Viral Computing Environments and Some Benevolent Computer Viruses

by Dr. Frederick B. Cohen ‡

Abstract

In this paper, we discuss the operation of viral computing environments ($\mathcal{VCE}$s), designed to provide an environment in which practical applications of computer viruses can be safely explored. We begin with some history and basic definitions, discuss $\mathcal{VCE}$s in general and the $\mathcal{VCE}$s used to perform our experiments, describe practical virus implementations currently in commercial and experimental use, consider the future of this technology and its limitations, summarize results, draw conclusions, and propose further work.

Search terms: Computer Viruses, Artificial Life

‡ Funded by ASP, PO Box 81270, Pittsburgh, PA 15217, USA

1 Background

An informal definition of 'Computer Viruses' was first published in 1984 [1], and was soon followed by a formal definition first published in 1985 [2] based on Turing's model of computation [3]. An alternative formal definition was proposed in 1989 [4] based on set theory, but this definition never gained widespread acceptance or use. In each of these cases, detection of viruses was shown to be undecidable, and numerous other results were derived.

We begin here with an informal discussion by presenting a pseudo-code example of a very simple virus:

V := [COPY V TO RANDOM-FILE-NAME;]

This virus simply replicates into files with random names, and is unlikely to be successful in current computing environments because the chances are very poor that any of the replicas will ever be run, and even if they were run, they would eliminate the functions previously provided by their hosts, and would probably be detected rapidly. Since the replicas are of no practical value, they would likely be hunted down and destroyed. More purposeful viruses, both malicious and benevolent, have been shown quite powerful, primarily due to their prodigious reliability and ability to spread. Viruses have also caused quite a problem in some environments.

With the ability to replicate comes a host of other issues. The most obvious issue is that a replicating program might exhaust all of the available resources in a system, thus causing a system failure. As an example, the above virus 'V' has a very high probability of consuming extra file space each time it is executed, and in most systems, assuming replicas are run, it will eventually consume all of the disk space.

Another important aspect of viruses is their ability to 'carry' additional information with them. For example, in 1984 [1] a pseudo-code example was provided for a compression virus. This virus carried a compression and decompression algorithm with it, compressed executable files during infection, and automatically decompressed those files upon execution, in order to implement a time/space tradeoff. Another common example of a practical virus is a backup program that copies itself onto the backup media during backup and restores itself during restoration. In this case, the virus carries copies of all of the backed-up files with it to provide enhanced reliability. Malicious viruses typically include some damaging code which is triggered by some system condition. For example, some viruses overwrite critical disk areas on a particular date, while some others cause errors in systems with particular configurations.

Another interesting feature of self-replicating programs is their resilience. In environ-

ments where non-replicating programs often fail or get destroyed through errors or omissions, viruses seem to thrive. This is because of the natural redundancy provided by replication. In an environment of 'survival-of-the-fittest', viruses seem to be more fit than non-viral programs. By example, the Internet virus was sufficiently resiliant to resist persistant and widespread attempts be systems administrators to remove them, while some people have 'battled' viruses in their environment for a year or more [5].

From a protection standpoint, viruses offer unique problems. Their resilience makes then very hard to remove from an operating environment, while their transitive spread bypasses most modern protection methods, which tend to wrongly assume non-transitive information flow. General purpose detection is undecidable [1,2,4], while special purpose methods are not cost effective [9]. Their ability to evolve makes detection for removal quite difficult even for relatively simple and well known evolutionary schemes, while the general problem of tracking down evolutions has been shown undecidable [1,2] and even relatively straightforward evolutionary schemes present severe detection problems. [5]

There are many other interesting properties of self-replicating programs, including their ability to improve performance relative to a metric through random variation and selective survival, their ability to form complex phenotypes from simple genotypes, and the unpredictability of phenotypes based on genotypes. For a reasonably good summary, we refer the interested reader to some of the recent literature in this area [6].

Because of the impact of malicious computer viruses on the current computing environment, some less knowledgeable researchers have mistakenly proclaimed that viruses are universally malicious, but if we examine the facts, we come to a very different conclusion. In a sense, that is the purpose of this paper; to present some results on benevolent computer viruses and how they have been successfully used in real-world applications.

We begin with the most commonly used formal definition, describe some of the important results attained in previous works, briefly consider why alternative definitions have not taken hold, describe some of the useful features of this definition, and overview the manner in which this definition fits into previous notions of life. Next we consider 'viral computing environments' ($\mathcal{VCE}$s), describe the rational for their use, and detail the $\mathcal{VCE}$s used for our experiments. We then discuss several practical viral computing systems in which computer viruses safely coexist with non-viral computer programs, useful computation is performed by the computer viruses, and the use of viruses provides some advantages over alternative methods. Some of the results show the limitations imposed by the use of computer viruses for computation, and we describe what we consider to be the major tradeoffs in viral computation for current common applications. We extend these results by describing a path for future work in which larger scale $\mathcal{VCE}$s could be implemented, and considering some of the sorts of computing environments that may exist in the future.

2 Some Formalities

Cohen [2] presents 'viral sets' in terms of a set of 'histories' with respect to a given machine. A viral set is a set of symbol sequences which, when interpreted, causes one or more elements of the viral set to be written elsewhere on the machine in all of the histories following the interpretation. We include here some of the relevant definitions required for the remainder of the paper, starting with the definition of a set of Turing-like [3] computing machines '$\mathcal{M}$' as:

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

where:

$\mathcal{N} = \{0 \ldots \infty\}$	(the 'natural' numbers)
$\mathcal{I} = \{1 \ldots \infty\}$	(the positive 'integers')
$S_M = \{s_0, ..., s_n\}, n \in \mathcal{I}$	($\mathcal{M}$ states)
$I_M = \{i_0, ..., i_j\}, j \in \mathcal{I}$	($\mathcal{M}$ tape symbols)
$d = \{-1, 0, +1\}$	($\mathcal{M}$ head motions)
$\$_M : \mathcal{N} \rightarrow S_M$	($\mathcal{M}$ state over time)
$\square_M : \mathcal{N} \times \mathcal{N} \rightarrow I_M$	($\mathcal{M}$ tape contents over time)
$P_M : \mathcal{N} \rightarrow \mathcal{N}$	(current $\mathcal{M}$ cell at each time)

The 'history' of the machine H_M is given by $(\$, \square, P)$,[1] the 'initial state' is described by $(\$_0, \square_{0,i}, P_0), i \in \mathcal{N}$, and the set of possible $\mathcal{M}$ tape sub-sequences is designated by I^*. We say that; M is halted at time $t \Leftrightarrow \forall t' > t, H_t = H_{t'}, (t, t' \in \mathcal{N})$; that M halts $\Leftrightarrow \exists t \in \mathcal{N}, M$ is halted at time t; that p 'runs' at time $t \Leftrightarrow$ the 'initial state' occurs when P_0 is such that p appears at $\square_{0,P_0}$; and that p runs $\Leftrightarrow \exists t \in \mathcal{N}, p$ runs at time t. The formal definition of the viral set ($\mathcal{V}$) is then given by:

$$\forall M \forall V (M, V) \in \mathcal{V} \Leftrightarrow$$
$$[V \subset I^*] and [M \in \mathcal{M}] \; and \; \forall v \in V \forall H \forall t, j \in \mathcal{N}$$
$$[[P_t = j] \; and \; [S_t = S_0] \; and \; (\square_{t,j}, ..., \square_{t,j+|v|-1}) = v] \Rightarrow$$
$$\exists v' \in V, \exists t', t'', j' \in \mathcal{N} \; and \; t' > t$$
$$1)[[(j' + \mid v' \mid) \leq j] or [(j + \mid v \mid) \leq j']] \; and$$
$$2)[(\square_{t',j'}, ..., \square_{t',j'+|v'|-1}) = v'] \; and$$
$$3)[\exists t''[t < t'' < t'] \; and \; [P_{t''} \in j', ..., j' + \mid v' \mid -1]]$$

[1] For convenience, we drop the M subscript when we are dealing with a single machine except at the first definition of each term.

Several important results about viral sets were also reported [2]:

$|\mathcal{V}|$ is uncountable for some M.
Every sequence of symbols is a virus on some M.
Virus detection is undecidable.
Any program that always copies itself is a virus.
Detecting evolutions of a known virus is undecidable.
Virus evolution is as general as $\mathcal{M}$ computation.

Many authors publish far different 'definitions' of a virus, but none of these have gained widespread acceptance. There are probably three major reasons for this:

- The mathematical results derived under this definition have far reaching implications, and no other definition has been used to derive any substantial results. Most authors use the results derived under this definition to make claims about viruses, even if their linguistic definitions don't resemble the ones used to derive these results.
- Most other published definitions aren't scientific in that there can be no definitive test against the definition or are palpably inconsistent. One example is the use of the word 'malicious' in a definition of a virus, which is a matter of opinion and not fact. One well known virus defense author tried defining 'real viruses' as viruses (under our definition) that the user doesn't know about. The definition was widely rejected when we pointed out that a 'real virus' for you may not be a 'real virus' for me! Another author tried to define a *virus* as an 'unauthorized' virus. Whenever you run any program in modern systems, you implicitly authorize it to do whatever it does, which is to say, if a program has a virus, the virus is authorized when you run the program, and is therefore not a *virus*. If you claim you didn't authorize the virus because you didn't know about it, you return the the 'real virus' problem.
- Some rather well known developers of anti-virus products and members of the 'computer security' industry have come up with definitions designed to illicit fear responses, but these definitions invariably result in major inconsistencies or unacceptable decision procedures. The use of fear as a marketing tool may be effective, but it does not seem to lead to useful scientific results.

This definitional problem has led to some very bizzarre claims. One author (citation intentionally not included) claims that since a 'committee of experts' at an 'Artificial Life' conference could find no beneficial application of computer viruses, there could be none. We conclude that the attendees of this particular conference were not sufficiently expert to understand the implications the most commonly used definition of the term they were discussing, or were simply unaware of the definition itself. We also conclude that the author,

in using a finite number of confirming instances to draw a universal conclusion about an infinite set, was unaware of Popper's work in this regard [11]. [2]

From a historical perspective, it turns out that the mathematical definition of computer viruses described herein was originally proposed as a possible definition for life [2]. The present definition considers anything to be alive in a particular environment if it 'replicates'. One of the unique features of this definition is that replicas need not be exact. In fact, we only require that some replica be capable of replication, which forms a basis for considering evolution. This general purpose evolution is fundamental to utility as computing mechanisms. Furthermore, the environment is as vital to the definition of life as is the organism under consideration, for surely, people are not alive in the middle of the sun. Under the present definition, any combination of 'living' organisms and their environments are also considered 'alive'.

This definition also seems to meet the 'metabolism' criterion which requires living creatures to use energy and produce waste, for certainly it requires energy to replicate, and any use of energy in our world results in waste. Similarly, the 'entropy' based definition that views life as locally decreasing entropy is fulfilled by this definition because it creates 'order' out of 'disorder' in the region in which it replicates by replacing any previous symbol sequences with symbol sequences from the viral set (which in all of the interesting cases is a subset of the totality of symbol sequences). It is also consistent with a Mule being alive, in that a Mule is composed of living cells, even though the organism as a whole may not be able to procreate.

This combination of factors makes the formal definition of computer viruses a serious contender for a general purpose definition of life in the information theoretic sense. Only it's dependence on the Turing model of computation is limiting in terms of its viability for more general application, and it appears that this limitation is not as severe as it might be considered at first glance [2]. A possible extension would be the use of a statistical model of computation and a resulting statistical model of the likelihood of replication.

3 Viral Computing Environments

One of the vital features of the formal definition is the linkage between the environment and the viral set. In a fixed environment, we have a very static situation that is relatively easily analyzed, but in general, a viral set can have a serious impact on its environment.

[2] It is this claim that motivated me to write this paper for this conference.

3.1 Some Philosophy Mixed with Reality

In some cases, viruses cause their own death through their impact on their environments. For example, many malicious real-world viruses are designed to destroy their operating environment, which ultimately leads to their own 'death'. The interaction of viruses in an environment may also cause the environment to change in ways that are not immediately apparent. A simple example is the viral set used to show that 'vaccination' is in general impossible [5], in which the vaccination against one virus provides 'food' for its competitor, which eats the vaccines, thus providing more food for the virus vaccinated against. This simple feedback system is an example of a larger class of self-stabilizing viral sets that assure particaular mixes of computation even in the presence of multiple errors [12].

In the immediate context, when we speak of an environment for viral computation, we are considering the design of a specific ecosystem for specific sorts of viruses, but in the more general sense, $\mathcal{VCE}$s are the environments formed by the evolution of the entire living system. Even though our $\mathcal{VCE}$s are somewhat limited by design, they represent only the very beginning of the inquiry in this area, and should be considered in that more general context.

As an example, we could potentially have human beings living in the middle of the sun, provided they brought along a suitable ecosystem for their continued survival. The distinction between the living entity and its environment are continually blurred, and hierarchical living systems may form a multitude of ecosystems within other ecosystems. If the people living in the middle of the sun replicated their ecosystem and themselves to form a second colony, the entire ecosystem could be considered alive, while within it the human beings would independently be alive, just as within them, their cells are alive. Just as our genes create us for their survival [13], we create environments for our survival, and eventually, they may create their own environments for their own survival.

In the same manner, $\mathcal{VCE}$s may one day be created by computer viruses for their own survival, and these $\mathcal{VCE}$s may in turn create their own environments, eventually even replicating and transforming the very media in which they exist. Consider the implications of mapping the human genome to the point where we can effectively simulate its operation in a computer. In an information theoretic sense, our genes would then have replicated into a completely different media. But we have already replicated informationally into the computer. When the first human designed self-replicating program was implemented, that replica was an evolved form of the human being, at least in the sense that we speak of evolution in the formal definition of computer viruses. Perhaps this will someday be used by psychologists to justify the profile motive for virus writers as related to supressed sexual urges, but that is too far afield for this work.

3.2 Some Findings on $\mathcal{VCE}$s

In exploring the potential for practical viral computation, we eventually came upon the idea that most of the current computing environments are poorly suited for this purpose.

The lack of adequate large grained mandatory access control [5] in combination with unlimited sharing [1,2] makes viral computation using standard computing environments somewhat risky. For example, the vast majority of current malicious computer virus problems would be largely eliminated if the computing community had selected the protection models of Denning [14] over those of Harrison et. al. [8] and the designers of DOS had decided to provide some sort of protection in their design. Since the most current environments do not provide adequate protection, we selected a different approach of providing special system facilities which both facilitate viral computation and make the interpretation of experimental viruses in other environments infeasible.

The compiled code environment is also an impediment to efficient replication and evolution. The first problem is that the translation from the 'high-level' concepts of the programmer into the 'low-level' details of the environment forces the virus designer to either operate at the low-level or repetedly translate from high-level to low-level in order to implement the programming concepts. Because evolutionary viruses are constantly rewriting themselves as part of the process of replication, high-level viruses would have to repeatedly use compilers and loaders, and this would consume a great deal of time and space, as well as introduce a lot of unneeded complexity. Finally, the enforced separation of fixed 'program' from variable 'data' used in most compiled code systems requires that a compiled code virus maintain a redundant copy of its source code to be used for evolution. In order to make viral programming simple and efficient, we decided to use a fairly fast interpreted language in which data and program are represented in the same manner. This leaves very few choices.

Another important issue is communication. In systems requiring a great deal of interaction between data (e.g. cells in a spread sheet) viruses create performance problems because viral programs tend to carry their data with them and have to be invoked in order to reveal their data. For this reason, designing an efficient viral application requires selecting an appropriate partitioning of the problem space and is most effective in applications where a multitude of relatively independent processes are performed. This is not surprising because in order to get the advantages of reliable distributed computation, viruses have to act as independently as possible. Otherwise, the failure of a central database would cease up the system, and bottlenecks would be introduced by attempts at simultaneous access.

In the biological world, this communication issue shows itself in terms of physical proximity. Cells communicate with physically neighboring cells through biochemical and electromagnetic means. Each cell in turns communicates with its neighbors in a relatively uniform

fashion. In the current computing world, communications distances work quite differently. Communication between viruses in a given processor can be very rapid, while interprocessor communications is usually far slower. In global networks, there are often delays of several seconds between far distant processes. This difference may have many impacts on networked viral computation, but for the purposes of this paper, this distinction is ignored.

3.3 Some Real World $\mathcal{VCE}$s

We decided on two $\mathcal{VCE}$s; one for **PC**s running the DOS operating system, and one for systems running the Unix^{tm} operating system.

In the DOS environment, we selected **Lisp** as the language of choice because of its interpretive and incrementally compiled operation, the availability of high speed inexpensive **Lisp** systems for the **PC**, the equivalence of data and program structures, and the great ease of modifying programs with programs. The particular **Lisp** implementation we selected [15] has many of the features of 'Common Lisp', but also the ability to perform system calls where appropriate for a practical application, good performance, incremental compilation, and the small disk and memory footprint required for DOS operation.

In the Unix^{tm} environment, portability is a vital concern, since unlike the DOS environment, compiled code on Unix^{tm} systems is not universally portable. The high cost of widely supported **Lisp** systems for Unix^{tm} along with the free and universally available presence of the 'sh' (shell) command interpreter led us to use the shell for the Unix^{tm} implementation. It turns out that the Unix^{tm} shell has a common format for data and program and that it is interpreted so we don't have problems related to compiled code, but the performance of shell scripts leaves a great deal to be desired. Fortunately, the Unix^{tm} environment also has a wide variety of reasonably fast standard programs that facilitate many of the nontrivial operations such as sorting, searching, and formatting. Formatted input is handled through special purpose compiled programs provided in 'C' source form.

Both $\mathcal{VCE}$s consist of a set of computing 'engines', quickly accessible 'collections' of 'viruses', special 'system calls', menu driven user interfaces, and various application support systems. From a standpoint of viral computation, the only components of interest are the 'engines', 'collections', and 'viruses'.

Each engine (E) is a finite state automata with essentially unbounded memory ($E \in \mathcal{M}$) that takes one virus ($c \in C$) at a time from the collection, and interprets c with a method appropriate to the application (E runs c). Each $c \in C$ replicates and/or evolves, placing the replicas back into the collection ($\Rightarrow c' \in C$). This maps directly into the definition of $\mathcal{V}$ when we map E into M, c into v, and C into V. Thus we have $(E, C) \in \mathcal{V}$

Most of the engines we work with are called by 'fair' schedulers that 'run' all viruses c ($c \in C_E$) and cause E to interpret c. Some engines are run at regular intervals depending on the details of the application, while some engines are called to operate on a particular virus because of some external event.

The engines facilitate safe viral computation by providing special calls that are not otherwise available in the environment, and which make writing viruses particularly simple. In the DOS $\mathcal{VCE}$, we provide a **Lisp** function 'replicate-and-evolve', which rewrites the virus replacing all previous data values with their new values. This facilitates simple run-time evolution. In the Unix^{tm} $\mathcal{VCE}$, we simply replicate by copying the current program using the Unix^{tm} 'cp' program, but restrict the location of copies through Unix^{tm} access controls.

There is a very close analogy to both 'event-driven' simulators and 'production systems'. In production systems, the computing engine repetedly scans a database testing each of a series of triggerring conditions which, if true, result in the interpretation of associated instructions. In event-driven simulators, a scheduling system selects a set of objects affected by each scheduled 'event' for interpertation, and schedules new events as a result of interpretation. The production system is similar because it scans a database for applicable operations and performs them, but it is also substantially different because it works from a central computing system with a global database, whereas with $\mathcal{VCE}$s, each virus carries all of the information required for its interpretation along with it. Event driven simulation is similar in that it schedules all applicable events and causes the appropriate objects to be invoked, but it is very different in that a major objective of a simulation engine is to provide effective communication between the objects, whereas a typical $\mathcal{VCE}$ provides little or no communication between viruses.

4 Some Practical Applications

We have implemented two substantial commercial systems and several non-commercial experimental systems using the $\mathcal{VCE}$s described above. First, we will concentrate on two commercial applications; a bill collection system, and a systems maintenance facility; and then we will discuss an experimental distributed database system. The collection system has been implemented in both DOS and Unix^{tm} $\mathcal{VCE}$s, and is designed to automate much of the bill collection process for businesses and collection agencies. The maintenance system has only been implemented under Unix^{tm}, primarily because Unix^{tm} is a timesharing environment in which substantial amounts of maintenance are required and can be performed in the background. The distributed database system has only operated under Unix^{tm} to this point, but a DOS implemention is quite straight forward with modern networking capabilities. The

Unix^{tm} application began operating in 1986 and the DOS system was developed for purely commercial application in 1991.

4.1 The Viral Bill Collector

The collection systems exemplify one extreme in viral computation. They use special purpose engines that embody state transformation information appropriate to the collection process, and the viruses carry only the state information used by the transformation process. The engines provide a limited function $\mathcal{VCE}$, in that they do not, in and of themselves, provide viruses with the ability to perform general purpose computations, however, the engines can be customized for different collection processes, and the engines themselves are not limited in their function. Each viral computation consists of an engine interpreting a virus to initialize and transform state information into a new replica.

The computing environment 'births' a new 'bill collector' every time a new case is entered by a user, kills 'bill collectors' whenever the user indicates that a bill is fully paid, and allows the bill collector to write letters and evolve through its life-cycle. In its simplest form, each bill collector is just a small program that collects a single bill by sending a series of letters over time depending on real-world events or the lack thereof. The state information associated with each case is stored as part of its bill collector.

It turns out that writing a computer program to collect a single bill is not very hard to do. In fact, in a few hours, an average programmer can write a simple bill collecting program that will do a pretty good job of collecting a single bill, based on the methods of collection done by an expert human bill collector. There are usually several different scenarios depending on how the client and debtor react to the methods employed, the amount owed, and other circumstances, but otherwise the process is simple.

If we were writing a high volume collection system in the standard fashion, we would then implement a complex database management system and alter the small bill collection program to interface with it. Next, we would devise a technique for scanning through this database periodically to determine which collection cases required action at any given time, and based on this list, have the bill collection subroutine perform collections on all applicable cases. To collect statistics and resolve the status of cases, we would then have to implement yet another database scanning system, and the list goes on and on. By the time we were done, we would have a very large and complex system, and the original bill collection program would only be a minor cog in the machinery.

With the viral programming approach, we take another tactic altogether. Instead of creating a large centralized bureaucracy which controls and directs all activities, we distribute

all functions and data to the individual bill collectors. Each bill collector only has information related to its own collection case and the ability to selectively call upon its various scenarios for bill collection. If some outside activity like a payment or a response to a previous action takes place, the human operators 'wake up' the appropriate bill collector by sending it the new information. The collector then reacts to the situation by 'evolving' its state of mind and line of pursuit to meet the new situation, reschedules its pending wake up calls, and goes back to sleep.

One major advantage of writing a simple virus for this task is that all we have to do is provide basic replication, evolution, and wake up call mechanisms, and we don't have to deal with the complexity of large databases or long database searches to determine when to do what. Another major advantage is that since we are running many very small and independent programs instead of one large program, we can more easily distribute the computing load over a multitude of machines, and we don't have to deal with issues like simultaneous access, file locking, and process blocking. There are also disadvantages, in that collecting data from all of the bill collectors is somewhat less efficient on a single computer than looking up all of the data in a common database, and making changes to all of the independent bill collectors requires a systemic evolution process.

To collect global data with a viral bill collector, we typically awaken every bill collector, ask for the required information, and collect the results. We typically end up writing several such data collection applications which we then schedule for operation in off hours. The results from the previous day are then always available the next morning, and since we are usually using otherwise idle computer time during off-peak hours, the inefficiency is not unduly bothersome. In an emergency, we can awaken all of the bill collectors during the day to get more instantaneous results, but in this case system wide performance suffers greatly. This feverish sort of activity has only been required in the rarest of circumstances.

To evolve all of the independent bill collectors in response to new information requirements, we again awaken each bill collector, provide the necessary information for it to change its programatic (as opposed to genetic) codes, and allow it to return to sleep. Just as in the global data collection case, we can perform this sort of change during off hours, and there is rarely a case where we need such a change on a moment's notice. We needn't use this sort of systemic evolution all of the time. Instead, we can take the less invasive approach of developing better and better bill collectors over time, and simply birthing them for collecting new bills. Eventually older bill collectors die out as they conclude their tasks.

Our first viral bill collection system was implemented by one programmer in one week, in 1986. It has evolved successfully in response to new needs without systemic changes since that time, and although this sort of evolution currently requires human design, the amount of work is minimal. The bill collection viruses also coexist in the environment with a set of 'maintenance' viruses that periodically awaken to perform cleanup tasks associated with

systems maintenance.

4.2 The Maintenance Viruses

The maintenance viruses represent the opposite extreme in viral computation in that they use a general purpose engine (the Unix^{tm} shell along with the rest of the programming environment) and carry only state transformation information with them. They examine the state of the computer system and perform transformations on that state to affect their purpose. Each viral computation consists of the common engine interpreting a virus which is programmed to cause one or more replicas to be created.

Maintenance viruses, as a class, seem to be one of the most useful forms of computer viruses in existence today. Put in the simplest terms, computer systems are imperfect, and these imperfections often leave residual side effects, such as undeleted temporary files, programs that never stop processing, and incorrectly set protection bits. As more and more of these things happen over time, systems become less and less usable, until finally, a human being has to repair the problems in order to continue processing.

In the case of the viral bill collector, the design of the system is such that temporary files are stored under identifiable names, processing for each virus tends to be relatively short, and protection bits are consistently set to known values. To reduce manual systems administration, we decided to implement viruses that replicate themselves in limited numbers, seek out known imperfections, and repair them. Over time, we reduced systems administration to the point where the viral bill collector operated for over two years without any systems administration other than adding and removing users. The maintenance viruses were so successful that they even removed side effects of failed maintenance viruses.

To assure the continued survival of the maintenance viruses, they are born with a particular probability every time a user awakens a bill collector, and to assure they don't dominate processing by unbounded growth, they have limited life spans and replicate with lower probability in each successive generation. With proper probabilities, this combination of factors successfully produces stable populations of maintenance viruses and is quite resilient.

These "birth/death" processes are central to the problem of designing viruses that don't run amok, as well as to the evolution of viral systems over time. If it weren't for the death of old bill collectors and maintenance viruses, the system would eternally be collecting bills and performing maintenance under old designs, and the number of bill collectors and maintenance viruses would grow without bound. A global modification of all of the existing bill collectors would be required to make a system change, and this might be very hard to accomplish in a complex network. Birth and death processes are also vital to optimization in viral systems

where the environment changes dramatically with time, since what is optimal today may not even survive tomorrow.

This may even explain why biological organisms that live in relatively dynamic environments (e.g. people on land) have limited life spans, while organisms that live in static environments (e.g. coral in an ocean) can live indefinitely without noticeable aging. There must be enough births to survive a reasonable number of deaths, and death is necessary to prevent a population from expanding to a size where it consumes too many resources and dies from starvation. In a dramatically changing environment, the threats to survival change with time, and without active evolution, stagnant strains may eventually meet unsurvivable threats. The more dramatic the rate of environmental change, the more dramatic the evolution must be in order to survive, but we must be careful to understand that what seems to be dramatic change to one species may go completely unnoticed to another. For example, a 10 degree change in average local temperature has almost no immediate impact on people, and people regularly go through these changes when moving from city to city, but the same change in temperature dramatically impacts the plants that can survive, and many plants simply die out when moved to different climates.

4.3 The Viral Distributed Database

We have heard several reports of others looking into the use of viruses for implementing distributed databases, and every once in a while, we even hear about a project that is or once was operational. We generally believe these reports, and have had similar ideas and performed similar experiments. This portion of our study is then dedicated to the many other researchers who have pursued similar lines and not reported them.

Our viral distributed database is designed to operate on any computer network which allows files to be sent to remote processors. This design choice was made for several reasons. First and foremost, even the simplest PC networks provide this sort of capability, so the database can be implemented at almost no cost for experimentalists (i.e. you can do a simple 10 PC example with a $25 software package and about $100 worth of connecting cables - assuming you have 10 PCs). Second of all, this capability existed and/or exists on the ARPA-net, the Internet, DECnet, NFS, MCI-mail, and almost every other computer network around; which means people on networks all around the world can apply the results very easily. Third of all, merely sending a file to a remote processor does not normally cause it to be interpreted, and thus we limit the viruses to processors where users explicitly choose to run them, or more to the point, processors running a virus interpreter.

For the purposes of our discussion, we will assume that our database consists of a set of sequential files spread out over any number of networked computers, and that the major

operation of interest is 'find-record', where the record being sought might be on any number of different connected processors. Performance, reliability, and ease of programming are the three major factors under consideration in this example.

*****

In this case, we have reformated listings for space compression by combining statements on lines even though syntax errors will result from actual use.

This is the interpreter that runs on all processors in the array.

```
a=‘pwd‘
while test -f running do
 for i in *.sh do
  if test -f $i.done then zz=0
  else
    if test -f $i then
      sh $i; mv $i $i.done
      echo "$a:$i done ‘date‘" >> logfile
  fi fi
  done; sleep 5
 done
```

The following is a searching virus that replicates from processor to processor, spreading the search as it goes. It is a virus because each time it runs, it replicates into the next higher processor number, the next lower processor number, or both, depending on where the endpoints of the processor array are.

```
a=‘pwd | sed ’s:/v/vdb/::g’‘
b=‘expr \( $a + 9 \) % 10‘;
c=‘expr \( $a + 1 \) % 10‘
grep @@@ *.db >> ../results
cp $0 ../$b;cp $0 ../$c
```

In this listing, we show a search number followed by a 'processor' number, an indicator showing that either the partial search was completed ('-'), the processor was halted ('H'), or the processor was continued ('C'), and the time of completion of the operation (in minutes past the hour). Time increases across each line (i.e. read this table as you would a book for increasing times).

1:3	-	04:38	*:5	H	04:57	*:0	H	05:01	1:2	-	05:06
2:7	-	05:06	1:4	-	05:09	*:0	C	05:23	1:1	-	05:27
2:8	-	05:27	2:6	-	05:31	3:9	-	05:36	4:4	-	05:39
*:5	C	05:44	1:0	-	05:59	4:3	-	06:02	3:8	-	06:06
5:6	-	06:12	*:8	H	06:12	2:9	-	06:18	1:5	-	06:31
3:7	-	06:31	3:0	-	06:33	3:1	-	06:34	*:2	H	06:36
4:2	-	06:38	*:8	C	06:46	1:9	-	06:57	3:6	-	07:01
*:2	C	07:03	5:7	-	07:06	2:0	-	07:08	4:1	-	07:14
2:5	-	07:17	1:8	-	07:21	1:6	-	07:35	1:7	-	07:41
4:9	-	07:46	2:1	-	07:47	2:4	-	07:48	4:0	-	07:51
4:5	-	07:58	5:8	-	07:58	2:3	-	08:12	2:2	-	08:14
5:9	-	08:27	4:6	-	08:28	3:5	-	08:40	5:0	-	08:42
4:7	-	08:50	*:5	H	08:55	5:1	-	09:05	3:4	-	09:14
4:8	-	09:14	5:5	-	09:20	*:5	C	09:23	3:3	-	09:33
5:2	-	09:39	5:4	-	09:46	7:5	-	09:57	7:5	-	10:04
5:3	-	10:13	3:2	-	10:15	7:4	-	10:16	7:6	-	10:21
7:3	-	10:35	7:7	-	10:42	7:2	-	10:52	7:8	-	10:59
7:1	-	11:11	7:0	-	11:22	7:9	-	11:22	ALL	H	DONE!

5 Other Aspects of Viral Computation

5.1 Toward Random Variation and Selective Survival

We have spoken of evolution, but to many, this concept doesn't seem to apply to computer programs in the same way it applies to biological systems. In its simplest form, we speak of systems evolving through human reprogramming, and indeed, the term evolution seems to accurately describe the process of change a system goes through in its life cycle, but this is only one way that programs can evolve.

Consider a collection virus that uses pseudo-random variables to slowly change the weighting of different collection strategies from generation to generation, and replicates with a probability associated with its profitability (the net fee collected after all expenses of collection). In this case, assuming that the parameters being varied relate to the success of the collection process in an appropriate manner, the 'species' of available viruses for the collection process will seem to 'evolve' toward a more profitable set of bill collectors.

If we use less variation on bill collectors that are more successful, we may tend toward local optima. To attain global optima, we may occasionally require enough randomness to shake loose from the local optima. Over time, we may find several local optima, each with

a fairly stable local population of bill collectors. Thus different species of bill collectors may coexist in the environment if there are adequate local niches for their survival. Cross breeding of species is feasible by taking selected parameters from different species to birth new bill collectors. Some will thrive, while some will not even survive. As the external environment changes, different species may perform better, and the balance of life will ebb and shift in response to the survival rates. This evolutionary process is commonly called "random variation and selective survival", and is roughly the equivalent of biological evolution as we now commonly speak of it.

5.2 Complex Behaviors, Generating Sets, and Communication

The behavior we are discussing is getting complex, involving local and global optimization, evolution over time, and even the coexistence of species in an environment; and yet the computer programs we are discussing are still quite simple. Our viral bill collectors consist of only a few pages of program code, and yet they perform the same tasks carried out by much larger programs. The inclusion of evolution in experimental systems has been accomplished in only a few lines of program code. We create a small "generating set" of instructions which creates a complex system over time through birth/death processes, random variation and selective survival, and the interaction of coexisting species in the environment. Even quite simple generating sets can result in very complex systems. In fact, in most cases, we cannot even predict the general form of the resulting system.

Our inability to accurately predict systemic behavior in complex systems stems, in general, from the fact that it is impossible to derive a solution to the 'halting problem', which is one of the fundamental unsolvable problems of computer science. The problem of determining whether a computer program will ever complete processing a problem was proven 'undecidable' for general purpose computers by Turing in 1936. *(A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", London Math Soc Ser 2, 1936.)* Through a mathematical proof technique called reduction, most of the 'interesting' behaviors of complex systems can be proven undecidable as well. More specifically, it was proven by Cohen *(see earlier 1985 reference)* that a virus can evolve in as general a fashion as a computer can compute, and therefore that the result of viral evolution is potentially as complex as Turing's computation.

It seems there is little we can do about predicting the behavior of general purpose evolutionary systems, but just as there are large classes of computer programs with predictable behavior, there are large classes of evolutionary systems with predictable behavior. Indeed, in the same way as we can generate computer programs from specifications, we can generate evolutionary systems from specifications, and guarantee that they will act within predefined

boundaries. In the case of the maintenance virus, we can even get enhanced system reliability with viral techniques. Unfortunately, we haven't yet developed our mathematical understanding of viruses in an environment to make good predictive models of these sorts of systems, but there is a general belief that many important problems are not intractable at the systemic level, and if that is true, we may be able to make good predictive models of the behavior of viral system.

One of the ways we can design predictable viral systems is by adding communications. Completely deaf and dumb viruses have a hard time surviving because they tend to be born and die without any controlling influences, and we get unstable situations which either consume too many resources and ruin the ecology or die from lack of sufficient biomass. With even rudimentary communications, viruses can survive far better. For example, most real-world computer viruses survive far better if they only infect programs that are not yet infected. Too much communication also makes viruses inefficient, because they have to address an increasingly global amount of information. We suspect that communications is beneficial to viral survival only to the extent that it helps to form stable population relative to the resources in the environment, but in terms of predictability, communications seems to be key.

The maintenance viruses described earlier provide a good example of viral communication. There is no direct communication between the maintenance viruses, but they end up communicating in the sense that what each virus does to the environment alters the actions of other viruses. For example, a maintenance virus that deletes temporary files that haven't been accessed in 24 hours or more will not delete any files that a previous maintenance virus has already deleted, since the earlier virus already consumed them. Since the latter virus acts differently based on the actions of the earlier virus, there is a rudimentary form of communication between the viruses via changes in the environment. In fact, humans communicate in much the same way; by making changes in the environment (e.g. sound waves) that affect other humans. The net effect is that maintenance viruses act more efficiently because they rarely interfere with each other.

6 The Future of Viral Computation

The possibilities for practical viruses are unbounded, but they are only starting to be explored. Unfortunately, viruses have gotten a bad name, partly because there are so many malicious and unauthorized viruses operating in the world. As of early 1992, there were over 1500 real-world viruses, with a newly designed unauthorized virus being detected in the global computing environment more than 4 times per day. If the computing community doesn't

act to counter these intrusions soon, society may restrict research in this area and delay or destroy any chance we have at exploiting the benefits of this new technology. There are now many useful tools for defending against malicious viruses and other integrity corruptions in computer systems, and they can often be implemented without undue restriction to normal user activity, but perhaps another tactic would also serve society well.

The tactic is simple; instead of writing malicious viruses, damaging other people's computer systems, hiding their identity, and risking arrest, prosecution, and punishment; virus writers could be provided with a legitimate venue for expressing their intellectual interest, and can get both positive recognition and financial rewards for their efforts. By changing the system of rewards and punishment, we may dramatically improve the global virus situation and simultaneously harness the creative efforts of virus writers for useful applications.

The emerging computer virus technology, like all new technology, is a two edged sword. Just as biological viruses can cause disease in humans, computer viruses can cause disease in computer systems, but in the same sense, the benefits of biological research on the quality of life is indisputable, and the benefits of computer virus research may same day pay off in the quality of our information systems, and by extension, our well being. To the extent that we can learn about our ecosystem systems by studying the informational ecosystems formed by computer viruses, we may save ourselves from the sorts of mistakes we have already made in dealing with our environment.

Somebody once said that computer systems are one of the greatest laboratory facilities we have. Their ability to model and mimic life and life-like situations is astounding both in its accuracy and in its ability to allow experiments that would be unconscionable or infeasible in any other laboratory. We have the unique opportunity to use this laboratory to get at the fundamental nature of living systems, if we can only get past our biases and get on with the important work awaiting us.

7 Summary, Conclusions, and Further Work

8 References

- **1)** F. Cohen, "Computer Viruses - Theory and Experiments", DOD/NBS 7th Conference on Computer Security, originally appearing in IFIP-sec 84 (1984), also appearing as invited paper in IFIP-TC11, "Computers and Security", V6#1 (Jan. 1987), pp 22-35 and other publications in several languages.
- **2)** F. Cohen, "Computer Viruses", ASP Press, PO Box 81270, Pittsburgh, PA 15217 USA, 1985, subsequently approved as a dissertation at the University of Southern California, 1986.
- **3)** A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", London Math Soc Ser 2, 1936.
- **4)** L. Adleman, "An Abstract Theory of Computer Viruses", Crypto-89
- **5)** F. Cohen, 'A Short Course on Computer Viruses', ASP Press, PO Box 81270, Pittsburgh, PA, 1991
- **6)** C. Langton, ed., "Artificial Life", Addison Wesley, 1989
- **7)** J. Rochlis and M. Eichin, "With Microscope and Tweezers: The Worm from MIT's Perspective", CACM, V32#6, June, 1989
- **8)** M. Harrison, W. Ruzzo, and J. Ullman, "Protection in Operating Systems", CACM V19#8, Aug 1976, pp461-471.
- **9)** F. Cohen, "A Cost Analysis of Typical Computer Viruses and Defenses", IFIP-TC11, "Computers and Security", 1991.
- **10)** F. Cohen, "Protection and Administration of Information Networks Under Partial Orderings", IFIP-TC11 Computers and Security, V6(1987) pp118-128.
- **11)** K. Popper, 'Logik der Forschung', 1935 (English translation 1959 in 'The Logic of Schientific Discovery')
- **12)** F. Cohen, 'A Formal Definition of Computer Worms and Some Related Results', IFIP-TC11 'Computers and Security' (accepted, pending publication)
- **13)** R. Dawkins, 'The Selfish Gene',
- **14)** D. Denning, Lattices
- **15)** A. Rich, 'Mulisp', The Soft Warehouse, 3615 Harding Ave., Suite 505, Honolulu, HI 96816-3735 USA