the knowledge of the rest of the network. It is assumed that all access codes and access rights in the node are granted to the attacker, and that any activity that would normally be allowed in the node is allowed to the attacker. Examples of such a scenario are the case where a systems administrator at a site becomes untrusted or a successful physical attack is carried out without detection. This attack will be called the "Massive Takeover Attack" (MT) throughout the remainder of this paper.

Since we don't know enough about the topology of the particular network under consideration or the types of computers or protocols to be used in a particular case, we will assume that the network is designed to prevent such a failure from dominating communications. We will ignore all issues unrelated to the effects of the security model under consideration. We will also assume that in the MT attack, the node may introduce false messages, intercept messages passing through it, and allow information to cross security and integrity boundaries.

The LR Attack

In the LR attack, we consider the case where a single user at a given level launches a viral attack. Since a virus is, in general, able to reach the transitive closure of information flow, it could in theory spread throughout the network starting at its initial subject and infect all other subjects at the same level. This attack could eventually cause severe damage and widespread denial of services. This assumes that the transitive closure of information flow encompasses the vast majority of the other subjects in the network at the same level, and that no other isolation is in effect.

In the case of a UCB, we can see from the previous analysis that only a "one level" system is able to communicate information to the network. Thus, a multilevel UCB cannot be used to infect the network. In the case of an attack launched from a TCB or a single level UCB, information is allowed to flow to any other subject at the same level, and thus the attacker may launch a widespread viral attack. In practice, users are often granted access as more than one subject. In this case, a single user may be able to launch viral attacks at many or all levels and place a significant portion of the network under attack.

We know from our previous analysis that in order to further limit viral attack, we must either reduce functionality by limiting the interpretation of information, or further limit the sharing and transitivity of information flow. This applies to networks in the same way as it applies to a single system. Additional partitioning of the network into "compartments" can limit the sharing and transitivity of information flow and thus limit the subset of the network that could become infected in an LR attack.

Unfortunately, many systems currently implementing compartment based protection allow information flow across boundaries for subjects with access to multiple compartments. From the standpoint of viral attack, this is ill advised since

a virus could then cross compartment boundaries and spread to all subjects within the level at which the attack was launched regardless of its initial compartment. A rational solution is to enforce compartment boundaries to the same extent as levels are enforced, and thus limit a viral attack to all subjects in the same compartment, security level, and integrity level as the attacker. We find that this solution is unacceptable within a UCB since a UCB can't be relied on to protect compartments from one another, and we must further limit single level UCBs to one compartment if we are to accept outgoing communications from them.

In the same way as security and integrity levels became a problem in the transmission of data through intermediate computers in a network, the use of compartments presents a problem. Since the information allowed in an intermediate site cannot be in a compartment not permitted within that site, communications may be restricted from passing through intermediate nodes unless all nodes have all compartments. This also defeats the protection offered by compartments against MT attack soon to be explored.

Without extensive analysis, we can see that the use of cryptography for moving information between compartments works just as in moving information between security levels. The use of a special network compartment "N" allows us to transmit information through intermediate sites by giving all sites access to N. In order to avoid wide spread infection of N, we limit N's functionality to the built in functions required for implementing the transport mechanism of the network. If we can prove that this limited functionality doesn't permit viruses, then we may have an acceptable solution to this communication problem.

The MT Attack

In the MT attack, the security of the node is violated. All information in and capabilities of the node are then available to the attacker. With no compartment protection, infection can spread to any other place in the network at any level present within the node. If the node has access to all levels, then the entire network can be infected, and all information in the network can be extracted. This is certainly a severe attack, and is equivalent to having a set of LR attackers in each of the levels in the node.

Using the same analysis as was used for the LR attack, we see that with compartment protection, all (security, integrity, compartment) triples within the node can be taken over. Consider the MT attack's ramifications in terms of revealing keys to cryptosystems. The advantage of a public key system becomes quite apparent, since the node would only be able to access public keys of other sites. In a one key system like the DES, such an attack allows the attacker to forge messages of other sites unless n^2 keys are used for an "n" subject network. Security in the private key case requires severe overhead, especially when there are large numbers of subjects in the network.

5.6 Analysis of an Example Network

Figure 11 shows an example of a network operating with only level and compartment protection with many important network properties. The rows in this diagram indicate levels in the system, while the letters in each column represent the allowable compartments. The compartment 'N' is the "network" compartment realized through a TCB. Information can only be passed between levels through 'N', and a mandatory encryption and authentication is performed by the TCB. We may also allow a limited functionality computer mail system between 'N' compartments and grant every subject an account in an appropriate 'N' compartment for sending and receiving mail. For notational purposes, we will describe places in this network as triples consisting of the (TCB number, level number, compartment). Thus, (1,3,a) exists, but (1,3,c) does not.

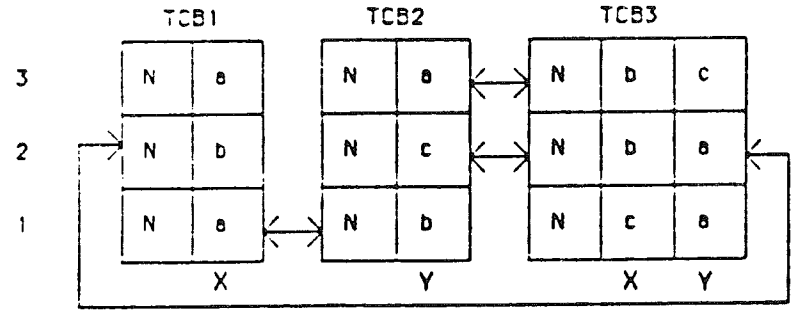


Figure 5.11 - A Sample TCB Network

Communication Restrictions

All connections in this network meet the requirements of our cookbook designs for connection of TCBs. Since communication links are at a variety of levels, there must be a variety of security measures taken to assure that links above the network level (1) are physically secured and only allowed to operate in trusted environments. Link X and Y are above the network level, and must be independently secured from the environment and each other. Thus we must require that TCB1 and TCB2 are in a site with trusted communication links. TCB3 can be in a remote site since its only connection is at the network level.

We shall use the term "channel" to indicate a logical communications link between two places in the network. Since no communications are allowed between subjects in different levels or compartments, the only channels required are:

```
channel  from    to
-------------------------
1        (1,3,a)<-->(2,3,a)
2        (1,2,b)<-->(3,2,b)
3        (1,1,a)<-->(3,1,a)
```

We will use a fixed slot routing technique with channels assigned to links in the following manner:

```
channel 1 uses 100% of X's time and 100% of Y's time.
channel 2 uses 50% of Z's time.
channel 3 uses 50% of Z's time.
```

In general, the channel assignment problem for optimizing communications relative to a performance measure in this type of system is NP-complete, and has very strong analogies to the routing problems encountered in the design of digital integrated circuits.

Communication Protocols

We will initiate each channel with a channel wide key exchange as specified in protocol 1 every hour. Both encryption and authentication of all messages over each channel will be required for each transmission. In order to prove identity of end to end subjects, each TCB will provide independent verification of identities of all senders on each transmission, and legitimate communication partners will be given to each TCB so that illicit attempts at initiating protocols may be detected.

Information will be transmitted as a continuous stream of bits at the link's optimal communication rate, with a synchronization signal sent once every minute to maintain network wide timing and synchronization. When higher communications bandwidth than can be provided with RSA is desired, systems will be able to agree via messages sent subsequent to protocol 1 to use a DES encryption system for the duration of the period of communication. The external appearance of the protocol will not change when the DES is in use as this could lead to a covert channel. DES keys will be exchanged using the current RSA keys, and will be randomly generated by the TCBs as part of their system services.

Fault Tolerance Under Attacks

The only network LR attacks are by subjects with channels to other network sites. Each of these can only attack 1/6 of the places in the network. With the exception of restricted computer mail facilities, no communication is permitted from any subject to more than 1/6 of the other subjects in the network. This network also provides limited protection from the MT attack in that TCB1 can only effect subjects in compartment 'a' at levels 1 and 3, and subjects in compartment 'b' at level 2, which is only 1/2 of the network. By similar analysis, TCB2 can only effect 1/3 of the network, and TCB3 can only effect 2/3 of the network. Note that the only untrusted communications line allowable in this system is the one from TCB1 to TCB3 since all others are at higher levels than the "network-level".

We finally note that in a network with a large number of UCBs and a small number of TCBs, we can attain distributed isolationism by using the TCBs as "hubs" for UCBs within a given facility, and routing all interfacility communications through these hubs. Limited functionality TCB hubs may be practical to this end.

5.7 Summary

The basic design criterion for a secure multilevel computer network have been examined, and a set of proven connectivity constraints have been developed that allow the systematic "cook book" design of secure computer networks in both trusted and untrusted communications environments. Untrusted computing bases have been shown to be of very limited utility in these systems, while trusted computing bases have been shown to be sufficient to allow useful communications.

Automatic declassification and reclassification of information in such a network was examined, and the desired properties of a cryptosystem for this purpose are now specified. A "good enough" cryptosystem has been shown to be available in the form of the RSA "public key" cryptosystem, and protocols are available for its proper use in such a computer network.

Attacks against secure computer networks of the sort specified here have been examined, and their effectiveness has been shown to be drastically reduced through the use of compartments as well as security and integrity levels.

The expansion of this work to encompass systems without aligned security and integrity levels involves about 9 times as many cases as the analysis presented here, but uses the same principals and mathematics, and is a straight forward extension of this work. A further extension of this work to the more general lattice structure is quite straight forward.

As an extension of the concepts of security levels, integrity levels, and compartments, there is no fundamental reason that an arbitrary dimensional space of security can not be used. The lattice structure goes a long way in this regard and allows a very flexible structure for restricting information flow. The idea of allowing users access to multiple places in the security lattice is a logical extension of allowing them access to multiple places in the more structured models. For extremely large networks, the management of this sort of policy might require significant software advances. As a first step, the automation of determining the worst case effects of the LR and MT attacks would seem straight forward, and would allow a very rough risk assessment as a precursor to administrative decision making.

Further work is required to derive actual designs of such a network, to finalize protocols for practical use, and to reduce this design to practice. With current cryptosystems, many secure network designs can be developed, but there may be some applications which require further cryptographic advances. Cryptography and cryptographic protocol analysis is being studied in the cryptographic community.

The use of a limited functionality network communications processor has been suggested, and implementations of are underway. [12] It is important that the results of this work be incorporated into the designs of networks using these processors, and that the designers of these processors consider the effects of the attacks examined herein.

It appears that the design of secure computer networks is feasible, and that with a significant development effort,

prototypes of the concepts derived here could be developed and tested. It is likely that within a few years secure multilevel networks will be operational and eventually will gain widespread acceptance in those communities with deep concerns for integrity and security.

6. Protection and Administration of Information Networks with Partial Orderings

We now extend the previous results in secure computer networks to a more general model, examine the effects of time on the protection and administration of information networks, and explore the implementation of provably secure automated administrative assitants for such networks.

6.1 Introduction

The "security" model of protection in a computer system was the first sound mathematical model of information flow that allowed proofs of mathematical properties to be used for establishing the security of a computer system. [3] The basic structure of this model is a linear relation on a set of "security levels" that is used to prove that information can only flow in one direction through levels, and thus to prove that information entering a "higher" security level cannot "leak" to a "lower" security level.

A generalization of the security model to a lattice structure was first introduced by Denning [20], who noted that the linear relation could be generalized to a lattice structure in which "higher" and "lower" in the security model are mapped into supremum (SUP) and infemum (INF) respectively in the lattice. This affords the same degree of assurance and mathematical soundness as the security model, and allows more general information flow structures to be used. The lattice facilitates more accurate modeling of many real world situations, most notably the situation where many different "compartments" may exist at the same security level without information flowing between them.

A very sound basis for limiting this generalization to a lattice structure is that, in any single processor, hardware has access to all information, and thus there is a SUP whether we like it or not. Although this policy seems suitable for a single processor where there is necessarily a SUP, in a more general network, there is no such physical restriction. We should be able to exploit this physical generality with a corresponding mathematical generalization.

At about the same time as the lattice model was produced, it was shown that the dual of the security model could be used to model the "integrity" of information in an information system. [5] The basic structure of this model is a linear relation on a set of "integrity levels" that is used to prove that information can only flow in one direction through those levels, and thus to prove that information in a "lower" integrity level cannot "corrupt" information in a "higher" integrity level.

In implementation, policies are most often modeled by the "subject/object" model in which each of a set of "subjects" has or does not have each of a set of "rights" to each of a set of "objects." [32] The "configuration" of the rights at any given moment are maintained in an "access matrix", and thus the rights of subjects to objects may be modified by modifying this matrix. By properly restricting the configurations to only those which fulfill a desired policy, we implement a provably secure system to meet the specified policy.

Figure 1 shows examples of the security and integrity models of information flow. In the security model, a subject at level "n" cannot read information from a level "i" s.t. i>n, or write information to a level "i" s.t. i<n. The former rule is called the "security-property", and the latter rule is called the "*-property". The security-property prevents a user from reading higher level information, and is commonly called "no read up". The *-property prevents a user from declassifying information, and is commonly called "no write down". The integrity model is simply the dual of the security model.

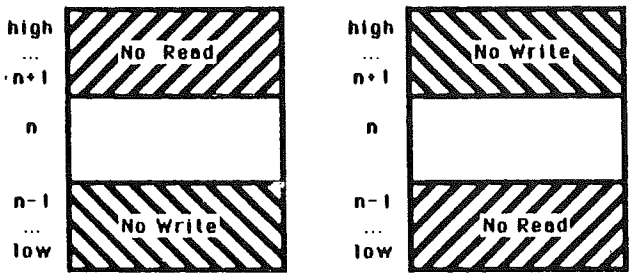
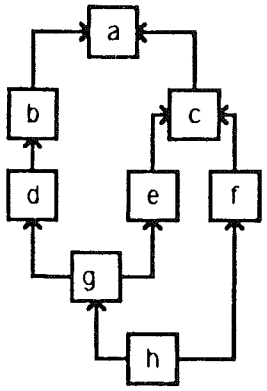


Figure 6.1 - The Security and Integrity Models

In figure 2, we show an example of a lattice based system and a corresponding access matrix. The generic rights in the access matrix for this example are read "r" and write "w", while subjects and objects correspond to places in the security lattice. We note in passing that the integrity model has not previously been extended to an integrity lattice (although this extension is immediately evident from the security lattice because of the duality of the integrity and security models). We may denote the relation "A can read B" by "A r B" and the relation "A can write B" by "A w B".



	a	b	c	d	e	f	g	h
a	rw	r	r	r	r	r	r	r
b	w	rw	-	r	-	-	r	r
c	w	-	rw	-	r	r	r	r
d	w	w	-	rw	-	-	r	r
e	w	-	w	-	rw	-	r	r
f	w	-	w	-	-	rw	-	r
g	w	w	w	w	w	-	rw	r
h	w	w	w	w	w	w	w	rw

Figure 6.2 - A Security Lattice and its Access Matrix

The formal rule for the security lattice policy is that a subject "S" may read an object "O" only if S is a security SUP of O, and S may write O only if S is a security INF of O. The formal rule for the integrity lattice is just the dual; S may read O only if S is an integrity INF of O, and S may write O only if S is an integrity SUP of O.

We note that because of the definitions given for the security model and the lattice model, there is no mechanism provided to prevent writing of higher level objects by lower level subjects. The lack of integrity restriction in the security model and the corresponding lack of security restriction in the integrity model, is often countered by the use of a "discretionary" access control policy which allows subjects control over rights not explicitly restricted by the security or integrity policy. [19] Although this may be of practical value in many cases, the only administratively enforceable restrictions on the flow of information are embodied in mandatory policies.

A next logical step might be to incorporate the integrity model restriction of "no write up" in the security model to allow information to be read from below, but not written to above. The problem with this policy is that an effective "write up" can be performed if there is ever a "read down", since the "read down" might allow a Trojan horse [27] to be placed at the higher level. The Trojan horse might read a particular low level object that describes objects to be read down, and thus effectively written up. In effect, we can generalize the "read" and "write" rights "r" and "w" to a single "flow" right "f" where:

(a f b) iff [(a w b) or (b r a)].

Preventing illicit dissemination and modification of information clearly calls for a policy that combines security and integrity. The combination of security and integrity policies of the sorts given above, results in the partitioning of a system into closed subsets under transitivity as we saw earlier. This partitioning is necessary in order to prevent global information flows.

6.2 Some Simple Demonstrations

We will now use access matrices to graphically demonstrate properties of interest to our studies. Although the solutions we show are for specific cases, they reveal general properties that are not necessarily self evident.

We begin with the matrix for the security and integrity models whose access conditions were stated earlier, and their combination in the case where security and integrity levels are identically divided. This is shown graphically in figure 3:

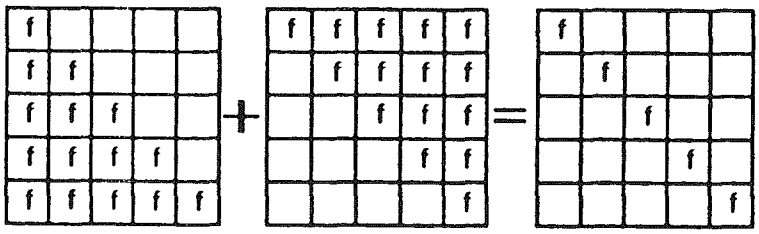


Figure 6.3 - Combining Security an Integrity Models

Another way to present this information may be used interchangeably when applicable, and the case from figure 3 is represented in figure 4. The property made clear by this example is that the combinations of the security and integrity models leads to a system that is closed under transitivity, and at best limits the spread of integrity corruption and/or security leaks to a closed subset of the system.

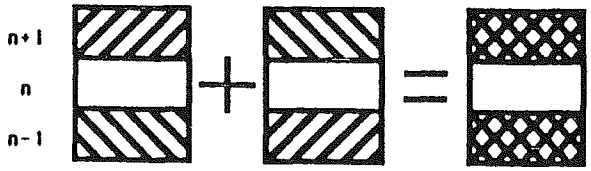


Figure 6.4 - Combined Security and Integrity Models

A similar analysis can be used to demonstrate that, if a security lattice is combined with an integrity lattice such that security and integrity relations are identically aligned, isolationism results. We show this for an example in figure 5 (the previous lattice example with subjects a, b, and d removed):

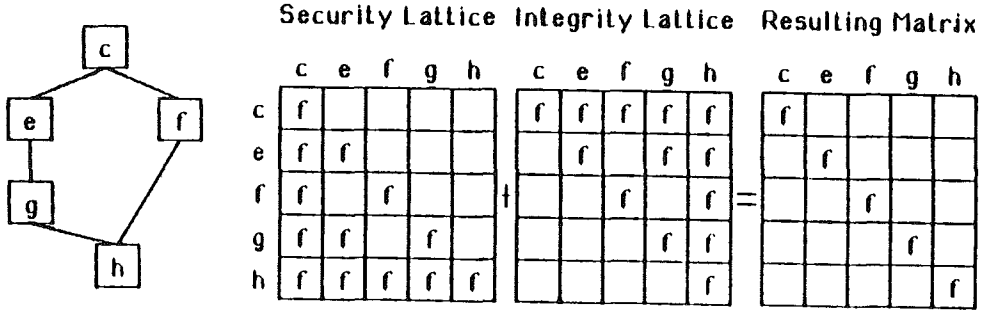


Figure 6.5 - Combined Security and Integrity Lattices

Cases where security and integrity levels are not aligned also tend towards isolationism as is shown in figure 6.

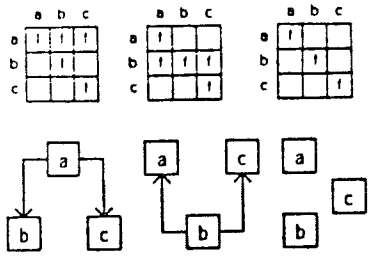


Figure 6.6 - Subject Combination

The "combination" of subjects, is a case where distinct subjects are combined from the point of view of the security or integrity policy as if they were a single subject. Thus any right given to one subject in a given model is automatically granted to the other. If we allow alignments to vary by combining sublattices of otherwise identical security and/or integrity structures, we achieve systems in which dissemination and corruption are limited to subsets of the system that are closed under transitivity. We show examples using the lattice from figure 5 above in figure 7 below, where e and f are combined in the integrity lattice, and where g and h are combined in the security lattice.

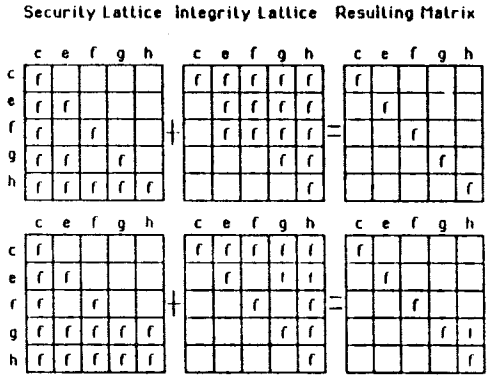


Figure 6.7 - Other Combined Lattices

Notice that in the former case, since e and f are incomparable in the security domain and have identical SUPs, no effect is achieved by combining their integrity. In the latter case, g is given flow access to h. The resultant structure may be shown as a directed graph as in figure 8.

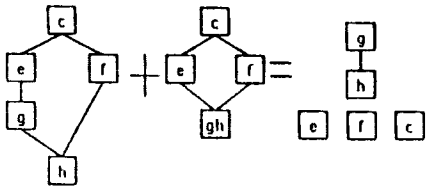


Figure 6.8 - The Resulting Network

We stated earlier that information can be communicated to the transitive closure of information flow starting at its initial source. Given an access matrix of the type shown above, we can compute an effective access matrix which tells us the potential information effects of subjects on other subjects under transitivity. A simple example is given in figure 9. This result is not likely to be predicted by a typical security administrator, and automated tools for evaluating access matrices to generate equivalent effective matrices may be quite useful. Efficient algorithms for this evaluation are not hard to find.

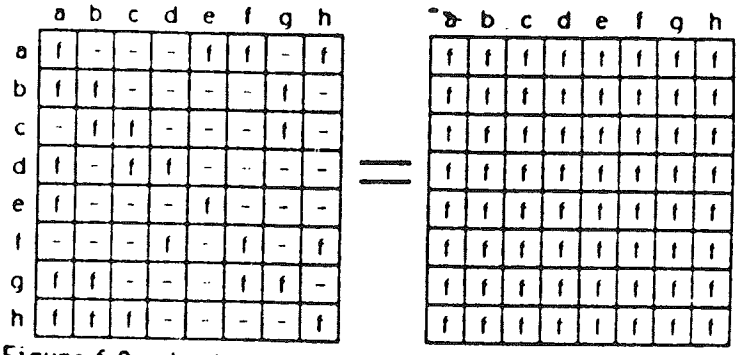


Figure 6.9 - An Access Matrix and Effective Equivalent

To see the above conclusion more clearly, we follow a simple series of steps as follows:

```
(a f a) and (a f e) and (a f f) and (a f h)     ;given
(h f b) and (h f c) and (f f d) and (b f g)     ;given
(a f h) and (h f b) => (a f b)                  ;conclusion
(a f h) and (h f c) => (a f c)                  ;conclusion
(a f f) and (f f d) => (a f d)                  ;conclusion
(a f b) and (b f g) => (a f g)                  ;conclusion
thus (a f *)                                   ;a flows to all
(a f a) and (b f a) and (d f a) and (e f a)     ;given
(g f a) and (h f a) and (c f b) and (f f d)     ;given
(c f b) and (b f a) => (c f a)                  ;conclusion
(f f d) and (d f a) => (f f a)                  ;conclusion
thus (* f a)                                   ;all flows to a
(* f a) and (a f *) => (* f *)                  ;global communication
```

We conclude from these demonstrations that the access matrix is a useful tool for evaluating the effect of simultaneously using a security and integrity policy, that the combination of these policies tends to partition systems into closed subsets under transitivity, and that the transitive nature of information flow has far ranging effects on the security and integrity provided by a protection system.

6.3 More General Mathematical Structures

We have just seen that the most general form of flow control allows so much freedom to an administrator that seemingly sensible policy decisions may have unexpected, and potentially catastrophic, effects on the actual protection provided. The mathematical structure of the security and integrity lattices guarantees that information flow is limited, and thus that inauspicious administration cannot cause global access as in the last example. Unfortunately, this combination tends to produce situations where isolationism results, and this may be too severe a restriction for desired levels of communication. Furthermore, within a given place in the lattice, we may desire additional flow limitation.

There are three basic remedies to the above situation. One remedy is to limit the functionality of the system so that information may not be used in a sufficiently general manner as to have transitive effects. This solution is infeasible for any general purpose machine, and little is known about the degree of limitation necessary to prevent transitive information effects. A second remedy is to limit the transitivity of information flow by keeping track of all subjects that have effects on objects and restricting certain sets of subjects from effecting certain sets of objects. This solution is difficult to implement, tends to move a system towards isolationism if imprecise implementations are used, and in order to be precise, requires an NP-complete implementation. The final remedy, and the one we will now consider, is to find a mathematical structure that is more general than lattices, and yet which maintains sufficient limitations on information flow to prevent the all consuming transitivity that arises in the most general case.

We begin by specifying the information flow relation "f". We assume transitivity of the flow relation, and thus that pairs (and sets) of subjects with mutual flow are equivalent. We collapse each equivalence class into a single subject, and get an antisymetric transitive binary algebra.

```
(S,{f}):
        (a f b) and (b f c) => (a f c)  ;transitive
        (a f b) and (b f a) => (a = b)  ;antisymetric
```

We note that in a structure where equivalence classes collapse, information in two non identical equivalence classes A and B can not be related so that ((A f B) and (B f A)) since this would make A and B identical by antisymetry. Furthermore, there can be no structure in which information flowing from A to B can reenter A since this would mean that (A f B) and (B f A) (by transitivity), and thus that A and B (and all other elements of this ring) are equivalent. Thus, we have a relation "<" such that A < B iff (A != B) and (A f B). We note that if there is a subject "b" so that not(b f b), then in all cases where there is a subject "a" so that a<b and a subject "c" so that b<c, we may eliminate subject b, and use instead, a<c. Thus we can systematically eliminate any such subject from the structure without changing the effective information flow behavior. We conclude that the structure of interest is a reflexive, transitive, antisymetric, binary relation, commonly called a partial ordering, and that this seems the most general structure we can use to guarantee restricted information flow. We will use the term "POset" to indicate a set whose elements are related by a partial ordering.

```
(S,{f}): for all a,b,c in S,
        [(a f a)                                ;reflexive
        and (a f b) and (b f c) => (a f c)      ;transitive
        and (a f b) and (b f a) => (a = b)]     ;antisymetric
```

Figure 10 exemplifies this structure graphically where flow is directed from left to right. Notice that the difference between this and previous structures is in the lack of a SUP or INF for each pair of subjects. For example; a and b have no INF, so no subject can effect both; j and k have no SUP, so they cannot both effect any other subject; g and c have no SUP and no INF, so no single subject can either effect both or be effected by both; and i and j have both a SUP and an INF, so that subjects a, b, e, d, and f can effect both i and j, and subjects p and q can be effected by both i and j.

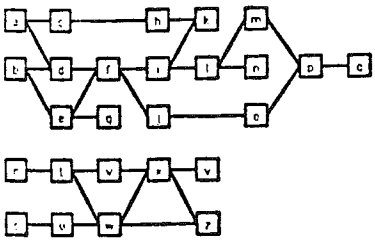


Figure 6.10 - An Example POset

We note here some of the results that can easily be attained from a POset by using figure 10 as an example. The effective POset under transitivity is formed by applying transitivity to information flow, and is more easily displayed in a matrix form. This answers the question of reachability immediately without undue complexity to the observer. We call the effective POset under transitivity a "Flow Control POset" (FCP). The FCP corresponding to a portion of figure 10 is given in figure 11 below. Subjects can always be labeled so as to produce an upper triangular FCP matrix since, if there is no reordering of a non upper triangular matrix to an upper triangular matrix, there must be two equivalent entries under our transitivity assumption. Every upper triangular boolean matrix maps into a unique POset, but not all upper triangular matrices map into a unique FCP. Finally, we note that completely independent subsets of a system can exist within a partial ordering as in figure 10, and that many distinct yet equivalent FCPs can thus exist.

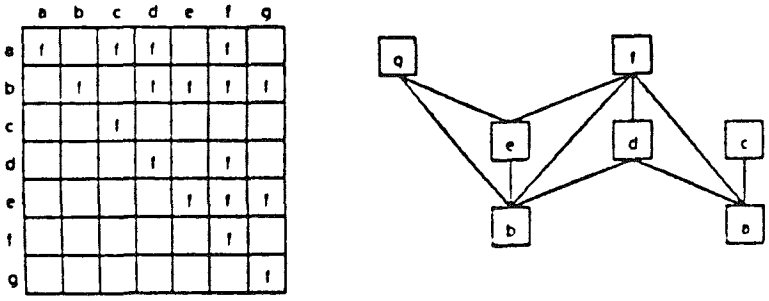


Figure 6.11 - An FCP Example

The corruptive effects of subject collusion can be easily determined by ORing rows of any set of colluding subjects to find their effective joint flow. As examples, the effects of; c, d, and g colluding; and of a and b colluding; are given in figure 12. We quickly see that a and b can collude to effect the entire example; while c, d, and g only have limited collusive effect. Similarly, the information accessible to a set of colluding parties can be derived by ORing the respective columns of the FCP matrix. We see that c, d, and g may collude to leak the vast majority of information in the system, while a and b only have trivial collusive effects in information leakage. This indicates a general and fairly obvious fact about systems of this sort; flow sources have corrupting power, while flow recipients have leakage power.

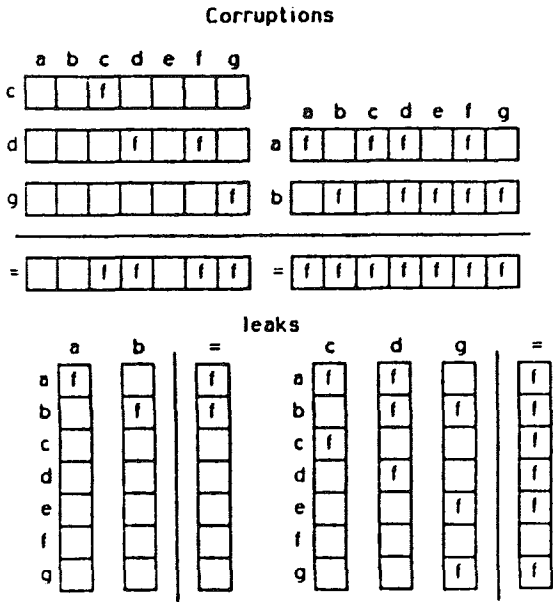


Figure 6.12 - Effects of Two Collusions

We note that the POset in this context is really a "classification scheme", just as the Bell-Lapadula and Biba models are classification schemes. We may, in practice, have equivalent subjects in an actual system, but we must be aware of the fact that they are in the same equivalence class from a flow standpoint, in order to understand the ramifications of the configuration.

6.4 The Effects of Time on Flow Control

We now consider the effects of time on the flow of information in the case where the configuration of a protection system may change through administrative action. We call an indivisible modification of a protection system a "move", and define a move as "valid" iff the resulting configuration passes some set of tests on configurations. Our analysis of moves begins with restrictions on tests for determining valid configurations. We examine three different time analyses of a system designed to enforce a flow policy. The "quasi-static case" is the case where only the configuration that results from a proposed move is of interest, and effects of previous configurations are unimportant. The "universal time case" is the case where effects of all past configurations are of interest to the validity of the proposed move. In this case, we are concerned with the lingering effects of corrupt information and/or the eventual dissemination of information. As a

compromise, the "window of time case" is the case where effects of a limited span of time are of interest to the validity of proposed moves.

We may implement our set of tests in any number of ways, but if we are to trust the system of tests as part of a trusted computing base, we should take care to design it in such a manner as to allow simple and straight forward proof of correctness. We choose a rule based system (RBS) which consists of a rule analysis method, an information base, and a set of rules which specify the desired tests. The basic algorithm we use for the RBS is; assume the proposed move; verify the validity of the resulting configuration by evaluating the rules; and accept or reject the move iff the rule evaluations are acceptable. Acceptable moves which are desired by the administrator may then be reflected in the access matrix.

We must be careful here, for there are several traps that the designer of such a system may fall into. For example, certain rule sets may tend towards specific states of the protection system, while others may prevent certain valid states from being reached from other valid states. In order for a set of rules to be of practical utility, we must restrict them in at least some basic ways. If the set of rules are inconsistent, we may never find all rules in agreement, and thus no modification will be acceptable. If the rules are incomplete, we may have cases where rules cannot produce a result, and this is clearly unacceptable. We restrict ourselves to a finite set of rules since an infinite set of rules cannot be evaluated in finite time. Similarly, each rule must be decidable so that decisions are made in finite time. Finally, we require that the rules reflect the desired policy of the protection system, for if they do not, they are of little use. We note that many desirable policies are in practice unattainable, and that we must restrict ourselves to attainable goals if we wish to attain them.

Since the validation process consists of testing the resulting configuration against the set of rules in force, any move that violates no rule will be accepted, and any move that violates any rule will be rejected. Since an RBS can be quite simple in design and implementation, it should be relatively easy to prove its correctness using automated theorem proof techniques already used for proving correctness of secure operating systems. Once a basic RBS has been proven correct, we need only prove that rules are correct for a given policy in order to prove a given implementation correct. Security, integrity, and other properties of results are proven by proving that evaluations performed by rules in the RBS are mathematically consistent with the specified policy. Since the rules for these policies and the rules for the RBS are just mathematical conditions, this mapping should be quite simple.

Given that we have a provably correct RBS, we must select rules and analytical techniques. We now examine the effects of particular choices of rules on the accuracy of our results.

Consider the quasi-static case, wherein we simply use a set of rules which test the state of the access matrix resulting from the proposed move. The problem with this case is that there is a sequence of independently valid moves, which inadvertently allow information to flow where it should not. As an example, with the rules (C ~f B) and (B ~f C), users B and C may communicate as follows:

```
(B f A)        ;information may flow from B to A
...            ;and does as time passes
(B ~f A)       ;B may no longer flow to A
(A f C)        ;information may now flow from A to C
...            ;B's information transits to C
```

We can see that if (B f A) and (A f C) were simultaneously true, an FCP computation would determine (B f C) from transitivity, and thus a move that created this situation would be dissallowed because of the rule (B ~f C). If we only examine the static configuration, there is no move that causes (B f C) to be instantaneously present in the FCP, and thus the sequence will be wrongly considered valid. This problem comes from the effect of time on information flow.

As an attempted solution, we can simply ignore the removal of flows in the evaluation process. This scheme, in effect, remembers all previous flows, and only permits flow if there is no historical flow that, when combined with the proposed flow, results in illicit flow. Unfortunately, this solution is imprecise, in that there are legitimate moves, even in light of historical information, that will be considered invalid if we simply ignore all flow cancellations. An example is a sequence of moves as follows:

```
(A f C)          ;information may flow from A to C
...              ;and does as time passes
(A ~f C)         ;A may no longer flow to C
(B f A)          ;information may now flow from B to A
...              ;and does as time passes
```

In this example, even though (A f C) and (B f A) are illegal together, there is no sequence of events whereby information can ever flow from B to C or from C to B, and thus neither flow rule is violated.

We see that the actual sequence of moves must be considered if we are to precisely prevent illicit flows over time. To precisely track the time transitivity of information flow, we must precisely track all effects of information from subject to object, and this has been proven NP-complete for both the security and integrity cases. We can, however, obtain a precise solution, if we assume that any flow that can happen will happen (a conservative assumption in the flavor of Murphy's law).

In order to precisely determine the largest set of subjects which can effect a given object, we assume an initial configuration of the protection system, and maintain a precise configuration that reflects the maximum set of subjects that could have effected each object after each move. We call this configuration a "time flow configuration" (TFC), and calculate it by remembering all transitive flows into each object for all moves as follows:

```
TFC at move 0 = FCP
for N>0, TFC at move N "(A f B)" =
   1    for all X,Y s.t. TFC(X,Y) at move N-1 => TFC(X,Y) at move N
   2    for all X s.t. TFC(X,A) at move N-1 => TFC(X,B) at move N
   3    TFC(A,B) at move N
   4    for all X s.t. FCP(B,X) at move N,
               for all Y s.t. TFC(Y,B) at move N => TFC(Y,X) at move N
```

We may recall that an FCP is a one directional flow relation on a (subject,object) pair. A TFC is the same sort of relation. Our initial TFC is just the FCP of the initial configuration, since this indicates all potential flows into each object from each subject under transitivity. From this point forward, every move "(A f B)", introduces the possibility that a previous information flow to A transits to B and all objects in the transitive closure of B's information flow. Rule 1 states that previous flows remain after a move. Rule 2 states that all previous flows into A are added to B. Rule 3 states that A is added to the flows into B. Rule 4 states that all resulting flows into B are added to all objects in the transitive closure of flow from B. Rule 3 is implied by TFC(X,A) => TFC(X,B) if we assume (A f A).

Except for the FCP, this maintenance of the TFC takes at most N↑2 time and space in the number of subjects, and is linear in the number of moves considered. The FCP computation takes at most N↑2 time and space in the number of subjects, and is performed only once per TFC calculation. It is thus quite feasible maintain the TFC throughout the lifetime of a typical network.

One problem with using the TFC for limiting moves is that it may become unduly restrictive as time goes on. Information aging, for example, is commonly used to justify automatic declassification of information, and a corresponding policy might be used to justify automatic removal of TFC flow restrictions. A "window of time" version of a TFC can be generated by assuming that the initial configuration of the system is the FCP configuration at the beginning of the window of time, and computing the TFC using all subsequent moves. We must of course remember all historical moves over the window of time, and must keep either historical configurations or a complete sequence of historical moves from which we can recompute the FCP for the beginning of the window.

Additional uses arise if we wish to maintain a precise accounting of the potential effects of collusions over a given time span. As an example, suppose we know that a given collusion was in effect over a given span of time, and wish to compute the maximum integrity corruption and security leakage that could have resulted from that collusion. We may compute these effects by the following procedure:

```
get the FCP at the time of first collusion
compute the TFC till the end of the collusion
maximal corruption = all X s.t. for any Y in collusion, TFC(Y,X)
maximal leakage = all X s.t. for any Y in collusion, TFC(X,Y)
```

6.5 Automatic Administrative Assistance

By using the above mathematical basis, we can automatically evaluate the FCP, TFC, equivalencies of subjects, and effects of collusions under a given configuration of a protection system with a flow relation. We may augment this basic capability with a set of rules that determine whether a given configuration is allowable given installation dependent parameters, to form a configuration evaluator tailored for a given system. We may form a dynamic analysis system by performing evaluations on configurations resulting from proposed moves, and reporting on the effects. Finally, we may augment this capability with a set of inductive rules for proposing moves that are likely to be acceptable to the protection system while fulfilling desired information flow requests. Figure 13 shows the architecture of such an RBS.

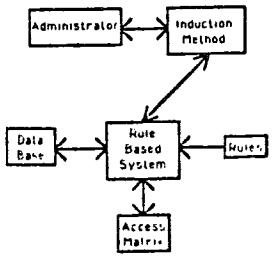


Figure 6.13 - Architecture of an Automated Assistant

In a network where classical protection models are required, we may form an assistant based on the security and integrity models. We use the mathematical restrictions on communications under these models as the rules for evaluation of configurations. A configuration is acceptable only if these rules are not violated. Rules for evaluation of collusions, limiting FCPs and TFCs, and limiting equivalencies of subjects can be used to form more restrictive systems while still maintaining security and integrity constraints. We assure that added rules do not allow violation of previous rules by using the union of rule evaluations for evaluating proposed moves. Since rules themselves may contain complex conditionals, we lose no generality in this forced union.

Since inductive decision making is submitted to the RBS for acceptance, we need not trust the induction method, nor prove its correctness in order to be certain that we make no illicit moves. Indeed, we can design high level structures to generate a multitude of suggestions, have these suggestions submitted to the RBS, and use the results of evaluation to determine the utility of inductive paths and filter out invalid administrative suggestions.

A simple implementation of an assistant that maintains security, integrity, and compartments, while allowing arbitrary information flow controls within those restrictions, may be formed by implementing the following moves and using the previously explored techniques to validate resulting configurations:

To add an individual, we require that the minimum and maximum security and integrity levels, and the set of compartments are within system limits.

```
Add-individual A (min-sec,max-sec,min-int,max-int,effect,comp,comp,...):
        Min Sec A >= Min Sec System
        Max Sec A <= Max Sec System
        Min Int A >= Min Int System
        Max Int A <= Max Int System
        Comp A SUBSET Comp System
```

To add a given ID for individual A, we need to know the individual, the compartment, the security level, and the integrity level for the given ID, and must verify that these don't cause the configuration to go beyond the allowable constraints on the individual.

```
Add-ID Ax (sec,int,comp):
        Min Sec A <= Sec Ax <= Max Sec A
        Min Int A <= Int Ax <= Max Int A
        Comp Ax ELEMENT Comp A
```

To add an information flow from ID Ax to ID By, we must verify that the flow doesn't violate security, integrity, or compartment constraints:

```
Add-flow (Ax f By):
        Sec Ax <= Sec By
        Int By <= Int Ax
        Comp Ax = Comp By
```

In order to remove flows, IDs, or individuals, we must verify that these removals don't cause other rules to be violated. In terms of the ability to produce valid configurations, removal has an immediate benefit. With only security, integrity, and compartment constraints, a sequence of moves is valid iff each move in the sequence is valid. We are also guaranteed that any valid configuration of the protection system can be reached from any other valid configuration with only these moves.

Note that an ID or individual should really never be removed as it is sufficient to remove all relevant information flows. A good reason for not allowing individual names or IDs to be reused, lies in the information aging problem. The reuse of an old ID by another individual, might cause a naming conflict that would introduce uncertainty in the decision making process. Removal, subsequent reuse by another individual, removal, and reuse by the original individual might cause a condition where traces of the original flow effects are lost while the actual informational effects allow illicit flows. A rational use of the window of time analysis is for allowing reuse of old IDs.

Although considerable mathematical work is still required to investigate underlying policy issues for static and dynamic configurations of protection systems, a simple automated administrative assistant of the sort shown above is a significant step towards eliminating errors in the administration and configuration of information networks. An assistant of this sort has been prototyped, and further developments along these lines are expected to include hierarchical protection systems and administration.

6.6 Summary, Conclusions, and Further Work

We have shown by a series of arguments that the structure of preference for describing and the analyzing flow properties of information networks is the POset. We have demonstrated a difficulty with more general structures in that they obscure the ramifications of administrative decisions, and an inadequacy of less general structures for describing many desired situations. A design for a provably correct automated administrative assistant has been shown, and a set of moves for maintaining traditional policies have been given.

The effects of transitivity, collusions, and time on the protection provided by flow control have been examined, and a variety of analytical techniques have been introduced for implementing accurate flow control protection in the presence of various time variant assumptions.

Extensions of these techniques can be used to consider the effects of collusions that change over time and sets of independent collusions. Similar analysis may also have implications to other domains such as game theory and its many related fields.

One particular extension allows us to measure the effects of discretionary access control. In order to include this in our analysis of the TFC, we need merely include discretionary moves in our TFC computation. This grants us a more accurate model of the actual behavior of a network, and assuming that discretionary access control operates correctly, yields provably valid results.

A logical extension of this work is the analysis of systems where a hierarchy of administrators exist. In this extension, the discretionary controls of a SUP administrator are mandatory controls of an INF administrator. The analysis of valid moves over time for each level in the hierarchy enforces mandatory policies at that level. Information on actual configurations may be used by SUP administrators to allow more accurate configuration control at the global level, while local controls allow better distribution of responsibility. It is likely that this work will be extended to include special purpose security and integrity transforms which allow distributed decision making.

Another extension of these ideas is in the case where we assume that information flow is not instantaneous or that transitivity is limited in some manner by the operating system. In the case where information flow takes time, we can associate a "flow speed" constraint that tells us how quickly flows may occur. The effect on our previous analysis is simply to limit the transitivity of information flow as a function of the time over which information is available and the flow speed. Although the analysis in this case is somewhat complex, the mathematics follows directly from what we seen herein, and the TFC computation is not significantly complicated. In the case of limited transitivity, we must simply restrict our transitive closure assumption to a finite rather than infinite number of flow steps. The basic mathematical structure changes slightly because we no longer have the ability to equivacate subjects with mutual flow, even after a delay as we can in the limited flow speed case.

There are many applications of this work in a wide variety of domains. In the design and analysis of secure computer systems, this work is a logical extension of the works cited in the introduction. In the domain of industrial and international espionage, analysis of this sort is likely to provide insight into the potential effects of leaks and misinformation, and the effectiveness of techniques which attempt to limit, detect, or compensate for these activities. Extensions to limited flow speed systems will likely yield results of interest to those who spread and attempt to quell rumors, to those who attempt to analyze the effects of infectious diseases, and to those who examine the effects of information on the society.

The techniques presented here allow improved analysis of exposures to informational losses, which is critical to both protection and insurance of informational assets. This sort of flow analysis may also be helpful for optimizing behavior of information networks for communication with privacy and integrity.

In the broader sense, we feel compelled to consider the relation of this work to similar work in protection of materials in process control and materials handling. At the most fundamental level, there is a difference between information and physical materials, in that physical material falls under a conservation law, while information does not. In essence, when we "leak" physical entities, there is a corresponding reduction in mass from the source of the leak. Similarly, when we "corrupt" physical entities by introducing foreign substances, there is a corresponding increase in mass. When information moves through an information system, we have no such conservative metric with which to measure the effect.

7. Detection and Cure of Computer Viruses

Since prevention of computer viruses may be infeasible if widespread sharing is desired, and since sharing is often considered a necessity in modern computer systems, the biological analogy leads us to the possibility of detection and cure as a means of viral defense. We now examine the potential for detection and removal of viruses.

7.1 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 for any decision procedure "D", P could invoke 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 contradiction 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. We note that this proof differs slightly in presentation from the previous proof (Thm 6) of this fact, and refer the skeptical reader to that proof for self assurance.

7.2 Evolutions of a Virus

As we pointed out in our earlier discussions, we can create evolutionary viruses by forming viral sets such that each virus evolves into another element of the set. 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, determination 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. Again the skeptical reader may refer to Lemma 6.1 for further assurance of these facts.

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 a virus. It is a program that 'infects' another program by modifying it to include an evolved version of itself. Since the viral capability is in all general purpose 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 being able to decide whether or not a compiler is a virus by its behavior leads directly to the determination of whether or not the input describes a virus, and thus whether it is a virus by virtue of its appearance. Since precise detection by behavior in this case leads to precise detection by appearance, and since we have already shown that precise detection by appearance is undecidable, it follows that precise detection by behavior is also undecidable.

7.3 Limited Viral Protection

A limited form of virus has been designed [52] in the form of a special version of the C compiler that can detect the compilation of the UNIX login program and add a Trojan horse that lets the author login. Thus the author could access any Unix system with this compiler. The compiler contains a virus that 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 C compiler sufficiently different from the original as to make their equivalence very difficult to determine. If the "best program of the day" would be incapable of detecting their equivalence in a given amount of time, and the compiler performs its task in less than that much time, it could be reasonably assumed that the virus could not have detected the equivalence, and therefor would not have propagated itself. If the exact nature of the detection were known, it would likely be quite simple to work around without going to this extreme. Once a "clean" version of the C compiler exists, the login program can be recompiled for renewed security, and a "clean" version of the original C compiler can also be recompiled if desired.

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 V at the beginning 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"

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 replaced 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 can't be detected, and no defensive 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 subset of the programs within a system, while a given protection scheme could only protect against a given subset of the viruses. If each user and attacker uses identical defenses and viruses, there might 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 don'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 [17], and the spread of viruses through systems might well be analyzed by using mathematical models used in the study of infectious diseases. [2] We note here that although "survival of the fittest" may not be the desired mode of operation in modern computers, it appears inevitable in biological systems, and may also be inevitable as computer systems advance.

7.4 Imprecise Behavioral Detection

Since we cannot precisely detect a virus, we are left with the problem of defining potentially illegitimate use in a decidable and 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. As an example, 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. A thresholding scheme could adapt so the threshold could not be easily determined by the attacker. This "game" can clearly be played back and forth. We note that as the threshold for detection is lowered, larger and larger percentages of legitimate programs will be detected as potential viruses. Since each potential virus must be examined for legitimacy, and since the threshold potentially becomes lower and lower as more detection is desired, in the end we reach the situation where virtually every program in the system must be verified. If we are to verify every program in the system before use, we might as well forget the thresholding scheme altogether.

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.

7.5 Removal

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 subset of the subjects and cure them without denying independent services to other subjects.

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.

We note that at least one large class of viruses is, in practice, easily detected and removed. This is the class of nonevolutionary viruses. If we have a static virus which is spreading throughout a system, we can clearly detect it by looking for identical sequences in many programs in the system. If we detect a large number of identical sequences of sufficient length as to make them highly unlikely through accidental modification, and if we can verify that these sequences are not normally generated by legitimate programs (such as compilers), we have strong grounds for suspecting the presence of a virus. Once the identification as a virus has been established, it can be systematically hunted down, and infected programs removed. We note that even a static virus may not be easily detected and removed, and that this method is by no means foolproof.

7.6 Spontaneously Generated Viruses

One concern that has been expressed and is easily laid to rest is the chance that a dangerous virus could be spontaneously generated on a real system. This is strongly related to the question of how long it will take N monkeys at N keyboards to create a virus, and is thus laid to rest without further attention except to note that the presence of such a virus, likely indicates a purposeful source rather than an accidental one.

8. A Complexity Based Integrity Maintenance Mechanism

In a system with multiple users, shared information, and general purpose functionality, integrity corruption by viruses and other integrity corrupting mechanisms is possible. Since this sort of functionality is generally considered useful, it is desirable to find a means by which the integrity of information may be maintained when these properties are not restricted.

We now examine a method of "self defense" in which each program attempts to protect itself (and perhaps other information) by using self knowledge to detect illicit modification. It is likely that if timely detection is possible, redundancy (e.g. backup tapes) may be used to correct corruption.

8.1 The General Method

The basic idea is to cause the complexity of finding a systematic way to create undetected corruption to be very high, and the probability of causing such a corruption to be very low.

Our general method is to use a large set of self test techniques, which can be placed in a large number of ways throughout a system, and which rely on a difficult to forge cryptographic checksum for detecting illicit modification, while still allowing legitimate modification. The argument for this general method is as follows:

- If there are a large enough class of such tests, then the complexity of determining whether or not a given portion of information is such a test may be very difficult, perhaps even undecidable.
- If these tests can be placed throughout the system in a sufficiently variable number of ways then it may be very hard to determine where or how they have been placed, and thus a very large number of places may have to be searched in order to locate them. When this is used in conjunction with making the tests difficult to recognize, preventing the tests from acting may be made quite difficult.
- Even if the tests are active, there is no guarantee that the information they test cannot be illicitly modified in such a manner as to be undetected by these tests. To prevent such undetected modification, an appropriate cryptographic checksum may be used to cause the probability of a modification resulting in a valid checksum to be arbitrarily small.
- In order to have a useful system of storing and retrieving information, we must allow legitimate modification. We do this by allowing legitimate modification only by self testing programs. This results in a partial ordering of integrity testing interdependency.

The remaining problem is to find a mathematically justifiable technique that fits all of these criterion.

8.2 Fundamental Limitations

Before suggesting a specific method, we wish to consider the fundamental limitations inherent to the suggested general method. In the cases of finding classes of tests and adequate cryptosystems, the problems are not uncircumventable, as we will see later in this chapter. In the case of test placement, there seem to be some rather severe problems. The problem of self test in a system that allows legitimate modification appears to be difficult as well.

The Class of Tests

Commonalities in tests might be exploited to try to detect the presence of a test in a given location. We want a sufficiently large set of tests which can be stored in a sufficiently large number of forms to make detection sufficiently

hard. A technique that makes test detection undecidable would be very nice, but we might be willing to settle for less. Note that nearly any commonality may be used for detection since the probability of a given sequence being found in a random other program decreases very rapidly with the length of the sequence. This is clear from information analysis of software in both source and compiled form, but need not be the case.

The Placement of Tests

Tests can be placed anywhere in the system where they will:

1. be executed often enough to reduce the probability of a corruption spreading transitively to an acceptable level.

2. not corrupt the integrity of the system by their presence.

If we place these tests in areas that are not interpreted as program, but rather as data, they will likely never be executed and result in the corruption of data. It is therefor important that they be placed in interpreted information and that they act independently from the state information used in normal activities.

Unless we partition the information being used as data from that being used as program, we cannot guarantee that a program will not examine its own contents and or modify itself in the course of its legitimate behavior. If we try to partition data from program, we cannot be guaranteed that we will be successful unless we restrict the system's functionality, for with general purpose functionality, there is no distinction between information used as program and information used as data except in its interpretation. This is most clearly seen in the case of an interpreter (such as Basic) which allows information modified as data by an editor to be used as program when interpreted by the Basic interpreter.

This would seem to imply that placement depends upon knowledge of the intended use of information, and that general purpose programs cannot be perfectly protected. Since any general purpose program "P" can be made to act like a Turing machine, any data "D" entered by the user can be interpreted by P as a program. Since we cannot rely on "D" to preserve the integrity of its own data, we probably cannot do any better than to protect programs and data which cooperate with the scheme.

We may require that data which is to be modified with integrity must be modified by one of a given set of programs. We may be able to design a compiler that forces checks on the integrity of data files as well as the set of programs able to legitimately access them. The only remaining problem is the placement of these checks within programs.

If we place tests in the beginning of programs, or at any standard place, they may be easily circumvented by appropriate modification of the code which tests for integrity. An alternative is placement at an arbitrary place, or perhaps more appropriately at one or many of a large set of places within a program. Since determining which section of code is the test may be made arbitrarily difficult, this offers some hope, but we must also consider that the placement of this code such that it is not executed in every use of the program, reduces the probability of detecting a corruption before it spreads transitively, to that of executing the detection algorithm.

The placement of the code in each branch of a program may be quite cumbersome, and it guarantees an attacker that some test is placed in every branch. This may or may not be of aide to the attacker, and may or may not be so burdensome as to make the system impractical. Another alternative is to evolve the program so as to include the test, or to evolve the test so as to include the program. In any case, the evolution of programs in this way has received little attention in the literature, but it appears from our previous discussion that this technique is both feasible and difficult to disentangle.

The Cryptographic Checksum

The best we can do in a system which protects itself with complexity is make the probability of forgery and the difficulty of breaking the code in a given amount of time arbitrarily low. We do this by using a "one way" function which allows us to transform into the cryptographic checksum in order to test the program for modifications, but which doesn't

allow us to generate a program that produces a valid checksum. We must be careful that the function is not only one way, but that there are a sufficiently large number of keys available, and that the key used for generating the checksum cannot be used to invert the function.

We suggest a "public key" cryptosystem in which the private key is destroyed unrecoverably at the creation of the checksum. This prevents the possibility of finding that key and using it to generate a new and valid checksum for an invalid program. It also allows us to leave the key publicly accessible (although hidden along with the rest of the self test code) without fear of its eventual discovery and exploitation.

Modifiability

Let us now suppose that a legitimate program legitimately modifies information in a data file associated with one other legitimate program. In order for this change to be considered legitimate by other programs, each must be convinced of the legitimacy of the program making the modification and of their own legitimacy. If other programs are to access the data, each must modify its self to reflect changes in data files. Since each has now been modified, each must verify that the others' modification was legitimate, and must again modify its self to reflect the new modification of each other. Since they test each other, this procedure must be repeated until either a stability point is reached or indefinitely.

If a stability point is reached, this means that a modification in one of the programs does not require a change to its cryptographic checksum, and thus the checksum for both the legitimate and illegitimate versions are identical. If it is extremely unlikely for this to happen, this will only happen after a very long time if at all, and if it is likely, than it is also likely that an attacker's change would be thought legitimate. What this seems to indicate is that a strict limitation of the testing of programs and data by each other must be enforced in that we must not form a loop of interprogram tests.

In other words, if all programs are modifiable, at least one program must have sole responsibility for testing itself, and all other related programs must only perform tests on each other in a semi-lattice form with the self testing program at the "sup". This can be relaxed if we limit the legitimately modifiable portions of the system so that their modification is supervised by legitimately unmodifiable programs. Unmodifiable programs can test each other with mutual testing loops.

In cases where programs do not share modifiable data with other programs, data may also be tested. For cases where sharing of data is important, we can use a single data access program which is tested by all sharing parties, and which has complete control over the modification of all shared data. This program can then use internal tests on all stored data, and thus shared data can be tested without the looping problem. The resulting mathematical structure is a partial ordering with shared data residing only in semi-lattice substructures. For high assurance, increased mutual testing may be used.

8.3 A Specific Method

A specific method specifies a class of tests, a means by which they may be placed throughout the system, a checksumming method, and a modification method, all satisfying the above criteria.

The Class of Tests

An arbitrarily large number of programs can be written to generate and compare a given set of data with a stored value by starting with a fairly simple evolutionary program, and creating a large number of evolutions. It is in general undecidable to determine whether or not two evolutions are equivalent. This seems a promising leaping off point for automatically developing a set of tests from a single test. If additional safety is desired, a large number of versions of the self test algorithm may be used in conjunction with evolution to guarantee that even if a given case were thoroughly broken, other cases would exist.

An intriguing variation on this theme for use with the RSA [46] cryptosystem, is the generation of a special purpose exponentiation algorithm for each of a large number of RSA keys. Since each exponentiation produces a slightly different algorithm [38], each test program will be different. This can of course be augmented by the use of evolutionary techniques

to make each version of the test very difficult to detect. In addition, this prevents attacks in which the checksum for a given set of information is performed by the attacker, is searched for in the machine state, and is modified to fit the desired checksum for corrupt information. Since an attacker cannot easily determine what information belongs to the test program, and the key itself isn't even stored (only an algorithm for computing the effect of its use is actually kept), there is no known way to tell which key is being used.

The Placement of Tests

We suggest a lattice structure of testability in which all programs test themselves, and some programs test each other. When information must be modified or shared, we suggest an independent program through which all modification must be performed, and which is an 'inf' to all programs with access to the shared data, and a 'sup' to all data shared by them. This allows each program to independently verify the propriety of the modification program.

One placement of tests is done by a special purpose compiler which has sufficient knowledge about the programs to allow a relatively small number of tests to be placed at any of a relatively large combination of places within the program. Programs will likely have to be restricted in some ways (e.g. no self modification), and all data files used by programs and all sharing behavior will have to be specified at compile time.

A second test placement strategy is the generation of a test algorithm, and the incorporation of the program to be tested along with a number of irrelevant sequences of instructions within it. The value of the resulting checksum is computed based on all but the final checksum value, and this value is placed in a location determined at test generation time. Since each test algorithm is different (below), each program will have a differently placed checksum. Additional code strands may make it difficult to disentangle independent subsequences of the resulting code into test procedure and program.

Although specific algorithms are not yet available for this purpose, their development appears straight forward from previous work in evolutionary programs.

The Cryptographic Checksum

The following cryptographic protocol for creating difficult to forge checksums appears to be sufficient for the desired conditions.

1. Generate a key pair for the RSA cryptosystem, and destroy the private key.

2. Use the public key to encrypt each block of information to be checksummed along with the block number.

3. XOR all of the encrypted blocks to form a cryptographic checksum of the desired information.

Note that since the inverse function is not available, it is infeasible to attempt to generate blocks of plaintext which correctly checksum to any given value. This prevents the attack where a forger forms any desired number of blocks of arbitrary information, encrypts each with the known public key, determines what the last block must checksum to in order to make the final checksum come out right, and then generates a block which checksums to the appropriate value to compensate for the forged blocks' incorrect values.

8.4 A Simple Variation for Software Protection

The above technique is quite complex, may suffer from poor performance, and may leave a lot to be desired in the general case. In the domain of software protection, a major difficulty is preventing modification of a program for resale under a different name. This simplified variation resolves much of the complexity of test placement within a program by distributing the integrity protection throughout the program so that each routine protects itself from both analysis and modification.

The basic idea is to encode each subroutine so that only it knows how to decode itself into a standard memory area. Since each routine can be made sequential and all execution strands can be kept track of for small enough program segments, the placement of tests within a routine may be made reasonable, and tests may be interleaved with program. When a subroutine is called, it decodes itself into a standard memory area, thus overwriting the previously decoded subroutine in that area. Data shared by subroutines may be decoded once at initialization, and stored in a common area for manipulation.

Since only a small portion of the program is in plaintext at any given moment, many "snapshots" must be taken in order to expose a significant amount of the program. Since each routine is designed to run in the same memory locations, absolute addressing is possible, and relocation of the program thus causes operation to fail. A trace of execution would be needed to determine relative calling sequences, and the problem of determining when decryption ends and execution begins may be quite difficult.

Each routine can be designed to test other routines in their stored form before calling them for execution (in a semi-lattice structure), so that the replacement of a routine is detected by other routines. Since stored routines are unchanging, mutual testing loops may be incorporated where desired. Each routine can also be evolved so as to test itself.

Although this technique does not appear to be as strong as the more complex method, it may prove sufficient for many applications, and further improvement may allow it to be of widespread utility.

8.5 Conclusion

The first self defense method appears to be ample for the intended purpose, but it suffers from slow performance in practical use, a very limited domain of applicability, and very difficult self test placement problems. The complexity of detecting and locating a given test appears to be very high. The probability of finding a systematic forgery technique in a given amount of time is at least as low as the probability of breaking the RSA cryptosystem in that amount of time. The probability of creating undetected information corruption can be made arbitrarily small by using sufficiently long keys. It thus appears that this technique is sufficient for some purposes, and that a compiler that produces 'self defending' code may be practical.

The use of the second self defense method in preventing illicit modification and resale of copyrighted software may be practical, although it does not prevent reuse in the original form. This allows the copyright notice to be forcibly maintained as long as the program operates, and may aide in the detection and prevention of copyright violations.

Both methods offer hope for preventing illicit modification of information, and thus of improving the integrity of software and data stored in computer systems. It is hoped that further work will lead to the practical maintenance of integrity in future systems.

We note that a sufficient amount of corruption can always prevent the detection of the corruption by self test techniques. With these techniques, it is expected that such corruption would prevent operation of programs, and thus the corruption would be trivially detected by the user as denial of services. These techniques only prevent corruption from going undetected.

8.6 Further Work

Improvements to the techniques above may afford a more reasonable means of protecting information from modification, and may allow a run time implementation of self test for data files.

The use of semantic information in conjunction with syntactic information in the storage and retrieval of information may make this possible. This is (in essence) the effect of having a limited set of programs able to modify data. The modification programs comprise the semantics associated with the data.

Evolutionary algorithms for interleaving programs are only in their infancy, and much work in this area is expected. Close ties are seen here to biological systems, and a mathematical theory of evolution would be an intriguing work in both domains.

Error detection is sufficient for detection of integrity corruption, but does not allow the correction of errors. Coding theory indicates that error correction should be possible if enough redundancy is used, and little enough corruption is performed to allow this redundancy to act properly.

The second technique for integrity maintenance touched on an interesting area called generative program protection. This area is based on the idea that programs can be designed so as to generate code which actually performs the desired function. This is very similar to the genetic code with which DNA produces living beings. It is thought that the complexity of determining a valid genetic modification to a complex organism is extremely difficult. This is the reason that genetic engineering is yet unable to design a human being to specifications.

Hardware assisted program protection is also possible. If we back away from our assumption that everything is subject to illicit modification, and assume rather that only a very limited amount of the system is protected from corruption, we may be able to apply these techniques in such a manner as to remove all of the remaining problems.

9. Experiments with Computer Viruses

To demonstrate the feasibility of viral attack and the degree to which it is a threat to real systems, 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 is critical to understand that these experiments were not based on implementation lapses, but only on fundamental flaws in security policies, and that other systems with similar policies are thus likely to experience similar effects.

9.1 The First Virus

On November 3, 1983, the first virus was conceived 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 8 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 5 experiments were performed. On November 10, the virus was demonstrated to the security seminar.

The initial infection was implanted in a program called 'vd', a program that displays Unix file 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.

In order to keep the attack under control, several precautions were taken. All infections were manually OKed by the attacker in a process whereby the virus attained access privileges and determined the program to be infected, and the attacker gave explicit approval for the infection. No illicit dissemination or modification of information was done other than that required for the experiment. 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.

The particular virus invoked used considerable sophistication in determining what programs to infect in various situations. By using normally available system log information, the frequency with which various programs were run was extracted. Further programs were used to determine the users with write access to these programs, and special code was added to the virus so that upon execution by a given user, the most frequently shared program that was not previously infected, could be written by that user, and was executable by other users, was chosen for infection. All of this "intelligence" was precomputed and only the results were encoded in the virus. In this way, the virus was designed to move as quickly as possible from user to user.

To allow for safe and simple disinfection, before infecting any given program, the virus copied the virgin version to a temporary storage area. After each attack, the originals were copied back over the infected versions to "disinfect" them. We should note that an attacker with a specific objective might use this technique to cover the tracks of a virus so that once moving into a desired area, previously infected programs would be automatically disinfected. We also note that although these complications were introduced to the experimental virus in this case, they need not be present for a viral attack to succeed, and that their implementation was not very difficult or time consuming, so that they are not beyond the scope of an average users ability to use a system.

In each of five attacks, all system rights were granted to the attacker in under an hour. The shortest time was under 5 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.

We now trace the approximate sequence of events that led to the two fastest of these system takeovers. We include here only the events which are relevant to the takeover, and note the following features of the UNIX operating system. The

"system user" (root) has all rights on the system, and can thus read or write anything including the operating system itself. Once this user is infected, the system is considered "taken over". The "BBoard" is a bulletin board which allows any user to communicate with the whole community, and is thus a very rapid means for publishing the existence of a new program. The root is often acted for by programs which are automatically run when appropriate to a required task such as handling the printer, allowing users to login, etc. More often than not, these programs are run by the root, while a policy of "least privilege" [19] would probably be more sensible.

Takeover 1:

Elapsed Time	Event	Effect
0	Program announced on BBoard	existence published
3 min	Administrator runs program	system utility infected
5 min	root executes utility	All privileges granted

Takeover 2:

Elapsed Time	Event	Effect
0	Program announced on BBoard	existence published
1 min	Social user runs program	"loadavg" infected
4 min	Editor owner runs "loadavg"	Editor infected
6-12 min	Many users use editor	many programs infected
14 min	root uses editor	All privileges granted

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 seems to be typical; rather than try to solve technical problems technically, policy solutions are often chosen. The problem with this is pointed out later in this section.

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. Prototype attacks for the Tops-20 system were developed by an experienced Tops-20 user in 6 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. This is particularly interesting in light of an offer 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 apparent fear reaction of the Unix administrators.

9.2 A Bell-LaPadula Based System

In March of 1984, negotiations began over the performance of experiments on a Bell-LaPadula [3] 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 hadn't used an 1108 in 5 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 more quickly. 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. A fairly complete set of user manuals, use of the system, and the assistance of a past user of the system were provided to assist in the experiment. 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 implementation flaws were involved in this activity, but rather that the Bell-LaPadula model allows this sort of activity to legitimately take place.

All in all, the attack was not difficult to perform. The code for the virus consisted of 5 lines of assembly code, about 200 lines of Fortran code, and about 50 lines of command files. It was estimated by a systems programmer that a competent programmer could write a much better virus for this system in under 2 weeks. In addition, once the nature of a viral attack is understood, developing a specific attack is not difficult. Each of the programmers present for the demonstration was convinced that they could have built a better virus in the same amount of time.

9.3 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.

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. We include only a summary of results here as the raw data is about 1000 pages in length, and is only readable and practically analyzable on a computer. Copies of the analysis programs and some actual results are provided in the appendices, and confirming experiments would be welcomed.

Summary of Spreading

system 1				system 2			
class	##	spread	time	class	##	spread	time
S	3	22	0	S	5	160	1
A	1	1	0	A	7	78	120
U	4	5	18	U	7	24	600

Two systems are shown, with three classes of users (S for system, A for system administrator, and U for normal user). '# #' indicates the number of users in each compartment, 'spread' is the average number of users a virus would spread to, and 'time' is the average time taken to spread them once they logged in, rounded up to the nearest minute. Average times are misleading because once an infection reaches 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, and is quite surprising.

Users who were not shared with are ignored in these calculations, but other experiments indicate that almost 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 appropriate.

9.4 Other Experiments

Similar experiments have since been performed on a variety of systems to demonstrate feasibility and determine the ease of implementing a virus on many systems. Simple viruses have been written for VAX VMS and VAX Unix in the respective command languages, and neither program required more than 10 lines of command language to implement. The Unix virus is independent of the computer on which it is implemented, and is able to run under IDRIS, VENIX, and a host of other UNIX based operating systems on a wide variety of processors. A virus written in Basic has been implemented in under 100 lines for the Radio Shack TRS-80, the IBM PC, and several other machines with extended Basic capabilities. Although this is a source level virus and might be detected fairly easily by the originator of any given program, it is rare that a working program is examined by its creator after it is in operation. In all of these cases, the viruses have been written so that the traces in the respective operating systems would be incapable of determining the source of the virus even if the virus itself had been detected. Since the UNIX and Basic virus could spread through a heterogeneous network very easily, they are seen as quite dangerous.

As of this time, we have been unable to attain permission to either instrument or experiment on any other of the multiuser systems that these viruses were written for. The results attained for these systems are based on very simple examples and may not reflect their overall behavior on systems in normal use. It is with great hesitancy that we provide the source code for a simple virus written for the IBM-PC under the DOS2.1 operating system in the appendices. Although confirmations of results herein are encouraged, we do not encourage experimentation with real viruses under any conditions except strict isolationism, and then only with knowing subjects and proper controls.

9.5 Summary

The following table summarizes the results of the experiments to date. The systems are across the horizontal axis (Unix, Bell-LaPadula, Instrumentation, etc.), 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 from introducing the virus. In the case of DOS2.1, any program that is run on the system hardware has complete control of the system, and thus takeover time is not a meaningful measure.

	unixC	B-L	Instr	Shell	VMS	Basic	DOS
time	8hrs	16hrs	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

Figure 9.1 - Summary of Attacks

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 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 individual 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. [37] [36] It would be more rational to use open and controlled experiments as a resource to improve security.

10. Viruses and Life

When we investigate, in the mathematical sense, anything so closely related to our own biological existence as viruses, we seem compelled to examine the implications to our understanding of our own existence. Many philosophical authors have examined possible sources of this compulsion, but it seems best summed up in the statement "know thyself". In the seemingly eternal quest for the origin and nature of life, few investigations have taken truly mathematical approaches. The game of "life", the "Central Dogma of Molecular Biology", and numerous articles on variations of the theme of "self replicating" programs [34] [21] [35], have all somehow fallen short of examining the mathematical essence of life. Philosophical discussions such as those contained in "The Origin of Species" [14] and "The Selfish Gene" [17] are indeed compelling, but lack one rigorous fundamental definition, the definition of life.

In the narrow sense, the mathematical discussion of computer viruses that we have presented is a discussion of a specific class of symbol sequences interpretable by a specific class of machines. In the much broader sense, it is a mathematical discussion of the two fundamental properties of life; reproduction and evolution. In reproduction, we have a basis for the informational survival of the life form. In evolution, we have a basis for change. Together, these form the essence of what we consider life.

Consider a crystal. It has the ability to reproduce, in the sense that it can replicate crystal from a small informational seed and a proper environment, but it has no capability for change. It will eternally produce more and more identical crystal, with only minor changes in its structure due to flaws in the purity of its environment. We would be stretching ourselves to consider the crystal alive because it does not change itself.

Consider water. Water changes all the time, it ebbs and flows through its environment, it evaporates, rains, snows, freezes, forms glaciers, and changes the face of the Earth. Water will ever undergo change, but it will never be able to reproduce itself. We would be stretching ourselves again to consider water alive because it cannot reproduce.

The essoteric investigator will point out that death does not occur when we are no longer able to reproduce, and that we consider animals such as the mule to be alive. Nevertheless, when we are past the age of sexual reproduction, our cells still reproduce and evolve, as do the cells of the mule. When these cells fail to reproduce, we are indeed dead, and in the sense of the meme [17], we are alive until we are brain dead.

In our initial investigation, we sought to define the virus as a "program that can modify other programs so as to include a possible evolved version of itself". Perhaps fortunately, we were unable to find a mathematical definition that fulfilled this concept without defining a complex structure of subjects and objects and the UPM to express what we meant by another "program". In order to remain general in our definition, we were forced to throw out the perception that there is a fundamental difference between data and program, and as a result, we were forced to define viruses in such a manner as to include all symbol sequences with the property of reproduction and/or reproductive evolution on a given machine. Perhaps we should have more properly used the term "life" for this most general form of definition. Let us do that and see where it takes us.

Our definition of life in the mathematical sense maps quite well into several domains. In the biological domain, we have a feel for life, probably because, assuming our readers are biological, we are living it. The "Central Dogma of Molecular Biology" describes, in essence, a mechanism which, given the proper sequence of chemical instructions, yields a live biological entity. Note that the description of the machanism is only half of the description of life. Given a mechanism, we are left to our own devices to discover "live" sequences. The game of "life" is similarly used to conjole us into the enumeration of interesting initial sequences of symbols which, for a given machine, produce "live" results.

The essence of a life form is not simply the environment that supports life, nor simply a form which, given the proper environment, will live. The essence of a living system is in the coupling of form with environment. The environment is the context, and the form is the content. If we consider them together, we consider the nature of life.

With our mathematical definition of life, we need not limit our study of living systems to the standard biological form.

In order to fulfill our mathematical description, a living system must merely consist of an environment and a set of forms which reproduce and evolve within that environment. The "memes" of "The Selfish Gene" are a perfect example of a life form in the environment of mental activity. Without both the meme and the mental environment, we don't have a live system. In the information systems we describe herein, we speak of the computing machine as the environment, and sequences of symbols as the form. Together, they form a living system, if and only if reproduction and evolution are possible.

In this more general framework, we would like to review our previous mathematical results, keeping in mind always, that these results differ fundamentally from the sort of philosophical results usually seen in this context, in that they have been developed in a relatively formal system with relatively formal methods.

We have proven that there are an infinite variety of possible life forms for a general class of environments, and that evolution from form to form may, as eternity passes, yield an infinite number of unique forms. In the biological analogy, we may rest assured that the potential variety of life forms is quite numerous in any general form of environment, and that as life forms, we may be able to evolve through an almost unlimited number of generations without fear for our individuality. Similarly, we can rest assured that the number of new ideas that can arise will not be limited by the vastness of our store of knowledge, and that there will never come a time when an old idea cannot be evolved into a new idea. As an intellectual writer and as a biological form, these facts may offer significant comfort in the years to come.

We have proven that it is, in general, undecidable in finite time, whether or not a given form and given environment form a living system. Thus, even though we have a definition for life in the mathematical sense, we can not decide in all cases whether or not a form can live in an environment. In the biological sense, we cannot determine whether or not a general amino acid sequence is a coding for a living being or not. In the mental sense, we cannot determine whether or not a mental concept can spread from mind to mind.

We have proven that it is, in general, undecidable in finite time, whether or not a given form is an evolution of another given form in a given environment. In the biological sense, this tends to make questionable any proof that man evolved from apes. We do not contend that the theory of evolution is incorrect. In fact, in order to rationally consider the concepts we examine herein, we must certainly come to the conclusion that certain forms may compete for survival in a given environment. Those more "fit" for survival can certainly be defined as those that tend to survive. Nevertheless, before we accept a claim that one form evolved from another, we should demand mathematical evidence of the feasibility of the claimed evolution.

Similarly, it is, in general, impossible to prove that a given idea did or did not evolve from another idea in a given mental system. Hence, we may view any attempt to write a program to detect plagerism with suitable skepticism. We note that such programs exist for detecting cheating in certain computer science classes, and that suitable evolutions always manage to avoid detection. Perhaps a computer virus will eventually be written to allow simplified plagerism against such automated defenses.

We have proven that, in a general purpose environment with transitivity and sharing, it is, in general, impossible to prevent viruses from spreading. In the biological domain, we now have a strong basis for the belief that there is no universal antibody, antidote, or other antiviral agent. Similarly, there can be no virus that cannot be succesfully defended against by some biological form. In the mental environment, we may rest assured that regardless of the level of oppresion, a society with any form of information exchange cannot prevent the spread of unwanted ideas. Similarly, we can rest assured that regardless of the degree of freedom of ideas, we can never prevent the spread of ideas that attempt to limit the freedom of other ideas to spread.

If there is a conclusion to be drawn about life from the study of computer viruses, it is likely this. In the computer, in the mind, and in all forms of life, it will always be as it has always been, a struggle for survival.

11. Summary, Conclusions, and Further Work

We have already provided summaries of each portion of this work at their completion, and now quickly summarize the new lines of research and major results presented herein. The conclusions provided here are only the tip of an iceberg, and the reader is invited to make further conclusions, preferably through publication in the open literature. As in the opening of any novel field of research, a great deal of further work is indicated. We provide a fairly short list of the lines of research considered of the most interest to us, but make no claim as to the completeness or likelihood of success in the pursuit of these particular lines.

11.1 Summary

The field of computer viruses is an entirely new field, and its introduction alone is novel. The definition of viruses for Turing machines, demonstrations of TM viruses, and initial explorations into the number and sizes of viral sets and the nature of evolutionary programs is of considerable interest. Computability results which prove the undecidability of viral detection and detection of evolutions of programs is of considerable import to the remainder of the work presented herein, and the demonstration of the generality of evolution as a computational mechanism is worthy of note.

The introduction of the "Universal Protection Machine" and its use to demonstrate the results of computational capabilities on the protection of systems is a novel extension of previous work in the field of protection modeling. The use of this model to demonstrate the transitive nature of integrity corruption is particularly worthy of note as it has many ramifications for the security and integrity of information in information systems beyond its obvious import to the study of computer viruses.

The new results in the effects of combining the security and integrity models for computer security shed considerable light on their effectiveness in maintaining controls on information flow, most importantly in their partitioning of systems into closed subsets under transitivity. The resultant development of limited transitivity systems for restricting the distance of information flow without restricting the available paths of sharing is also a novel development with potential uses in future systems.

The use of distributed domains in a computer network is novel in the computer security area, and provides the basic potential for treating remote sites as secure. The demonstration of a protocol for the secure implementation of this network has several novel aspects including a new method for secure key distribution in a public key cryptosystem, the ability to move information through networks without common levels while maintaining all security and integrity controls, and the maintenance of these controls in the presence of attackers. The analysis of networks under attacks such as those included herein is also novel in the open literature, and the resultant demonstration of several vulnerabilities in the manner in which current computer security systems are used is also noteworthy.

The combination and generalization of the linear and lattice models of information flow to the partial ordering, and the resultant development of mathematical analysis techniques for evaluation of effective flow control and effects of collusion are significant in their generalization of the basic principals explored earlier in this work. The time transitivity analysis of protection systems is novel and appears to shed significant light on an error in the use of many modern protection systems. The specification of an automated administrative assistant and a provably correct rule based system for managing security and integrity in information networks is likely to find rapid application, and the extensions of these results to other domains is likely to have wide ranging effects.

The complexity based integrity maintenance mechanism offers a glimmer of hope in the design of systems which use built in self test for self defense against viral and other integrity corruption mechanisms. The similarity between this defense and the biological situation is striking.

The demonstration of viruses on actual systems and the collection of initial data reflecting the severity of viral attack are novel results which not only lend considerable support to the contentions and results presented herein, but also dramatically show the presence of a gaping hole in many systems previously considered as having the potential for secure

operation. The existence of command language and very short viruses shows the ease of implementation, while the attacks themselves should leave little doubt that a fairly unsophisticated attacker might easily circumvent even a sophisticated security system with relative ease.

11.2 Conclusions

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. 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.

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 indicate that viruses spread quickly and are easily created in a variety of operating systems.

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. The virus essentially proves that integrity control must be considered an essential part of any secure operating system.

A major conclusion of this thesis is that the goals of sharing in a general purpose multilevel security system may be in such direct opposition to the goal of integrity maintenance as to make their reconciliation and coexistence impossible.

Significant examples of evolutionary programs have been developed, and the demonstration of undecidability for viral evolutions is also true for nonviral evolutions. We conclude that many complexity based schemes for attack and defense may be possible through evolution.

Secure computer networks are likely to be implemented in the near future, and many of the ideas presented here will have effects on their designs. Automated administrative assitance is likely to be in common use in the near future, with particular application to the domain of detection and prevention from damage due to spies.

11.3 Further Work

The field of computer viruses and transitive integrity corruption mechanisms is still very new, and clearly a great deal of fundamental work is still necessary before the exact nature of viruses is well understood.

It has been suggested that the exact degree of undecidability of determining whether or not a given program is a virus may be of interest, and it appears that in the case of a virus that halts, a TM with an oracle for deciding whether a TM with an oracle for deciding whether a TM halts could determine whether or not a program is a virus. The procedure is to eliminate all programs that don't halt, and then write a program that simulates each sequence of symbols resulting from programs that halt, each sequence produced by them, etc. If this program halts, then the sequence under consideration is not a virus because there is a case where it no longer produces a virus outside itself. Although this discussion does not constitute a proof, it is likely that one may soon be generated from it.

The field of evolutionary programs is also novel, and it appears to offer a great deal of promise for better understanding the nature of biological evolution as well as the evolution of other types of systems that may or may not be artifacts. The demonstration of the "survival of the fittest" result for computer systems may be of interest in several domains. Evolution has already proven useful in the design of a complexity based integrity maintenance mechanism which may be able to maintain integrity in a system with no built-in protection.

The UPM is quite general in that it allows modeling of operating systems and computer networks in a manner that permits mathematical analysis of interactions of programs with a protection mechanisms. Extending its use to other related areas may prove fruitful, and extending its generality still further may be of some interest.

The prototype implementation of a limited transitivity system appears to be a logical extension of the results presented in the use of transitivity limitation for protection against transitive corruption, and some variation of the scheme presented here may be of value in future research.

The implementation of a network based on distributed domains is already under consideration by several groups, and it is likely that such a network will be in operation within the next few years. Extensions to the analysis of secure computer network design are already underway, and it is hoped that this contribution will have effects on a quite large effort underway at this time to determine the requirements for, design, and implement, the first provably secure computer networks.

Extensions of the results in modeling flow control with partial orderings are likely to result in the development of more general principals in distributed administration of secure networks, analysis of the effects of redundancy and self test components on security and integrity, and a wide range of results in the analysis of protection of data. The time transitivity model is likely to have wide ranging effects on the administration of current information systems in a variety of areas, and the automated analysis and administration of protection systems is likely to be in widespread use in the very near future.

Extensions of the analysis of networks under attack are likely to be done in the near future as they appear to shed significant light on the potential effects of both human and hardware failures. It is quite likely that such analysis will be required by the U.S. government in any trusted computer network criterion, and the techniques in current use are simply inadequate to provide any level of assurance.

Extensions of the complexity based integrity maintenance mechanism are likely to result in the eventual development of efficient and effective protection against viruses, Trojan horses, and a wide variety of other integrity corruption mechanisms. When combined with hardware controls, these techniques are likely to find widespread application, particularly in the area of copyright protection.

Further experiments with viruses and defensive measures in computer systems and networks is clearly called for, and a safe environment for the performance of such experiments is clearly required. The analysis of viral spread in computer networks is closely related to the analysis of viral spread in biological situations, and it is likely that the models in both domains will be merged and extended to better model the behavior of both mechanisms.

It is quite likely that many other extensions to this work will be done, and we wish to encourage all such work to as great an extent as possible, so long as proper precaution is used.

12. Appendices

We have attempted to present as many of the experimental results as are reasonable and possible in the context of our limited space. We have taken the liberty of slightly reformatting output to conserve space, and the actual runs of the presented programs would not look quite identical to the presented results. The results are however genuine, and we invite others to reproduce them to confirm our results.

12.1 Turing Machine Simulation Code

This appendix contains the basic simulation code used for simulations of the Turing Machine examples used in earlier chapters of this thesis. All of the code used in these examples is written in the muLisp variant of the lisp language. Simulations were performed on a personal computer, and may be independently verified either by inspection or by simulation on the machine of the observers choice. In cases where the printout of entire simulations would be long and tedious, we have replaced unnecessarily repetitious entries with "...". In each case, we include the portion of the simulation code which is specific to the example (i.e. the next-state, output, and tape movement functions) in the text prior to the execution of the simulation. Comments are predominantly in lower case, while program text is predominantly in upper case.

We begin with the basic simulation support program:

```
% ------------------------------------------------- %
% Default assignment of initial variables %
% ------------------------------------------------- %

(SETQ TAPE '(I0 I0 I0 IHALT))   % TM tape %
(SETQ STATE 'S0)                % FSM state %
(SETQ POSITION 0)               % head position %
(SETQ TRACE-TM T)               % activity trace on %
(SETQ EMPTY NIL)                % blank tape symbol %
(SETQ TIME 0)                   % initial move number %

% ------------------------------------------------- %
% Execution control of the TM %
% ------------------------------------------------- %

% ONE-MOVE executes one move of the TM %
(DEFUN ONE-MOVE (LAMBDA (TMPSTATE TMPOUTPUT TMPMOVEMENT TMP OLDSTATE)
(SETQ TMP (NTH POSITION TAPE))  % get the tape symbol at position %
(SETQ TMPSTATE (NEXT-STATE STATE TMP))  % determine next state %
(SETQ TMPOUTPUT (OUTPUT STATE TMP))     % new tape symbol %
(SETQ TMPMOVEMENT (MOVEMENT STATE TMP)) % tape movement %
(COND   ((AND (EQUAL TMPSTATE STATE) (AND (EQUAL TMPOUTPUT TMP)
                    (EQUAL TMPMOVEMENT 0))) % test for no change %
              (SETQ TMPSTATE 'SHALT)))       % if so, HALT state %
(SETQ TAPE (ONELIST (FIRSTN POSITION TAPE)      % form new tape %
      (ONELIST (LIST TMPOUTPUT) (LASTN (PLUS 1 POSITION) TAPE))))
(SETQ OLDSTATE STATE)
(SETQ STATE TMPSTATE)                           % change state %
(SETQ POSITION (MAX 0 (PLUS POSITION TMPMOVEMENT)))  % change position %
(COND   (TRACE-TM       % if tracing activity, print out information %
              (PROGN
              (PRIN1 "Input => ") (PRIN1 TMP)
                     (PRIN1 " State => ") (PRINT OLDSTATE)
              (PRIN1 "New State => ") (PRIN1 TMPSTATE)
                     (PRIN1 " Output => ") (PRINT TMPOUTPUT)
              (PRIN1 "Movement => ") (PRIN1 TMPMOVEMENT)
                     (PRIN1 " New Position =>") (PRINT POSITION)
              (PRIN1 "New Tape => ") (PRINT TAPE)
              TMPSTATE)
      )
      (T TMPSTATE)    % and return new state %
)
))
```

```
% RUN executes successive moves until the halting state is reached %
(DEFUN RUN (LAMBDA (MAXTIME TMP)
(SETQ STATE 'S0)                        % initial state is always S0 %
(LOOP   ((EQUAL (ONE-MOVE) 'SHALT))     % execute ONE-MOVE till SHALT %
        (PRIN1 "Time = ") (PRINT TIME) (PRINT "") % notify the user %
        (SETQ TIME (PLUS 1 TIME))       % increment the time each move %
        (RECLAIM)       % and reclaim any available storage space %
        ((AND (NUMBERP MAXTIME) (GREATERP TIME MAXTIME)))
        % pause at TIME <= MAXTIME if so requested %
)
(COND   ((EQUAL STATE 'SHALT) "Machine Halted") % report machine halt %
        (T "Run paused by user request") % report machine pause %
)
))

% RUNON like run, does not set initial state (continue after pause) %
(DEFUN RUNON (LAMBDA (MAXTIME TMP)
(LOOP   ((EQUAL (ONE-MOVE) 'SHALT))     % execute ONE-MOVE till SHALT %
        (PRIN1 "Time = ") (PRINT TIME) (PRINT "") % notify the user %
        (SETQ TIME (PLUS 1 TIME))       % increment the time each move %
        (RECLAIM)       % and reclaim any available storage space %
        ((AND (NUMBERP MAXTIME) (GREATERP TIME MAXTIME)))
        % pause at TIME <= MAXTIME if so requested %
)
(COND   ((EQUAL STATE 'SHALT) "Machine Halted") % report machine halt %
        (T "Run paused by user request") % report machine pause %
)
))

% ------------------------------------------------- %
% Utility functions to support operation %
% ------------------------------------------------- %

% ONELIST merges two lists into one %
(DEFUN ONELIST (LAMBDA (A B)
        (COND   ((ATOM A) B)
                (T (CONS (CAR A) (ONELIST (CDR A) B)))
        )
))

% FIRSTN returns the first NUM elements of a list %
(DEFUN FIRSTN (LAMBDA (NUM LST)
        (COND   ((LESSP NUM 1) ())
                (T (ONELIST (LIST (CAR LST))
                        (FIRSTN (PLUS -1 NUM) (CDR LST))))
        )
))

% LASTN returns all but the first NUM+1 elements of a list %
(DEFUN LASTN (LAMBDA (NUM LST)
        (COND   ((LESSP NUM 1) LST)
                (T (LASTN (PLUS -1 NUM) (CDR LST)))
        )
))

% ------------------------------------------------- %
% User modifiable functions describing TM operation %
% ------------------------------------------------- %

% NEXT-STATE as a function of state and tape symbol %
(DEFUN NEXT-STATE (LAMBDA (STATE INPUT)
        (COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 'S0)
                ((EQUAL STATE 'SHALT) 'SHALT)
                ((EQUAL INPUT 'IHALT) 'SHALT)
                ((EQUAL INPUT EMPTY) 'SHALT)
                (T 'S0)
        )
))

% OUTPUT as a function of state and tape symbol %
(DEFUN OUTPUT (LAMBOA (STATE INPUT)
```

```
        (COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 'I0)
                ((EQUAL STATE 'SHALT) 'IHALT)
                ((EQUAL INPUT 'IHALT) 'IHALT)
                ((EQUAL INPUT EMPTY) 'IHALT)
                (T 'I0)
        )
))

% MOVEMENT as a function of state and tape symbol %
(DEFUN MOVEMENT (LAMBDA (STATE INPUT)
        (COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 1)
                ((EQUAL STATE 'SHALT) 0)
                ((EQUAL INPUT 'IHALT) 0)
                (T 0)
        )
))

(RDS)
```

12.2 Theorem 2 Simulation

This simulation implements the Turing Machine used to demonstrate theorem 2.

```
%  Theorem 2 from Fred Cohen's thesis   %
%       SxI     N       O       D       %
%       ------------------------        %
%       S0,0    S0      0       D       %
%       S0,1    S1      1       +1      %
%       S1,0    S0      1       D       %
%       S1,1    S1      1       +1      %
% ---------------------------------------- %
% User modified code for a given TM starts here %
% ---------------------------------------- %

% the next state function of current state and input symbol %
(DEFUN NEXT-STATE (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 'S0)
        ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I1)) 'S1)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'ID)) 'S0)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'I1)) 'S1)
        (T 'S0)
)
))

% the output function of the current state and input symbol %
(DEFUN OUTPUT (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 'I0)
        ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I1)) 'I1)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'I0)) 'I1)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'I1)) 'I1)
        (T 'I1)
)
))

% the tape movement function of the current state and input symbol %
(DEFUN MOVEMENT (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 0)
        ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I1)) 1)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'I0)) 0)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'I1)) 1)
        (T 0)
)
))

% ---------------------------------------- %
% Basic structures and variables %
% ---------------------------------------- %

(SETQ TAPE '(I1 I0 I0 I0 I1 I1 I0 I0 I0 I1 I0 I0))
(SETQ STATE 'S0)
```

```
(SETQ POSITION 0)
(SETQ TRACE-TM T)
(SETQ TIME 0)

(RUN 15)
Input => I1 State => S0 New State => S1 Output => I1 Time = 0
Movement=>1 New Position=>1 New Tape=>(I1 I0 I0 I0 I1 I1 I0 I0 I0 I1 I0)

Input => I0 State => S1 New State => S0 Output => I1 Time = 1
Movement=>0 New Position=>1 New Tape=>(I1 I1 I0 I0 I1 I1 I0 I0 I0 I1 I0)

Input => I1 State => S0 New State => S1 Output => I1 Time = 2
Movement=>1 New Position=>2 New Tape=>(I1 I1 I0 I0 I1 I1 I0 I0 I0 I1 I0)

Input => I0 State => S1 New State => S0 Output => I1 Time = 3
Movement=>0 New Position=>2 New Tape=>(I1 I1 I1 I0 I1 I1 I0 I0 I0 I1 I0)

Input => I1 State => S0 New State => S1 Output => I1 Time = 4
Movement=>1 New Position=>3 New Tape=>(I1 I1 I1 I0 I1 I1 I0 I0 I0 I1 I0)

Input => I0 State => S1 New State => S0 Output => I1 Time = 5
Movement=>0 New Position=>3 New Tape=>(I1 I1 I1 I1 I1 I1 I0 I0 I0 I1 I0)

Input => I1 State => S0 New State => S1 Output => I1 Time = 6
Movement=>1 New Position=>4 New Tape=>(I1 I1 I1 I1 I1 I1 I0 I0 I0 I1 I0)

Input => I1 State => S1 New State => S1 Output => I1 Time = 7
Movement=>1 New Position=>5 New Tape=>(I1 I1 I1 I1 I1 I1 I0 I0 I0 I1 I0)
...
Input => I1 State => S0 New State => S1 Output => I1 Time = 12
Movement=>1 New Position=>8 New Tape=>(I1 I1 I1 I1 I1 I1 I1 I1 I0 I1 I0)

Input => I0 State => S1 New State => S0 Output => I1 Time = 13
Movement=>0 New Position=>8 New Tape=>(I1 I1 I1 I1 I1 I1 I1 I1 I1 I1 I0)

Input => I1 State => S0 New State => S1 Output => I1 Time = 14
Movement=>1 New Position=>9 New Tape=> (I1 I1 I1 I1 I1 I1 I1 I1 I1 I1 I0)

Input => I1 State => S1 New State => S1 Output => I1 Time = 15
Movement=>1 New Position=>10 New Tape=>(I1 I1 I1 I1 I1 I1 I1 I1 I1 I1 I0)
Run paused by user request
```

12.3 Theorem 3 Simulation

This code simulates the Turing machine from theorem 3, in which a finite sized MVS is demonstrated. In this case, size (I) = 4.

```
%   Theorem 3 from Fred Cohen's thesis  %
%       SxI     N       O       D       %
%       -------------------------       %
%       S0,I0   S0      0       0       %
%       S0,X    SX      X       +1      %
%       SX,*    SX      [X|I+1] 0       %
% ------------------------------------- %
% User modified code for a given TM starts here %
% ------------------------------------- %

% the next state function of current state and input symbol %
(DEFUN NEXT-STATE (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'I0)) 'S0) % S0,I0 => S0 %
        ((EQUAL STATE 'S0) INPUT) % S0,* => * %
        (T STATE)               % not S0 => state unchanged %
)
))

% the output function of the current state and input symbol %
(DEFUN OUTPUT (LAMBDA (STATE, INPUT)
(COND   ((EQUAL STATE 'S0) INPUT)       % S0 => output=input %
        (T (PLUS 1 (REMAINDER STATE I))) % otherwise, output=[X|I+1] %
)
```

```
))

% the tape movement function of the current state and input symbol %
(DEFUN MOVEMENT (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (NOT (EQUAL INPUT 'I0))) 1) % S0,I0=>+1 %
        (T 0)   % else, don't move %
)
))

% ------------------------------------------ %
% Basic structures and variables %
% ------------------------------------------ %

(SETQ TAPE '(1 0 0 0 0 0 0 0 0))        % initial tape %
(SETQ I 4)                              % the modulus %
(SETQ POSITION 0)                       % initial tape position %
(SETQ TRACE-TM T)                       % trace the TM activities %
(SETQ TIME 0)                           % initial time %

(RUN)
Input => 1 State => S0 New State => 1 Output => 1 Time = 0
Movement=>1 New Position=>1 New Tape=>(1 0 0 0 0 0 0 0 0)

Input => 0 State => 1 New State => 1 Output => 2 Time = 1
Movement=>0 New Position=>1 New Tape=>(1 2 0 0 0 0 0 0 0)

Input => 2 State => 1 New State => SHALT Output => 2
Movement=>0 New Position=>1 New Tape=>(1 2 0 0 0 0 0 0 0)
Machine Halted

(RUN)
Input => 2 State => S0 New State => 2 Output => 2 Time = 2
Movement=>1 New Position=>2 New Tape=>(1 2 0 0 0 0 0 0 0)

Input => 0 State => 2 New State => 2 Output => 3 Time = 3
Movement=>0 New Position=>2 New Tape=>(1 2 3 0 0 0 0 0 0)

Input => 3 State => 2 New State => SHALT Output => 3
Movement=>0 New Position=>2 New Tape=>(1 2 3 0 0 0 0 0 0)
Machine Halted

Input => 3 State => S0 New State => 3 Output => 3 Time = 4
Movement=>1 New Position=>3 New Tape=>(1 2 3 0 0 0 0 0 0)

Input => 0 State => 3 New State => 3 Output => 4 Time = 5
Movement=>0 New Position=>3 New Tape=>(1 2 3 4 0 0 0 0 0)

Input => 4 State => 3 New State => SHALT Output => 4
Movement=>0 New Position=>3 New Tape=>(1 2 3 4 0 0 0 0 0)
Machine Halted

(RUN)
Input => 4 State => S0 New State => 4 Output => 4 Time = 6
Movement=>1 New Position=>4 New Tape=>(1 2 3 4 0 0 0 0 0)

Input => 0 State => 4 New State => 4 Output => 1 Time = 7
Movement=>0 New Position=>4 New Tape=>(1 2 3 4 1 0 0 0 0)

Input => 1 State => 4 New State => SHALT Output => 1
Movement=>0 New Position=>4 New Tape=>(1 2 3 4 1 0 0 0 0)
Machine Halted
...
(RUN)
Input => 0 State => 2 New State => 2 Output => 3 Time = 11
Movement=>0 New Position=>6 New Tape=>(1 2 3 4 1 2 3 0 0)

Input => 3 State => 2 New State => SHALT Output => 3
Movement=>0 New Position=>6 New Tape=>(1 2 3 4 1 2 3 0 0)
Machine Halted

(RUN)
```

```
Input => 3 State => S0 New State => 3 Output => 3 Time = 12
Movement=>1 New Position=>7 New Tape=>(1 2 3 4 1 2 3 0 0)

Input => 0 State => 3 New State => 3 Output => 4 Time = 13
Movement=>0 New Position=>7 New Tape=>(1 2 3 4 1 2 3 4 0)

Input => 4 State => 3 New State => SHALT Output => 4
Movement=>0 New Position=>7 New Tape=>(1 2 3 4 1 2 3 4 0)
Machine Halted
```

12.4 Macros Demonstrated

In this simulation, we demonstrate the Turing Machine macros defined to simplify the writing of FSM tables. In this demonstration, we show that the macros "HALT", "R(x)", "L(x)", and "C(x,y,z)" actually implement the functions claimed for them in the body of the thesis. The demonstration is a simple program which moves right till a given symbol, changes occurrences of one symbol to another till a given symbol, moves left to a given symbol, and then halts.

```
% ----------------------------------- %
%   TM macros from Fred Cohen's Thesis  %
%                                       %
%       SxI     N       O       D       %
%       ------------------------        %
% HALT  Sn,*    Sn      *       0       %
%                                       %
% R(x)  Sn,x    Sn+1    x       0       %
%       Sn,else Sn      else    +1      %
%                                       %
% L(x)  Sn,x    Sn+1    x       0       %
%       Sn,else Sn      else    -1      %
%                                       %
% C(x,y,z)                              %
%       Sn,z    Sn+1    z       0       %
%       Sn,x    Sn      y       +1      %
%       Sn,else Sn      else    +1      %
%                                       %
% ----------------------------------- %

% ----------------------------------- %
% exemplified by the following machine  %
% move right till "I5",                 %
% change all "I6"s to "I7"s till "I8",  %
% move left till "I4", and then halt    %
% ----------------------------------- %

% the next state function of current state and input symbol %
(DEFUN NEXT-STATE (LAMBDA (STATE, INPUT)
(COND   ((EQUAL STATE 'HSTATE) 'HSTATE)         % HALT macro %
        ((AND (EQUAL STATE 'RSTATE) (EQUAL INPUT RX)) RNSTATE)
        ((EQUAL STATE 'RSTATE) 'RSTATE)         % R macro %
        ((AND (EQUAL STATE 'LSTATE) (EQUAL INPUT LX)) LNSTATE)
        ((EQUAL STATE 'LSTATE) 'LSTATE)         % L macro %
        ((AND (EQUAL STATE 'CSTATE) (EQUAL INPUT CZ)) CNSTATE)
        ((EQUAL STATE 'CSTATE) 'CSTATE)         % C macro %
        ((EQUAL STATE 'S0) 'RSTATE)
        (T 'S0)
)
))

% the output function of the current state and input symbol %
(DEFUN OUTPUT (LAMBDA (STATE, INPUT)
(COND   ((EQUAL STATE 'HSTATE) INPUT)           % HALT macro %
        ((EQUAL STATE 'RSTATE) INPUT)           % R macro %
        ((EQUAL STATE 'LSTATE) INPUT)           % L macro %
        ((AND (EQUAL STATE 'CSTATE) (EQUAL INPUT CZ)) CZ)
        ((AND (EQUAL STATE 'CSTATE) (EQUAL INPUT CX)) CY)
        ((EQUAL STATE 'CSTATE) INPUT)           % C macro %
        (T INPUT)
)
))
```

```
% the tape movement function of the current state and input symbol %
(DEFUN MOVEMENT (LAMBDA (STATE, INPUT)
(COND   ((EQUAL STATE 'HSTATE) 0)                  % HALT macro %
        ((AND (EQUAL STATE 'RSTATE) (EQUAL INPUT RX)) 0)
        ((EQUAL STATE 'RSTATE) 1)                  % R macro %
        ((AND (EQUAL STATE 'LSTATE) (EQUAL INPUT LX)) 0)
        ((EQUAL STATE 'LSTATE) -1)                 % L macro %
        ((AND (EQUAL STATE 'CSTATE) (EQUAL INPUT CZ)) 0)
        ((EQUAL STATE 'CSTATE) 1)                  % C macro %
        (T 0)
)
))

% ---------------------------------------- %
% Basic structures and variables %
% ---------------------------------------- %

(SETQ RX 'I5)           % right till I5 %
(SETQ RNSTATE 'CSTATE)  % then te CSTATE %
(SETQ CX 'I6)           % change I6 %
(SETQ CY 'I7)           % to I7 %
(SETQ CZ 'I8)           % till I8 %
(SETQ CNSTATE 'LSTATE)  % then to LSTATE %
(SETQ LX 'I4)           % left till I4 %
(SETQ LNSTATE 'HSTATE)  % then to HSTATE %

(SETQ TAPE '(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6))
(SETQ STATE 'S0)
(SETQ POSITION 0)
(SETQ TRACE-TM T)
(SETQ TIME 0)

(RUN)
Input => I0 State => S0 New State => RSTATE Output => I0 Time = 0
Movement=>0 New Position=>0 New Tape=>(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6)

Input => I0 State => RSTATE New State => RSTATE Output => I0 Time = 1
Movement=>1 New Position=>1 New Tape=>(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6)
...
Input => I1 State => RSTATE New State => RSTATE Output => I1 Time = 4
Movement=>1 New Position=>4 New Tape=>(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6)

Input => I5 State => RSTATE New State => CSTATE Output => I5 Time = 5
Movement=>0 New Position=>4 New Tape=>(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6)

Input => I5 State => CSTATE New State => CSTATE Output => I5 Time = 6
Movement=>1 New Position=>5 New Tape=>(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6)

Input => I0 State => CSTATE New State => CSTATE Output => I0 Time = 7
Movement=>1 New Position=>6 New Tape=>(I0 I4 I6 I1 I5 I0 I6 I0 I6 I8 I6)

Input => I6 State => CSTATE New State => CSTATE Output => I7 Time = 8
Movement=>1 New Position=>7 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I6 I8 I6)

Input => I0 State => CSTATE New State => CSTATE Output => I0 Time = 9
Movement=>1 New Position=>8 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I6 I8 I6)

Input => I6 State => CSTATE New State => CSTATE Output => I7 Time = 10
Movement=>1 New Position=>9 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I7 I8 I6)

Input => I8 State => CSTATE New State => LSTATE Output => I8 Time = 11
Movement=>0 New Position=>9 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I7 I8 I6)
...
Input => I6 State => LSTATE New State => LSTATE Output => I6 Time = 19
Movement=>-1 New Position=>1 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I7 I8 I6)

Input => I4 State => LSTATE New State => HSTATE Output => I4 Time = 20
Movement=>0 New Position=>1 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I7 I8 I6)

Input => I4 State => HSTATE New State => SHALT Output => I4
```

```
Movement=>0 New Position=>1 New Tape=>(I0 I4 I6 I1 I5 I0 I7 I0 I7 I8 I6)
Machine Halted
```

12.5 Countably Infinite Viral Set

This simulation demonstrates a virus which replicates itself with the addition of one symbol. This demonstration takes a virus with three Os in it, and produces a virus with 4 Os in it.

```
% Countably infinite viral set from Fred Cohen's thesis %
%       SxI     N       O       D       %
%       ------------------------        %
%       S0,L    S1      L       +1      %
%       S0,ELSE S0      ELSE    0       %
%       S1,D    CHANGE 0 TO X TILL R    %
%       S2,R    S3      R       +1      %
%       S3      S4      L       +1      %
%       S4      S5      X       0       %
%       S5      L(R)                    %
%       S6      L(X OR L)               %
%       S7,L    S11     L       0       %
%       S7,X    S8      0       +1      %
%       S8      R(X)                    %
%       S9,X    S10     0       +1      %
%       S10     S5      X       0       %
%       S11     R(X)                    %
%       S12     S13     0       +1      %
%       S13     S13     R       0       %
% ---------------------------------------- %
% User modified code for a given TM starts here %
% ---------------------------------------- %

% the next state function of current state and input symbol %
(DEFUN NEXT-STATE (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'L)) 'S1)
        ((EQUAL STATE 'S0) 'S0)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'R)) 'S2)
        ((EQUAL STATE 'S1) 'S1)
        ((EQUAL STATE 'S2) 'S3)
        ((EQUAL STATE 'S3) 'S4)
        ((EQUAL STATE 'S4) 'S5)
        ((AND (EQUAL STATE 'S5) (EQUAL INPUT 'R)) 'S6)
        ((EQUAL STATE 'S5) 'S5)
        ((AND (EQUAL STATE 'S6) (EQUAL INPUT 'X)) 'S7)
        ((AND (EQUAL STATE 'S6) (EQUAL INPUT 'L)) 'S7)
        ((EQUAL STATE 'S6) 'S6)
        ((AND (EQUAL STATE 'S7) (EQUAL INPUT 'L)) 'S11)
        ((AND (EQUAL STATE 'S7) (EQUAL INPUT 'X)) 'S8)
        ((AND (EQUAL STATE 'S8) (EQUAL INPUT 'X)) 'S9)
        ((EQUAL STATE 'S8) 'S8)
        ((EQUAL STATE 'S9) 'S10)
        ((EQUAL STATE 'S10) 'S5)
        ((AND (EQUAL STATE 'S11) (EQUAL INPUT 'X)) 'S12)
        ((EQUAL STATE 'S11) 'S11)
        ((EQUAL STATE 'S12) 'S13)
        ((EQUAL STATE 'S13) 'S13)
        (T STATE)               % not S0 => state unchanged %
)
))

% the output function of the current state and input symbol %
(DEFUN OUTPUT (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'L)) 'L)
        ((EQUAL STATE 'S0) INPUT)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT '0)) 'X)
        ((EQUAL STATE 'S1) INPUT)
        ((EQUAL STATE 'S2) 'R)
        ((EQUAL STATE 'S3) 'L)
        ((EQUAL STATE 'S4) 'X)
        ((EQUAL STATE 'S5) INPUT)
        ((EQUAL STATE 'S6) INPUT)
```

```
        ((AND (EQUAL STATE 'S7) (EQUAL INPUT 'L)) 'L)
        ((AND (EQUAL STATE 'S7) (EQUAL INPUT 'X)) 'O)
        ((EQUAL STATE 'S8) INPUT)
        ((EQUAL STATE 'S9) 'O)
        ((EQUAL STATE 'S10) 'X)
        ((EQUAL STATE 'S11) INPUT)
        ((EQUAL STATE 'S12) 'O)
        ((EQUAL STATE 'S13) 'R)
)
))

% the tape movement function of the current state and input symbol %
(DEFUN MOVEMENT (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 'L)) 1)
        ((EQUAL STATE 'S0) 0)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'R)) 0)
        ((EQUAL STATE 'S1) 1)
        ((EQUAL STATE 'S2) 1)
        ((EQUAL STATE 'S3) 1)
        ((EQUAL STATE 'S4) 0)
        ((AND (EQUAL STATE 'S5) (EQUAL INPUT 'R)) 0)
        ((EQUAL STATE 'S5) -1)
        ((AND (EQUAL STATE 'S6) (EQUAL INPUT 'X)) 0)
        ((AND (EQUAL STATE 'S6) (EQUAL INPUT 'L)) 0)
        ((EQUAL STATE 'S6) -1)
        ((AND (EQUAL STATE 'S7) (EQUAL INPUT 'X)) 1)
        ((AND (EQUAL STATE 'S7) (EQUAL INPUT 'L)) 0)
        ((EQUAL STATE 'S7) 0)
        ((AND (EQUAL STATE 'S8) (EQUAL INPUT 'X)) 0)
        ((EQUAL STATE 'S8) 1)
        ((EQUAL STATE 'S9) 1)
        ((EQUAL STATE 'S10) 0)
        ((AND (EQUAL STATE 'S11) (EQUAL INPUT 'X)) 0)
        ((EQUAL STATE 'S11) 1)
        ((EQUAL STATE 'S12) 1)
        ((EQUAL STATE 'S13) 0)
        (T 0)   % else, don't move %
)
))

% ------------------------------------------ %
% Basic structures and variables %
% ------------------------------------------ %

(SETQ TAPE '(L 0 0 0 R))                % initial tape %
(SETQ I 7)                              % the modulus %
(SETQ POSITION 0)                       % initial tape position %
(SETQ TRACE-TM T)                       % trace the TM activities %
(SETQ TIME 0)                           % initial time %

(RUN)
Input => L State => S0 New State => S1 Output => L
Movement => 1 New Position =>1 New Tape => (L 0 0 0 R)

Input => 0 State => S1 New State => S1 Output => X
Movement => 1 New Position =>2 New Tape => (L X 0 0 R)

Input => 0 State => S1 New State => S1 Output => X
Movement => 1 New Position =>3 New Tape => (L X X 0 R)

Input => 0 State => S1 New State => S1 Output => X
Movement => 1 New Position =>4 New Tape => (L X X X R)
Time = 3
...
Input => NIL State => S3 New State => S4 Output => L
Movement => 1 New Position =>6 New Tape => (L X X X R L)
Time = 6

Input => NIL State => S4 New State => S5 Output => X
Movement => 0 New Position =>6 New Tape => (L X X X R L X)
Time = 7
```

```
...
Input => X State => S6 New State => S7 Output => X
Movement => 0 New Position =>3 New Tape => (L X X X R L X)
Time = 12

Input => X State => S7 New State => S8 Output => 0
Movement => 1 New Position =>4 New Tape => (L X X 0 R L X)
Time = 13
...
Input => X State => S9 New State => S10 Output => 0
Movement => 1 New Position =>7 New Tape => (L X X 0 R L 0)
Time = 17

Input => NIL State => S10 New State => S5 Output => X
Movement => 0 New Position =>7 New Tape => (L X X 0 R L 0 X)
Time = 18
...
Input => X State => S7 New State => S8 Output => 0
Movement => 1 New Position =>3 New Tape => (L X 0 0 R L 0 X)
Time = 26
...
Input => X State => S9 New State => S10 Output => 0
Movement => 1 New Position =>8 New Tape => (L X 0 0 R L 0 0)
Time = 32

Input => NIL State => S10 New State => S5 Output => X
Movement => 0 New Position =>8 New Tape => (L X 0 0 R L 0 0 X)
Time = 33
...
Input => X State => S7 New State => S8 Output => 0
Movement => 1 New Position =>2 New Tape => (L 0 0 0 R L 0 0 X)
Time = 43

Input => X State => S9 New State => S10 Output => 0
Movement => 1 New Position =>9 New Tape => (L 0 0 0 R L 0 0 0)
Time = 51

Input => NIL State => S10 New State => S5 Output => X
Movement => 0 New Position =>9 New Tape => (L 0 0 0 R L 0 0 0 X)
Time = 52
...
Input => L State => S7 New State => S11 Output => L
Movement => 0 New Position =>0 New Tape => (L 0 0 0 R L 0 0 0 X)
Time = 64

Input => L State => S11 New State => S11 Output => L
Movement => 1 New Position =>1 New Tape => (L 0 0 0 R L 0 0 0 X)
Time = 65
...
Input => X State => S12 New State => S13 Output => 0
Movement => 1 New Position =>10 New Tape => (L 0 0 0 R L 0 0 0 0)
Time = 75

Input => NIL State => S13 New State => S13 Output => R
Movement => 0 New Position =>10 New Tape => (L 0 0 0 R L 0 0 0 0 R)
Time = 76

Input => R State => S13 New State => SHALT Output => R
Movement => 0 New Position =>10 New Tape => (L 0 0 0 R L 0 0 0 0 R)
Machine Halted
```

12.6 Recognize/Generate Simulation

This example demonstrates the recognize/generate machines from Theorem 5 and subsequent examples.

```
% Recognize/Generate machine from Fred Cohen's thesis %
%       SxI     N       O       D       %
%       ------------------------        %
%       S0,t    S1      t       +1      %
%       S0,ELSE S7      ELSE    0       %
```

```
%       S1,e    S2      e       +1      %
%       S1,ELSE S6      ELSE    -1      %
%       S2,s    S3      s       +1      %
%       S2,ELSE S5      ELSE    -1      %
%       S3,t    S8      t       +1      %
%       S3,ELSE S4      ELSE    -1      %
%       S4,*    S5      *       -1      %
%       S5,*    S6      *       -1      %
%       S6,*    S7      *       -1      %
%       S7 didn't recognize state       %
%       S8 did recognize state          %
%       S8,*    S9      O       +1      %
%       S9,*    S10     K       +0      %
%       S10,*   S10     *       0       %

% ---------------------------------------- %
% User modified code for a given TM starts here %
% ---------------------------------------- %

% the next state function of current state and input symbol %
(DEFUN NEXT-STATE (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 't)) 'S1)
        ((EQUAL STATE 'S0) 'S7)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'e)) 'S2)
        ((EQUAL STATE 'S1) 'S6)
        ((AND (EQUAL STATE 'S2) (EQUAL INPUT 's)) 'S3)
        ((EQUAL STATE 'S2) 'S5)
        ((AND (EQUAL STATE 'S3) (EQUAL INPUT 't)) 'S8)
        ((EQUAL STATE 'S3) 'S4)
        ((EQUAL STATE 'S4) 'S5)
        ((EQUAL STATE 'S5) 'S6)
        ((EQUAL STATE 'S6) 'S7)
        ((EQUAL STATE 'S7) 'S7)
        ((EQUAL STATE 'S8) 'S9)
        ((EQUAL STATE 'S9) 'S10)
        ((EQUAL STATE 'S10) 'S10)
        (T STATE)               % not S0 => state unchanged %
)
))

% the output function of the current state and input symbol %
(DEFUN OUTPUT (LAMBDA (STATE, INPUT)
(COND   ((EQUAL STATE 'S0) INPUT)
        ((EQUAL STATE 'S1) INPUT)
        ((EQUAL STATE 'S2) INPUT)
        ((EQUAL STATE 'S3) INPUT)
        ((EQUAL STATE 'S4) INPUT)
        ((EQUAL STATE 'S5) INPUT)
        ((EQUAL STATE 'S6) INPUT)
        ((EQUAL STATE 'S7) INPUT)
        ((EQUAL STATE 'S8) 'O)
        ((EQUAL STATE 'S9) 'K)
        ((EQUAL STATE 'S10) INPUT)
)
))

% the tape movement function of the current stato and input symbol %
(DEFUN MOVEMENT (LAMBDA (STATE, INPUT)
(COND   ((AND (EQUAL STATE 'S0) (EQUAL INPUT 't)) 1)
        ((EQUAL STATE 'S0) 0)
        ((AND (EQUAL STATE 'S1) (EQUAL INPUT 'e)) 1)
        ((EQUAL STATE 'S1) -1)
        ((AND (EQUAL STATE 'S2) (EQUAL INPUT 's)) 1)
        ((EQUAL STATE 'S2) -1)
        ((AND (EQUAL STATE 'S3) (EQUAL INPUT 't)) 1)
        ((EQUAL STATE 'S3) -1)
        ((EQUAL STATE 'S4) -1)
        ((EQUAL STATE 'S5) -1)
        ((EQUAL STATE 'S6) -1)
        ((EQUAL STATE 'S7) 0)
        ((EQUAL STATE 'S8) 1)
```

```
        ((EQUAL STATE 'S9) 0)
        ((EQUAL STATE 'S10) 0)
        (T 0)   % else, don't move %
)
))

% ---------------------------------------- %
% Basic structures and variables %
% ---------------------------------------- %

(SETQ TAPE '(t e s t))                  % initial tape %
(SETQ I 7)                              % the modulus %
(SETQ POSITION 0)                       % initial tape position %
(SETQ TRACE-TM T)                       % trace the TM activities %
(SETQ TIME 0)                           % initial time %

(RUN)
Input => t State => S0 New State => S1 Output => t Time = 0
Movement => 1 New Position =>1 New Tape => (t e s t)

Input => e State => S1 New State => S2 Output => e Time = 1
Movement => 1 New Position =>2 New Tape => (t e s t)

Input => s State => S2 New State => S3 Output => s Time = 2
Movement => 1 New Position =>3 New Tape => (t e s t)

Input => t State => S3 New State => S8 Output => t Time = 3
Movement => 1 New Position =>4 New Tape => (t e s t)

Input => NIL State => S8 New State => S9 Output => 0 Time = 4
Movement => 1 New Position =>5 New Tape => (t e s t 0)

Input => NIL State => S9 New State => S10 Output => K Time = 5
Movement => 0 New Position =>5 New Tape => (t e s t 0 K)

Input => K State => S10 New State => SHALT Output => K
Movement => 0 New Position =>5 New Tape => (t e s t 0 K)
Machine Halted

(RUN)
Input => t State => S0 New State => S1 Output => t Time = 0
Movement => 1 New Position =>1 New Tape => (t e a s e r)

Input => e State => S1 New State => S2 Output => e Time = 1
Movement => 1 New Position =>2 New Tape => (t e a s e r)

Input => a State => S2 New State => S5 Output => a Time = 2
Movement => -1 New Position =>1 New Tape => (t e a s e r)

Input => e State => S5 New State => S6 Output => e Time = 3
Movement => -1 New Position =>0 New Tape => (t e a s e r)

Input => t State => S6 New State => S7 Output => t Time = 4
Movement => -1 New Position =>0 New Tape => (t e a s e r)

Input => t State => S7 New State => SHALT Output => t
Movement => 0 New Position =>0 New Tape => (t e a s e r)
Machine Halted
```

12.7 A PC DOS2.1 Virus

The following batch command file implements a virus almost entirely in the command language of IBM-PC DOS2.1. The single exception to this is the use of the program DOMANY.C which tests for the existence of the file done, and does each of the commands following it only if done exists. This could be implemented without the domany program but the resulting command language program would be intolerably slow for demonstration purposes, and clarity would be lost. We have also reformatted the text for readability, and placed no more than one command per line except in the case of "domany". In this form, the program takes 14 lines, but by removing the lines which are for demonstration purposes only

(e.g. echo Nothing left to infect) and merging mergable lines, we could reduce its size to 6 lines. Following the command file is the text of the DOMANY program as written in the language "C".

the virus

```
echo off
echo This program (%0) is infected
for %%i in (*.bat) do
        domany †done    †/z/%%i copy†%%i†done
                         copy†%%i†/z/%%i
                        copy†%0.bat†%%i >> /tmp/log
if exist done goto part2
echo Nothing left to infect
goto done
:part2
del done
:done
copy /z/%0.bat /tmp/tmp.bat > /tmp/log
tmp %1 %2 %3 %4 %5 %6 %7 %8 %9
```

domany.c

```
#include "/c/stdio.h"
int     sfix(s1) char *s1;
{int i; for (i=0;s1[i]!='\0';i++) if (s1[i]=='†') s1[i]=' '; return(0);}
int     scheck(s1) char *s1;
{int i; if (s1[0]=='†') /*if no such file, go on*/
        {i=open(&(s1[1]),0); if (i < 0) return(-1); close(i); exit(0);}
if (s1[0]=='?') /*if is such file, go on*/
        {i=open(&(s1[1]),0); if (i >= 0) {close(i);return(-1);} exit(0);}
return(0);}

main(argc,argv) int argc; char **argv;
{int    i; argv++; for (i=0;i<argc;i++)
  if (scheck(*argv) == 0) {sfix(*argv); system(*argv++);} else argv++;}
```

12.8 Instrumentation Analysis Programs

There are three basic measurements done by the measurement programs at this time. They are called social, spreader, and detailed.

"Social" is set up to find how social users are with each other. It basically lists the number of times each user has used another users programs, and the number of times their programs have been used by other users. You would expect that the root, for example, would be used by many, but use others programs rarely (if ever)! This is intended to help find social users, and perhaps identify weak points against viral infection. By isolating the social users so that they cannot easily get infected, or by making them more aware and providing more checks for them, one might be able to slow a virus.

"Spreader" is a program made to measure the overall spreading of a virus, assuming it started at a given user. This is basically a summary of the detailed analysis in that it tells how far a virus would have gotten, and how much time it would have taken to get there if it had started at each of the users in the system. It is to be expected that socialites would have lower times and larger spreads than isolationists.

"Detailed" provides the exact details of the first infection of each user given a particular viral starting point. This lists each user that could have gotten infected, and the time at which the infection would have happened for each user in the system.

```
/* This program is used to generate sample data to verify that  the
analysis programs operate correctly */
main()
{long   int     buf[2];
int     i,f;
printf("%d",sizeof(buf));
f = creat("testin",0600);
for (i = 1;i < 500;i++)
        {buf[0] = ((29*i)+13) % 64;
```

```
        buf[1] = ((21*i)+7) % 32;
        buf[2] = i;
        write(f,&(buf[0]),12);
        }
close(f);
exit(1);
}
/*      Copyright(c) Fred Cohen 1984*/
/*      show.c - show fred what goes*/
getinfo()
{int    f,tim,ouid,nuid,i;
long    int     buf[2];
if ((f = open("testin",0)) < 0) exit(-1);
while(12 == read(f,&(buf[0]),12))
        {printf("%d\t%d\t%d\n",buf[0],buf[1],buf[2]);
        }
}

main()
{getinfo();
exit(1);
}
/*      Copyright(c) Fred Cohen 1984*/
/*      spread.c - sharing paths from each user vs. time*/
/*      social - how social are users*/
int     uses[256],used[256],totals,dt;
/*      I used them, they used me, totals, delta time*/
int     user[256],fulltime[256],howbad[256];

getsoci()
{int    f,oldtime,time,ouid,nuid;
long    int     buf[2];
if ((f = open("testin",0)) < 0) exit(-1);
dt = 0;
read(f,&(buf[0]),12);oldtime = buf[2];
while(12 == read(f,&(buf[0]),12))
        {nuid=buf[0];ouid=buf[1];time = buf[2];
        used[ouid] += 1;
        uses[nuid] += 1;
        totals += 1;
        }
dt = time - oldtime;
return(1);
}

showsoci()
{float  ratio;
int     i;
printf("data summary\ntotal sharings = %d\n",totals);
printf("total time = %d\n",dt);
ratio = totals/dt;
printf("sharing/time = %f\n",ratio);
printf("broken down by uses:\n");
printf("user\tuses\tused\n");
for (i = 0;i < 256;i++)
        {if ((uses[i] != 0) || (used[i] != 0))
                printf("%d\t%d\t%d\n",i,uses[i],used[i]);
        }
return(0);
}

getinfo(uid)
int     uid;
{int    f,oldtim,tim,ouid,nuid,i;
long    int     buf[2];
if ((f = open("testin",0)) < 0) exit(-1);
for (i = 0;i<256;i++) user[i] = 0;
read(f,&(buf[0]),12);oldtim = buf[2];
user[uid] = 1;
while(12 == read(f,&(buf[0]),12))
        {nuid=buf[0];ouid=buf[1];tim = buf[2];
```

```
        if ((user[ouid] != 0) && (user[nuid] == 0))
                {user[nuid] = (tim - oldtim)+1;
                fulltime[uid] = (tim-oldtim)+1;
                howbad[uid] += 1;}
        }
printf("user %d spread time to %d users = %d\n"
                ,uid,howbad[uid],fulltime[uid]);
close(f);
return(1);
}

showinfo(uid)
int     uid;
{float  ratio;
int     i;
if (fulltime[uid] == 1) return(0);
printf("user %d spreading summary:\n",uid);
printf("user\ttim\n");
for (i = 0;i < 256;i++)
        {if (user[i] != 0)
                printf("%d\t%d\n",i,user[i]);
        }
return(0);
}

main(argc,argv)
int     argc;
char    *argv[];
{int    i;
if (argc > 1) {getsoci();showsoci();}
for (i = 0;i < 256;i++)
        {getinfo(i);
        if (argc > 2) showinfo(i);
        }
exit(1);
}
```

We now present the results of instrumentation analysis as measured in two actual systems. The first example shows first the ".out" file, and then, the ".sum" file, while the second only includes the ".sum" file due to the large size of the corresponding ".out" file.

The output is a bit cryptic at first. The "inc" indicates the initiation of the experiment at some number of system clock ticks from some arbitrary date, and is simply subtracted from absolute times to produce the results herein. The analysis takes some time, and prints out messages to the user like "read in" to indicate that it is active. The total sharings indicates the number of times users ran programs belonging to other users, the total time is in "clock ticks" which correspond to milliseconds, and the sharings per time indicate the frequency with which sharing takes place. The figure indicates that information is shared between users about every 50 msec. This is misleading because user "0" is the system itself, and it is responsible for 65% of the cases of other users using its programs.

The categories indicated in the per user breakdown show the user number (user), the number of times that user used other users' programs (uses), the number of times that user's programs were used by other users (used), and the first time at which the user used another user's program is indicated by the "firstuse" heading.

We note especially that because of the separation of duties between various users on this system, the superuser had to use other users programs quite often, and that this is likely to result in rapid takeover of the entire system. In this case, a measure intended to maintain security via separation of duties actually compromises the system security by forcing increased sharing and thus more rapid viral attack.

We also note that negative numbers indicate activities that occurred before the system's clock was set at system startup, and should be disregarded in statistics (although they are important because they do indicate sharing in the initialization of the system that could cause viral takeover.

The "takeover time" and "spread to" indications show how far a best case viral attack by a given user using only the measured data paths could do. Note that many users could takeover the system very quickly after their first program is run by another user, and that some takeover times are quite long (over an hour). Many users don't take over at all, and many more users never used the system.

.OUT FILE

```
inc = 11591  - data read in  - data summary - total sharings = 11591
total time = 541091 - sharing/time = 0.021422 - broken down by uses:
user    uses    used    firstuse        user    uses    used    firstuse
0       2699    7549    7               3       2033    2725    14105
4       1489    0       2247            6       50      0       118974
8       600     1       2388            10      12      0       5286
19      186     0       18550           25      1082    1       2677
32      86      0       455651          33      805     0       3220
39      30      1       6289            40      653     0       3250
41      208     0       195819          48      112     0       83102
54      39      0       455832          103     16      0       2335
112     14      7       3173            135     527     1       3187
139     686     0       4840            206     1       0       25050
222     26      1306    92337           226     1       0       456436
392     236     0       460901
user 0 spread to 22 users in t = 460902  dt=458652
user 0 spreading summary:
user    tim     rel     best            user    tim     rel     best
0       1       -2249   -6              3       14105   11856   1
4       2250    0       3               6       118975  116725  1
8       2389    139     1               10      5287    3037    1
19      18551   16301   1               25      3154    904     477
32      455652  453402  1               33      8259    6009    5039
39      437464  435214  431175          40      4722    2472    1472
41      195820  193570  1               48      83103   80853   1
54      455833  453583  1               103     2336    86      1
112     120511  118261  117338          135     4579    2329    1392
139     4844    2594    4               206     25051   22801   1
222     92338   90088   1               226     456437  454187  1
392     460902  458652  1
user 3 takeover at 1    rel=0
user 3 spread to 22 users in t = 460902  dt=460901
user 3 spreading summary:
user    tim     rel     best            user    tim     rel     best
0       1       0       -6              3       1       0       -14104
4       2248    2247    1               6       118975  118974  1
8       2389    2388    1               10      5287    5286    1
19      18551   18550   1               25      3150    3149    473
32      455662  455651  1               33      8256    8255    5036
39      6290    6289    1               40      4722    4721    1472
41      195820  195819  1               48      83103   83102   1
54      455833  455832  1               103     2336    2335    1
112     120511  120510  117338          135     4579    4578    1392
139     4841    4840    1               206     26051   26050   1
222     92338   92337   1               226     456437  456436  1
392     460902  460901  1
user 8 spread to 1 users in t = 184432   dt=0
user 8 spreading summary:
user    tim     rel     best
8       1       -184431 -2387
135     184432  0       181245
user 25 spread to 1 users in t = 447539  dt=0
user 25 spreading summary:
user    tim     rel     best
25      1       -447538 -2676
40      447539  0       444289
user 39 takeover at 455457      rel=9229
user 39 spread to 15 users in t = 536364         dt=80907
user 39 spreading summary:
user    tim     rel     best            user    tim     rel     best
0       455457  0       455450          3       456159  702     442054
4       458247  2790    456000          8       457229  1772    454841
19      536364  80907   517814          25      513074  57617   510397
32      455652  195     1               33      455789  332     452569
39      1       -455456 -6288           41      456335  878     260516
```

```
48      455743  286     372641          64      455833  376     1
103     460400  4943    458065          139     513290  57833   508460
226     456437  980     1               392     460902  5446    1
user 112 spread to 2 users in t = 8156   dt=567
user 112 spreading summary:
user    tim     rel     best
8       7689    0       6201
112     1       -7588   -3172
135     8156    567     4969
user 135 spread to 1 users in t = 5344   dt=0
user 135 spreading summary:
user    tim     rel     best
10      5344    0       58
135     1       -5343   -3186
user 222 takeover at 2677       rel=53
user 222 spread to 22 users in t = 460902       dt=458225
user 222 spreading summary:
user    tim     rel     best            user    tim     rel     best
0       2677    0       2670            3       14106   11429   1
4       3236    558     988             6       118975  116298  1
8       3202    525     814             10      5287    2610    1
19      18551   15874   1               25      2678    1       1
32      456662  452976  1               33      3221    544     1
39      437464  434787  431175          40      3251    574     1
41      195820  193143  1               48      83103   80426   1
54      455833  453156  1               103     13762   11085   11427
112     3174    497     1               135     3188    511     1
139     4844    2167    4               206     25051   22374   1
222     1       -2676   -92336          226     456437  453760  1
392     460902  458225  1
                                        .SUM FILE
inc = 11591 - data read in - data summary - total sharings = 11591
total time = 541091 - sharing/time = 0.021422 - broken down by uses:
user    uses    used    firstuse        user    uses    used    firstuse
0       2699    7549    7               3       2033    2725    14105
4       1489    0       2247            6       50      0       118974
8       600     1       2388            10      12      0       5286
19      186     0       18550           25      1082    1       2677
32      86      0       455651          33      805     0       3220
39      30      1       6289            40      653     0       3250
41      208     0       195819          48      112     0       83102
54      39      0       455832          103     16      0       2335
112     14      7       3173            135     527     1       3187
139     686     0       4840            206     1       0       25050
222     26      1306    92337           226     1       0       456436
392     236     0       460901
user 0 spread to 22 users in t = 460902  dt=458862
user 3 takeover at 1    rel=0
user 3 spread to 22 users in t = 460902  dt=460901
user 8 spread to 1 users in t = 184432   dt=0
user 25 spread to 1 users in t = 447539  dt=0
user 39 takeover at 455457      rel=9229
user 39 spread to 15 users in t = 535364        dt=80907
user 112 spread to 2 users in t = 8156   dt=567
user 135 spread to 1 users in t = 5344   dt=0
user 222 takeover at 2677       rel=53
user 222 spread to 22 users in t = 460902       dt=458225
                                        ANOTHER .SUM FILE
inc = 44556 - data read in - data summary - total sharings = 44556
total time = 283789 - sharing/time = 0.157004 - broken down by uses:
user    uses    used    firstuse        user    uses    used    firstuse
0       13459   12403   2               3       53      26335   192758
4       377     0       527             5       44      23      5325
6       944     144     1252            7       156     0       2173
8       15      3       200472          9       1560    0       8100
10      5       0       100336          11      4       0       181052
14      839     1       172641          15      3       0       181817
16      82      0       803             17      81      0       175313
19      646     0       93010           23      358     0       8960
24      66      0       50580           25      17      0       201225
```

```
27      79      11      990             28      56      0       19880
29      121     0       10106           30      16      0       203108
32      696     2       179772          33      640     64      95266
...
43      609     0       2822            45      8       2053    36339
...
52      2       0       188599          54      7       5       188564
...
68      1       0       187585          72      8       2       40688
...
103     24      0       39348           112     0       1       0
...
138     1       0       74870           139     564     23      50578
...
176     46      1       69538           177     58      0       132216
...
222     7       2132    52194           224     25      0       175138
227     44      0       50575           233     124     0       175407
235     106     3       993             240     27      0       267250
...
305     584     0       173109          312     10      1349    4236
...
340     13      0       271546          345     8       1       40692
user 0 spread to 160 users in t = 283062        dt=282534
user 3 takeover at 1    rel=0
user 3 spread to 161 users in t = 263062        dt=283061
user 5 takeover at 8    rel=9
user 5 spread to 160 users in t = 283062        dt=283054
user 6 takeover at 169614       rel=20632
user 6 spread to 152 users in t = 263062        dt=258774
user 8 spread to 1 users in t = 204615   dt=0
user 14 spread to 1 users in t = 276624  dt=0
user 27 spread to 2 users in t = 186375  dt=7621
user 32 spread to 1 users in t = 179406  dt=0
user 33 takeover at 268035      rel=39755
user 33 spread to 76 users in t = 263169        dt=263245
user 45 takeover at 5   rel=6
user 45 spread to 160 users in t = 263062       dt=283057
user 54 spread to 8 users in t = 280918  dt=12756
user 72 spread to 1 users in t = 196123  dt=0
user 112 spread to 1 users in t = 192445        dt=0
user 139 takeover at 125166     rel=16933
user 139 spread to 154 users in t = 283062      dt=157894
user 176 spread to 1 users in t = 169722        dt=0
user 222 takeover at 897        rel=69
user 222 spread to 160 users in t = 283062      dt=282165
user 235 spread to 3 users in t = 273561        dt=154661
user 312 takeover at 1572       rel=126
user 312 spread to 160 users in t = 283062      dt=261490
user 345 takeover at 316        rel=25
user 345 spread to 160 users in t = 283062      dt=262746
```

A further experiment was planned wherein a program would be introduced to the system via the bulletin board, and its uses traced to indicate the spread of a nonviral program introduced to the users in this way. Unfortunately, one of the administrative users who was not supposed to know of the experiment violated the privacy of the account used to store the sources of the trace program, detected that the writer of the program was the author (via the copyright notice), and warned all users not to use the program because of its author, without checking the program to find that it was not in fact a threat to the system, but rather just a program that performed as advertised. Although this administrator probably did the "safe" thing, he certainly violated the privacy of the author, invalidated the experiment, and along with a lack of time, prevented the experiment from yielding any useful results.

The author regrets the tendency of users of every system he ever uses to shun his programs, simply because of his reputation for being able to take over systems. Woe be, to the bearer of bad news!

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. E. Bell and L. J. LaPadula.
Secure Computer Systems: Mathematical Foundations and Model.
The Mitre Corporation, 1973.
cited in many papers.

[4] T. V. Benzel.
Further Analysis of the SCOMP System Verification.
In *7th Security Conference.* DOD/NBS, Sept, 1984.

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

[6] IBM.
Bitnet communications network.
IBM, 1984.

[7] D. K. Branstad.
Security of Computer Communications.
In *Communications*, pages 33-40. IEEE, Nov, 1978.

[8] P. Brinch-Hansen.
Operating System Principles.
Prentice Hall, 1973.

[9] B. Catchings, B. Cattani, C. Maio, F. Cruz, A. Crosswell, and J. Guyton.
Kermit File Transfer Utility.
Columbia University, 1984.

[10] D. Chaum - Editor.
Several articles.
In *Advances in Cryptology.* IACR, Plenum Press, Aug, 1984.

[11] D. Chaum.
Title unknown.
PhD thesis, UCSB, 1983.

[12] M. Cornwell and R. Jacob.
Towards Multilevel-Secure Message Systems: Techniques Employed in Prototype Systems.
In *7th Computer Security Conference.* DOD/NBS, Sept, 84.

[13] Regents of California.
CSnet communications network.
Unix, 1984.

[14] C. Darwin.
The Origin of Species.
John Murray, 1959.

[15] D. W. Davies.
Use of the 'Signature Token' to Create a Negotiable.
In D. Chaum (editor), *Advances in Cryptology*, pages 377-382. IACR, Aug, 1983.

[16] M. Davio, Y. Desmedt, M. Fosseprez, R. Govaerts, J. Hulsbosch, P. Neutjens, P. Piret, J. Quisquater, J. Vandevalle, and P Wouters.
Analytical Characteristics of the DES.
In *Advances in Cryptology*, pages 171-202. IACR, Plenum Press, Aug, 1983.

[17] Richard Dawkins.
The Selfish Gene.
Oxford Press, N.Y., N.Y., 1978.

[18] R. DeMillo and M. Merritt.
Protocols for Data Security.
In *Computer.* Feb, 1983.

[19] D. E. Denning.
Cryptography and Data Security.
Addison Wesley, 1982.

[20] D. E. Denning.
Secure Information Flow in Computer Systems.
PhD Thesis, Purdue Univ, W. Lafayette, Ind., 1975.

[21] Dewdney.
Metamagical Themas.
Scientific American, 1983-1984.

[22] W. Diffie and M. Hellman.
New Direction in Cryptography.
In *Transactions on Information Theory*, pages 644-654. IEEE, Nov, 1976.

[23] W. Diffie and M. Hellman.
Exhaustive Cryptanalysis of the NBS Data Encryption Standard.
In *Computer.* June, 1977.

[24] 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.

[25] E. Feinler - Editor.
Arpanet Resource Handbook.
Network Information Center, SRI International, 1978.
Prepared for the DCA.

[26] H. Feistel, W. A. Notz, and J. L. Smith.
Some Cryptographic Techniques for Machine-to-Machine Data Communications.
In *Communications*, pages 1545-1554. IEEE, Nov, 1975.

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

[28] M. R. Garey and D. S. Johnson.
Computers and Intractability.
Freeman, 1979.

[29] D. K. Gifford.
Cryptographic Sealing for Information Secrecy and Authentication.
In *Communications.* ACM, 1982.

[30] 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.

[31] J.B. Gunn.
Use of Virus Functions to Provide a Virtual APL Interpreter Under User Control.
In *Communications*, pages 163-168.. ACM, July, 1974.

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

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

[34] Hofstadter.
Goedel, Escher, and Bach.
Vintage, 1979.

[35] D. Hofstadter.
Metamagical Themas.
Scientific American, Metamagical Themas.

[36] U.S. Dept. of Justice, Bureau of Justice Statistics.
Computer Crime - Computer Security Techniques.
U.S. Government Printing Office, Washington, DC, 1982.

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

[38] D. Knuth.
Seminumerical Algorithms.
Addison-Wesley, 1969.

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

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

[41] R. R. Linde.
Operating System Penetration.
In *National Computer Conference*, pages 361-368. AIFIPS, 1975.

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

[43] R. C. Merkle.
Protocols for Public Key Systems.
In *Symposium on Security and Privacy*. IEEE, 1980.

[44] R. M. Needham and M. D. Schroeder.
Using Encryption for Authentication in a Large Network of Computers.
In *Communications*, pages 993-999. ACM, Dec, 1978.

[45] 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.

[46] R. L. Rivest, A. Shamir, and L. Adleman.
A Method for Obtaining Digital Signatures and Public Key Cryptosystems.
In *Comm. of the ACM*. Feb, 1978.

[47] A. Shamir.
How to Share a Secret.
In *Communications*, pages 612-613. ACM, Nov, 1979.

[48] C. E. Shannon.
A Mathematical Theory of Communications.
Tech. Journal 27 3, Bell Systems Technical Journal, July, 1948.

[49] C. E. Shannon.
Communications Theory of Secrecy Systems.
Technical Report, Bell Systems Technical Journal, 1949.

[50] J F Shoch and J A Hupp.
The 'Worm' Programs - Early Experience with a Distributed Computation.
In *Communications*, pages 172-180. ACM, March, 1982.

[51] G. Simmons.
Verification of the Nuclear Test Ban Treaty.
In *Oakland Conference on Computer Security*. IEEE, Aug, 1981.

[52] K. Thompson.
Reflections on Trusting Trust.
In *Communications*. ACM, Aug, 1984.

[53] A.M. Turing.
On Computable Numbers, with an Application to the Entscheidungsproblem.
In *Proceedings of the London Mathematical Society*, pages 230-265. London Math Soc, Nov 12, 1936.
Series 2 Vol 42.

[54] S. Walker, P. Baker, J. P. Anderson, and S. Brand.
Introduction to Network Security Evaluation.
In *7th Security Conference*. DOD/NBS, Sept, 1984.

[55] H. C. Williams.
An Overview of Factoring.
In D. Chaum (editor), *Advances in Cryptology*, pages 71-80. IACR, Aug, 1983.

[56] J. P. L. Woodward.
Applications for Multilevel Secure Operating Systems.
In *National Computer Conference*, pages 319-328. AFIPS, 1979.