Computers & Security, 8 (1989) 325–344

Refereed Article

Computational Aspects of Computer Viruses

Fred Cohen

The Radon Project, Pittsburgh, PA, U.S.A.

This paper formally defines a class of sets of transitive integrity-corrupting mechanisms called "viral sets" and explores some of their computational properties.

1. Introduction

A "virus" may be loosely defined as a sequence of symbols which, upon interpretation, causes other sequences of symbols to contain (possibly evolved) virus(es). If we consider an interpreted sequence of symbols in an information system as a "program", viruses are interesting to computer systems because of their ability to attach themselves to programs and cause them to contain viruses as well.

We begin the discussion with an informal discussion of "viruses" [1] 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 if they attack a system. We then formally define a trivial generalization of "Turing machines", define "viral sets" in terms of these machines, and explore some of their properties. We define a computing machine and a set of (machine, tape-set) pairs which comprise "viral sets" (VS). We then define the terms "virus" and "evolution" for convenience of discussion. We show that the union of VSs is also a VS, and that therefore a "largest" VS (LVS) exists for any machine with a viral set. We then 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 sequences 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 therefore, viruses are at least as powerful as Turing machines as a means for computation.

2. 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 unspecified.

```
program virus:=
{1234567;

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

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

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

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

next:}
```

A Simple Virus "V"

This example virus "V" searches for an uninfected executable file "E" by looking for executable files without the "1234567" at 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 (prepend is used to mean "attach at the beginning"). 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 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 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
        then goto loop;
        compress file;
        prepend compression-virus to file;
        }

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

A Compression Virus "C"

This program "C" finds an uninfected executable "E", compresses it, and prepends C to form an

infected executable "I". It then uncompresses the rest of itself into a temporary file and executes normally. When I is run, it will seek out and compress another executable before decompressing E into a temporary file and executing it. The effect is to spread through the system, compressing executable files and decompress them as they are to be executed. 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;}
...
```

A Denial of Services Virus

As an analogy to this virus, consider a biological disease that is 100% infectious, spreads whenever people communicate, kills all infected persons instantly at a given moment, and has no detectable side effects until that moment. If a delay of even 1 week were used between the introduction of the disease and its effect, it would be very likely to leave only a few people in 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.

3. Symbols Used in Computability Proofs

Throughout the remainder of this paper, 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 "}" and the elements of sets by symbols separated by commas within the scope of these brackets (*e.g.* {a,b} stands for the set comprising elements a and 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 they are both sets and elements of sets, and 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 (*i.e.* {0, 1,...}) and II will be used to represent the integers (*i.e.* {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 ordered *n*-tuples (sequences), and elements of the sequence will be separated by ","s (*e.g.* (1,2,...) is the sequence of integers starting with 1).

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 ∪ 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).

4. Computing Machines

We begin our discussion with a definition of a computing machine [2] which will serve as our basic computational model for the duration of the discussion. The basic class of machines we will be discussing is the set of machines which consist of a finite state machine (FSM) with a "tape head" and a semi-infinite tape (Fig. 1). The tape head is pointing at one tape "cell" at any given instant of time, and is capable of reading and writing any of a finite number of symbols from or to the tape, and of moving the tape one cell to the left (−1), right (+1), or keeping it stationary (0) on any given "move". The FSM takes input from the tape, sets its next state, produces output on the tape and moves the tape as functions of its internal state and maps.

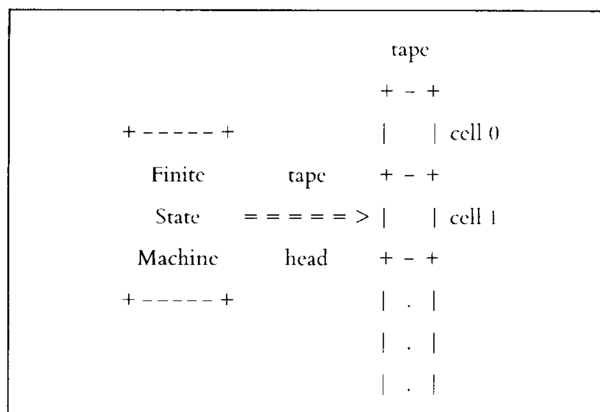


Fig. 1. A Turing machine.

A set of computing machines "TM" is defined as follows:

$$\forall M[M \in TM] \text{ iff}$$
$$M: (S_M, I_M, O_M: S_M \times I_M \rightarrow I_M,$$
$$N_M: S_M \times I_M \rightarrow S_M,$$
$$D_M: S_M \times I_M \rightarrow d)$$

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 tape, to the tape symbol in that cell after that move

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

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 $(\mathbb{S}_M, \square_M, P_M)$, the "history" (H_M) of the machine, and the H_M for a particular move number (or instant in time) 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

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

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

$$\begin{aligned}
&\forall t \in \mathbb{N} \\
&\quad [\mathbb{S}_M(t+1) = N(\mathbb{S}_M(t), \square_M(t, P_M(t)))] \text{ and} \\
&\quad [\square_M(t+1, P_M(t)) = 0(\mathbb{S}_M(t), \square_M(t, P_M(t)))] \text{ and} \\
&\quad [\forall j \neq P_M(t), \square_M(t+1, j) = \square_M(t, j)] \text{ and} \\
&\quad [P_M(t+1) = \text{Sup}(0, P_M(t) + D(\mathbb{S}_M(t), \square_M(t, P_M(t))))]
\end{aligned}$$

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 time.

We will say that "M Halts at time t" iff

$$\begin{aligned}
&[\forall t' > t \\
&\qquad [\mathbb{S}_M(t) = \mathbb{S}_M(t')] \text{ and} \\
&\qquad [\forall i \in \mathbb{N} [\square_M(t, i) = \square_M(t', i)]] \text{ and} \\
&\qquad [P_M(t) = P_M(t')]]
\end{aligned}$$

and that "M Halts" iff

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

We say that "x runs at time t" iff

$$\begin{aligned}
&[[x \in I_M{}^i \text{ where } i \in \text{II}] \text{ and} \\
&[\mathbb{S}(t) = \mathbb{S}_0] \text{ and} \\
&[(\square(t, P(t)), \ldots, \square(t, P(t) + i)) = x]]
\end{aligned}$$

and that "x runs" iff

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

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:

$$\begin{aligned}
&[\forall M \in TM [\forall v [\forall i \in \text{II} \\
&\qquad [v \in TP_M] \text{ iff} [v \in I_M{}^i]]]]
\end{aligned}$$

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

$$\begin{aligned}
&[\forall M \in TM [\forall V [V \in TS] \text{ iff} \\
&\qquad \text{i) } [\exists v \in V] \text{ and} \\
&\qquad \text{ii) } [\forall v \in V [v \in TP_M]]]]
\end{aligned}$$

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 therefore abbreviate throughout this paper by removing the subscript when it is unnecessary.

5. Formal Definition of Viruses

We now define the central concept under study, the "viral set". In earlier statements, we have informally defined a "virus" as a "program" that modifies other "programs" so as to include a (possibly "evolved") version of itself. The mathematical embodiment of this definition for Turing machines, given below, attempts to maintain the generality of this definition.

Several previous attempts at definition failed because the idea of a singleton "virus" makes the understanding of "evolution" of viruses very difficult, and as we hope to 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 sequences of tape symbols we call "viruses" depend on the machine on which they are to be interpreted. 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 pair "VS" as follows:

[1] $\forall M \forall V$
[2] $(M,V) \in VS$ iff
[3] $[V \in TS]$ and $[M \in TM]$ and
[4] $[\forall v \in V [\forall H_M$
[5] $[\forall t \forall j$
[6] $[\quad P_M(t) = j$ and
[7] $S_M(t) = S_{MO}$ and
[8] $(\square_M(t,j), \ldots, \square_M(t, j+|v|-1)) = v$
[9] $] \Rightarrow$
[10] $[\quad \exists v' \in V [\exists t' > t [\exists j'$
[11] $[\quad [[(j'+|v'|) \leqslant j]$ or $[(j+|v|) \leqslant j']]$ and
[12] $(\square_M(t',j'), \ldots, \square_M(t', j'+|v'|-1)) = v'$ and
[13] $[\exists t''$ s.t. $[t < t'' < t']$ and
[14] $[P_M(t'') \in \{j', \ldots, j'+|v'|-1\}]]$
[15] $]]]\ ]\quad]$

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 the tape head is in front of cell j at time t and
[7] TM is in its initial state at time t and
[8] 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' such that
[11] at place j' far enough away from v
[12] the tape cells starting at j' hold virus v′
[13] and at some time t'' between time t and time t'
[14] v′ is written by M

For convenience of space, we will use the expression

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

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:

$$[\forall B[\forall C \quad [(M,C)\in VS] \text{ iff } \quad [[C\in TS] \text{ and } [M\in TM] \text{ and } \quad [\forall a\in C[a \overset{B}{\Rightarrow} C]]]]]$$

We have defined 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 could easily 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" with respect to M

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

and define the term "virus" with respect to M as

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

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

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

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

$$[(M,V)\in VS \quad [\exists i\in \mathbb{N}[\exists V'\in V^i \quad [v\in V] \text{ and } [v'\in V] \text{ and } \quad [\forall v_k\in V'[v_k \overset{M}{\Rightarrow} v_{k+1}]] \text{ and } \quad [\exists l\in \mathbb{N} \quad [\exists m\in \mathbb{N} \quad [[l < m] \text{ and } [v_l = v] \text{ and } [v_m = v']]]]]]]$$

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

6. Basic Theorems

Our most basic theorem states that any union of viral sets is also a viral set

Theorem 1:

$$\forall M \forall U^* \quad [\forall V\in U^* \text{s.t.} (M,V)\in VS] \Rightarrow [(M, \cup U^*)\in VS]$$

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,

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

by assumption,

$$[\forall V\in U^* \quad [\forall v\in V[v \overset{M}{\Rightarrow} V]]]$$

thus since

$$[\forall v\in U[\exists V\in U^*[v \overset{M}{\Rightarrow} V]]]$$

and $[\forall V\in U^*[V\subset U]]$

$$[\forall v\in U[\exists V\subset U[v \overset{M}{\Rightarrow} V]]]$$

hence $[\forall v \in U[v \overset{M}{\Rightarrow} U]]$

thus by definition, $(M,U) \in VS$
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 with respect to that machine.

Lemma 1.1:
$[\forall M \in TM$
$\quad [[\exists V \text{ s.t. } [(M,V) \in VS]] \Rightarrow$
$\quad [\exists U$
$\qquad$ i) $[(M,U) \in VS]$ and
$\qquad$ ii) $[\forall V[[(M,V) \in VS] \Rightarrow$
$\qquad\quad [\forall v \in V[v \in U]]]]]]]$

We call U the "largest viral set" (LVS) w.r.t. M, and define

$(M,U) \in LVS$ iff[i and ii]

Proof:
assume $[\exists V[(M,V) \in VS]]$
choose $U = \cup\{V \text{ s.t. } [(M,V) \in VS]\}$
now prove i and ii
Proof of i: (by Theorem 1)
$\quad (M,[\cup\{V \text{ s.t. } [(M,V) \in VS]\}) \in VS$
thus $\quad (M,U) \in VS$

Proof of ii by contradiction:
assume ii) is false:
thus $\quad [\exists V$ s.t.
$\qquad$ 1) $[(M,V) \in VS]$ and
$\qquad$ 2) $[\exists v \in V \text{ s.t. } [v \notin U]]]$
but $\quad [\forall V \text{ s.t. } (M,V) \in VS$
$\qquad [\forall v \in V[v \in U]]]$ (definition of union)
thus $\quad [v \notin U]$ and $[v \in U]$ (contradiction)
thus ii) is true
Q.E.D.

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

We define SVS as follows:

$[\forall M[\forall V$
$\quad [(M,V) \in SVS]$ iff
$\qquad$ 1) $[(M,V) \in VS]$ and
$\qquad$ 2) $[(\nexists U$ s.t.
$\qquad\quad [U \subset V]$ (proper subset) and
$\qquad\quad [(M,U) \in VS]]]]$

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:

$[\exists M[\exists V$
$\quad$ i) $[(M,V) \in SVS]$ and
$\quad$ ii) $[|V| = 1]]]$

Proof: by demonstration
$\quad M: S = \{s_0, s_1\}, I = \{0,1\}$,

$S \times I$	N	O	D
$s_0,0$	s_0	0	0
$s_0,1$	s_1	1	+1
$s_1,0$	s_0	1	0
$s_1,1$	s_1	1	+1

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

$[(M,\{(1)\}) \in SVS]$ iff
$\quad$ 1) $[(M,\{(1)\}) \in VS]$ and
$\quad$ 2) $[(M,\{\}) \notin VS]$

$(M,\{\}) \notin VS$ (by definition since $\{\} \notin TS$)

as can be verified by the reader:

$\quad (1) \overset{M}{\Rightarrow} \{(1)\} \quad (t' = t+2, t'' = t+1, j' = j+1)$
thus $\quad (M,\{(1)\}) \in VS$
Q.E.D.

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 with respect to the machine on which it is self-duplicating.

Lemma 2.1:

$$[\forall M \in TM [\forall u \in TP \quad [[u \overset{M}{\Rightarrow} \{u\}] \Rightarrow [(M,\{u\}) \in VS]]]]$$

Proof:
by substitution into the definition of viruses:

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

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 spurs 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 an SVS 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. 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:

$$[\forall i \in II \quad [\exists M \in TM [\exists V \quad 1)\ [(M,V) \in SVS] \text{ and } 2)\ [|V| = i]]]]$$

Proof: By demonstration

$M: S = \{s_0, s_1, \ldots, s_i\}, I = \{0, 1, \ldots, i\},$
$\forall x \in \{1, \ldots, i\}$

S×I	N	O	D	
$S_0,0$	s_0	0	0	;if I = 0, halt
S_0,x	s_x	x	+1	;if I = x, goto state x, move right
...				;other states generalized as:
s_x,*	s_x	$[x\|i]+1$	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) $[\nexists U [[U \subset V] \text{ and } [(M,U) \in VS]]]$

proof of "1) (M,V) ∈ VS"

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

...

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

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

and $(1) \in V, \ldots,$ 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) $[\nexists U[[U \subset V]$ 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 can 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 $[\nexists x' \neq [x|i]+1[x \overset{M}{\Rightarrow} \{x'\}]]$

now $[\forall (x) \in V$

$[([x|i]+1) \notin V \Rightarrow [(x) \notin V]]]$

assume $[\exists U \subset V[(M,U) \in VS]]$

$[U = \{\}] \Rightarrow [(M,U) \notin VS]$ thus $U \neq \{\}$

by definition of proper subset

$[U \subset V] \Rightarrow [\exists v \in V[v \notin U]]$

but $[\exists v \in V[v \notin U]]$

$\Rightarrow [\exists v' \in U[[v'|i]+1 = v]$

and $[v \notin U]$

and $[\nexists v'' \in V[v' \overset{M}{\Rightarrow} v'']]]$

thus $[\nexists v \in U[v' \Rightarrow V]]$

and $[v' \in U]$

thus $[(M,U) \notin VS]$ which is a contradiction

Q.E.D.

7. Abbreviated Table Theorems

We 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. These "macros" are simply 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 ref. [2], and we refer the reader to this 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 S_n. 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 S_n so that the next state will always be S_n. 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	S_n,*	S_n	*	0

(halt the machine)

The "right till x" macro describes a machine which increments the tape position (P(t)) until such posi-

tion 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 S_n 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)	S_n,x	S_{n+1}	x	0
	S_n, else	S_n	else	+1

(R(x): right till x)

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)	S_n,x	S_{n-1}	x	0
	S_n,else	S_n	else	−1

(L(x): left till x)

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)	S_n,z	S_{n-1}	z	0
	S_n,x	S_n	y	+1
	S_n,else	S_n	else	+1

(C(x,y,z): change x to y till z)

The "copy from x till y 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 the 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 until 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)		
	S_n	R(x)			; right till x
	S_{n+1}	S_{n+2}	"N"	0	; write "N"
	S_{n+2}	R(y)			; right till y
	S_{n+3}	R(z)			; right till z
	S_{n+4}	S_{n+5}	z	+1	; right one more
	S_{n-5}	S_{n+6}	"M"	0	; write "M"
	S_{n+6}	L("N")			; left till "N"
	S_{n+7}	S_{n+8}	x	0	; replace the initial x
	S_{n+8}, y	S_{n+9}	y	+1	; if y, done
	S_{n+8}, *	S_{k+5^*}	"N"	−1	; else write "N" an
					; goto S_{n+5} times the input
					; symbol number
	S_{n+9}	R(M)			; right till "M"
	S_{n-10}	S_{n+11}	y	0	; copy completed
	S_{k+5^*}	R("M")			; goto the "M"
	S_{k+5^*+1}	S_{k+5^*+2}	*	+1	; write the copied symbol
	S_{k+5^*-2}	S_{k+5^*+3}	"M"	0	; write the trailing "M"
	S_{k+5^*+3}	L("N")			; left till "N"
	S_{k+5^*+4}	S_{n+8}	*	+1	; rewrite * and go on

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 integers from the generated set.

Theorem 4:

$[\exists M \in TM \, \exists V \in TS$ s.t.

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

2) $[|V| = |\mathbb{N}|]$

Proof by demonstration:

	S,I	N	O	D	
M:	S_0,L	S_1	L	+1	; start with L
	S_0,else	S_0	x	0	; or halt
	S_1,0	C(0,x,R)			; change 0s to xs till R
	S_2,R	S_3	R	+1	; write R
	S_3	S_4	L	+1	; write L
	S_4	S_5	X	0	; write x
	S_5	L(R)			; move left till R
	S_6	L(x or L)			; move left till x or L
	S_7,L	S_{11}	L	0	; if L goto s11
	S_7,x	S_8	0	+1	; if x replace with 0
	S_8	R(x)			; move right till x
	S_9,x	S_{10}	0	+1	; change to 0, move right
	S_{10}	S_5	x	0	; write x and goto S5
	S_{11}	R(x)			; move right till x
	S_{12}	S_{13}	0	+1	; add one 0
	S_{13}	S_{13}	R	0	; halt with R on tape

V = {(LOR), (LOOR),..., (LO...OR),...}

proof of 1) (M,V) ∈ VS
definition:

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

by inspection,
$\quad [V \in TS]$

now $\quad [\forall (LO...OR)[\exists (LO...OOR) \in V$
$\quad\quad [(LO...OR) \overset{M}{\Rightarrow} \{(LO...OOR)\}]]]$
$\quad\quad$ as may be verified by simulation

thus $\quad [(M,V) \in VS]$

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

$[\forall v_n \in V [\exists v_{n-1} \in V$
$\quad [\forall k \leqslant n$
$\quad\quad [\nexists v_k \in V [v_k = v_{n-1}]]]]$

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

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 Theorem 4) or by simply enumerating all non viruses (by Lemma 4.1).

Lemma 4.1:

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

Proof:

using M from Theorem 4, we choose
$\quad W = \{(x),(xx),\ldots,(x\ldots x),\ldots\}$

clearly $\quad [M \in TM]$ and $[W \in TS]$ and $[|W| = |\mathbb{N}|]$

since (from the state table)
$\quad [\forall w \in W$ [w runs at time t] $\Rightarrow$
$\quad\quad$ [w halts at time t]]
$\quad [\nexists t' > t [P_M(t') \neq P_M(t)]]$

thus $\quad [\forall w \in W [\nexists W' \subset W [w \Rightarrow W']]]$
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 non-viral.

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:

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

Proof by demonstration:

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

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

We 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 with respect to 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:
$v = \{v_0, v_2, \ldots, v_k\}$ where $[k \in \mathbb{N}]$ and $[v \in I^*]$
(definition of TP)

	S,I	N	O	D	
M:	s_0,v_0	s_1	v_0	+1	(recognize 1st element of v)
	s_0,else	s_0	0	0	(or halt)
	...				(etc till)
	s_k,v_k	s_{k+1}	v_k	+1	(recognize kth element of v)
	s_k,else	s_0	0	0	(or halt)
	s_{k+1}	s_{k+2}	v_0	+1	(output 1st element of v)
	...				(etc till)
	s_{k+k}	s_{k+k}	v_k	−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 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				
S_n, v_0	S_{n-1}	v_0	+1	(recognize 0th element)
S_n,*	$S_{n+z+z-1}$	*	0	(or rewind 0)
...			(etc till)	
S_{n+k}, v_k	S_{n+k+1}	v_k	+1	(recognize *k*th element)
S_{n-k},*	$S_{n+z+z-k}$	*	−1	(or rewind tape)
...			(etc till)	
S_{n+z-1}, v_z	S_{n+z-z}	v_z	+1	(recognize the last one)
S_{n+z-1},*	S_{n+z}	v_z	+1	(or rewind tape)
S_{n+z},*	S_{n+z+1}	*	−1	(rewind tape one square)
...				(for each of *k* states)
$S_{n+z+z-1}$		("didn't recognize" state)		
S_{n+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*			
S_n	S_{n+1}	v_0	+1
...			
S_{n+k}	S_{n+k+1}	v_k	+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)			
S_n recognize(v)			
$S_{n+2\|v\|-1}$,*	else-state	*	0
$S_{n+2\|v\|}$,*	then-state	*	0

The "pair" macro simply appends one sequence of states to another and thus combines 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)			
S_n	a		
S_m	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 (v_0) (pair(generate(v_0),halt)) else
  if (v_1) (pair(generate(v_1), halt)) else
    ...
    if (v_k) (pair(generate(v_k),halt)) else (halt)

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

if (v_0) (pair(generate(v_1),halt)) else
    if (v_1) (pair(generate(v_2),halt)) else
        ...
        if (v_k) (pair(generate(v_0),halt)) else (halt)

and a machine which generates any desired dependency.

if (v_0) (pair(generate(v_x),halt)) else
    if (v_1) (pair(generate(v_y),halt)) else
        ...
        if (v_k) (pair(generate(v_z),halt)) else (halt)
where $v_x, v_y, \ldots, v_z \in \{v_1, \ldots, v_k\}$

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

Lemma 5.1:

$$[\exists M \in TM \; [\forall v \in TP [\exists V \; [[v \in V] \text{ and } [(M,V) \in LVS]]]]]$$

Proof by demonstration:

$I = \{x\}$, $S = \{S_0\}$

	S,I	N	O	D
M:	S_0, x	S_0	x	+1

trivially seen from state table:

$$[\forall \text{ time } t [\forall 8 [\forall P [\text{not } M \text{ halts}]]]]$$

and $[\forall n \in \mathbb{N} [\forall v \in I^n$

$$[[v \overset{M}{\Rightarrow} \{(X)\}] \text{ and } [(M,\{(X),v\}) \in LVS]]]]$$

hence $[\forall v \in TP [(M,\{v,(X)\}) \in VS]]$

and by Theorem 1, $[\exists V [[v \in V] \text{ and } [(M,V) \in LVS]]]$

Q.E.D.

8. 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 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 [2], that any program that replicates itself is a virus [Lemma 2.1], and thus that $[(M,V) \in VS]$ is undecidable.

Theorem 6:

$$[\nexists D \in TM [\exists s_1 \in S_D, \; [\forall M \in TM [\forall V \in TS \; 1) [D \text{ halts}] \text{ and } 2) [S_D(t) = s_1] \text{ iff } [(M,V) \in VS]]]]]$$

"Proof by reduction from the Halting Problem:

$$
\begin{array}{l}
[\forall M \in TM[\exists M' \in TM \\
\quad [\text{"L"} \notin I_{M'}] \text{ and } [\text{"R"} \notin I_{M'}] \text{ and} \\
\quad [\text{"l"} \notin I_{M'}] \text{ and } [\text{"r"} \notin I_{M'}] \text{ and} \\
\quad [\forall S_{M'}[I_{M'} = \text{"r"}] \Rightarrow \\
\qquad [[N_{M'} = S_{M'}] \text{ and } [O_{M'} = \text{"r"}] \\
\qquad\qquad \text{and } [D_{M'} = +1]]] \\
\quad \text{and } [\forall S_M \\
\qquad [[N_M = S_M] \text{ and } [O_M = I_M] \text{ and } [D_M = 0]] \\
\quad \Rightarrow [[N_{M'} = S_x] \text{ and } [O_{M'} = I_M] \text{ and } [D_{M'} = 0]]] \\
]]
\end{array}
$$

We must take some care in defining the machine M′ to ensure 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 ensure 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′ so that for all states such that "M halts", M′ goes to state S_x.

now by [2]

$$
\begin{array}{l}
[\nexists D \in TM \\
\quad [\forall M' \in TM[\forall V' \in TS \\
\quad 1)\ [D \text{ halts}] \text{ and} \\
\quad 2)\ [S_D(t) = s_1] \text{ iff } [(M',V') \text{ halts}]]]]
\end{array}
$$

We now construct (M,V) s.t.

$[(M,V) \in VS]$ iff $[(M',V')$ Halts$]$

as follows:

	S,I	N	O	D	
M:	S_0,L	S_1	L	0	; if "L" then continue
	S_0,else	S_0	x	0	; else halt
	S_1	CPY("l","r","R")			; Copy from 1 till r after R
	S_2	L("L")			; left till "L"
	S_3	R("R")			; right till "R"
	S_4	S_5	l	+1	; move to start of (M′,V′)
	S_5	M′			; the program M′ goes here
	S_x	L("L")			; move left till "L"
	S_{x+1}	CPY("L","R","R")			; Copy from L till R after R

$V = \{(L,l,V',r,R)\}$

Since the machine M requires the symbol "L" to be under the tape head in state S_0 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.

$$
\begin{array}{l}
\forall t \in \mathbb{N}[\forall S_M \leqslant s_x \\
\quad [\nexists P_M(t)[[I \neq \text{"L"}] \text{ and } [O = \text{"L"}]]]]]
\end{array}
$$

This restricts the ability to generate members of VS such that V only produces symbols outside itself containing the symbol "L" in state S_0 and S_{x-1}, and thus these are the ONLY states in which replication can take place. Since S_0 can only write "L" if it is already present, it cannot be used to write a virus that was not previously present.

$$
\begin{array}{l}
[\forall t \in \mathbb{N}[\forall S(S_5 \leqslant S \leqslant S_x) \\
[\text{not}[M' \text{ halts at time } t]] \\
\qquad \text{and } [P_M(t+1) \text{ not within } V]]]
\end{array}
$$

If the execution of M' on V' never halts, then S_{x+1} is never reached, and thus (M,V) cannot be a virus.

$[\forall Z \in TP s.t. Z_0 \neq \text{“L”}]$
$\quad [M \text{ run on } Z \text{ at time } t]$
$\quad\quad \Rightarrow [M \text{ halts at time } t+1]$
$[(M',V') \text{ Halts}] \text{ iff}$
$\quad [\exists t \in \mathbb{N} s.t. \mathcal{S}_t = s_{x+1}]$
thus $[\text{not}(M',V') \text{ Halts}] \Rightarrow [(M,V) \notin VS)]$

Since S_{x+1} replicates v after the final “R” in v, M' halts implies that V is a viral set with respect to M

$[\exists t \in \mathbb{N} s.t. \mathcal{S}_t = s_{x+1}] \Rightarrow$
$\quad [\forall v \in V s.t. [v \overset{M}{\Rightarrow} \{V\}]]$
and from Lemma 2.1
$\quad [\forall v \in V v \overset{M}{\Rightarrow} V] \Rightarrow [(M,V) \in VS]$
thus $[(M,V) \in VS] \text{ iff } [(M',V') \text{ Halts}]$
and by [2]
$[\nexists D \in TM$
$\quad [\forall M' \in TM [\forall V' \in TS$
$\quad 1) [D \text{ halts}] \text{ and}$
$\quad 2) [S_D(t) = s_1] \text{ iff } [(M',V') \text{ halts}]]]]$

thus
$[\nexists D \in TM$
$\quad [\forall M \in TM [\forall V \in TS$
$\quad 1) [D \text{ halts}] \text{ and}$
$\quad 2) [S_D(t) = s_1] \text{ iff } [(M,V) \in VS]]]]$
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 $[(M,V) \in VS]$, 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 virus, for example v with a slightly different sequence V'' instead of V'.

Lemma 6.1:
$[\nexists D \in TM$
$\quad [\forall (M,V) \in VS$
$\quad\quad [\forall v \in V [\forall v'$
$\quad\quad\quad 1) [D \text{ halts}] \text{ and}$
$\quad\quad\quad 2) [S(t) = S_1] \text{ iff } [v \overset{M}{\Rightarrow} \{v'\}]]]]]$
sketch of proof by demonstration:
modify machine M above s.t.:

M:				
S_0,L	$S_{0'}$	L	0	; if “L” then continue
S_0,else	S_0	x	0	; else halt
$S_{0'}$	CPY(“L”,“R”,“R”)			; replicate initial virus
$S_{0''}$	L(“L”)			; return to replicated “L”
S_1	CPY (“l”,“r”,“R”)			; Copy from l till r after R
S_2	L(“L”)			; left till “L”
S_3	R(“r”)			; right till “R”
S_4	S_5	r	−1	; move to start of (M',V')
S_5	M'			; the program M' goes here
S_x	L(“L”)			; move left till “L”
S_{x+1}	R(“R”)			; move right till “R”
S_{x+2}	S_{9-k}	“R”	+1	; get into available space
S_{x+3}	generate(v')			; and generate v'

assume $[v' \text{ is a virus w.r.t. } M]$
since $[S_{x+3} \text{ is reached}] \text{ iff } [(M',V') \text{ halts}]$
thus $[v' \text{ is generated}] \text{ iff } [(M',V') \text{ halts}]$
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) successive iterations of the "Universal Computing Machine" [2]. 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 [2]:

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

Using the original description of the "Universal Computing Machine" [2], we modify the UTM so that each successive iteration of the UTM interpretation of a "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.

S×I	N	O	D	
b:	f(b₁,b₁,"::")			; initial states of UTM print out
b_1:	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 the machine loops back 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⟩

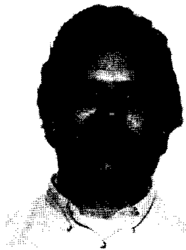
The modification of the "anf" state breaks the normal interpretation loop of the UTM, and replaces it with a replication into which we then position the tape head so that upon return to "anf" the machine will operate as before over a different portion of the tape. The second modification ensures that 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:

S×I	N	O	D	
S_0,L	S_1	L	+1	; start with "L"
S_0,else	S_0	else	0	; or halt
S_1...				; states from modified UTM



Fred Cohen received a B.S. degree in electrical engineering from Carnegie-Mellon University in 1977, an M.S. degree in information science from the University of Pittsburgh in 1981, and a Ph.D. degree in electrical engineering from the University of Southern California in 1986. He was a professor of Computer Science and Electrical Engineering at Lehigh University from January 1985 through April 1987, a professor of Electrical and Computer Engineering at The University of Cincinnati from September 1987 through to December 1988, and is currently Director of The Radon Project in Pittsburgh. He is a member of the ACM, IEEE, the ASEE, and the IACR and a member of the international board of reviewers of the IFIP/TC11 journal, *Computers & Security.*

Dr. Cohen has published over 20 professional articles, has recently completed a graduate text titled *Introductory Information Protection,* and has designed and implemented numerous devices and systems. He is most well known for his groundbreaking work in computer viruses, where he did the first in-depth mathematical analysis, performed many startling experiments which have since been widely confirmed, and developed the first protection mechanisms, many of which are now in widespread use. His current research interests are concentrated in high integrity systems.

References

[1] F. Cohen, Computer viruses — theory and experiments, *7th Security Conf., DOD/NBS, September 1984.*

[2] A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.,* 42(2) (1936) 230–265.