Biologically Inspired Defenses Against Computer Viruses

Jeffrey O. Kephart, Gregory B. Sorkin,
William C. Arnold, David M. Chess,
Gerald J. Tesauro, and **Steve R. White**
High Integrity Computing Laboratory
IBM Thomas J. Watson Research Center
Yorktown Heights NY 10598

Abstract

Today's anti-virus technology, based largely on analysis of existing viruses by human experts, is just barely able to keep pace with the more than three new computer viruses that are written daily. In a few years, intelligent agents navigating through highly connected networks are likely to form an extremely fertile medium for a new breed of viruses. At IBM, we are developing novel, biologically inspired anti-virus techniques designed to thwart both today's and tomorrow's viruses. Here we describe two of these: a neural network virus detector that learns to discriminate between infected and uninfected programs, and a computer immune system that identifies new viruses, analyzes them automatically, and uses the results of its analysis to detect and remove all copies of the virus that are present in the system. The neural-net technology has been incorporated into IBM's commercial anti-virus product; the computer immune system is in prototype.

1 Introduction

Each day, an army of perhaps a few hundred virus writers around the world produces three or more new computer viruses.[1] An army of comparable size, the anti-virus software developers (representing an approximately $100 million per year industry), works feverishly to analyze these viruses, develop cures for them, and frequently distribute software updates to users.

Currently, the battle is roughly even. Our statistics, based on observation of a sample population of several hundred thousand machines for several years [Kephart and White, 1993; Kephart *et al.*, 1993], suggest that in medium to large businesses roughly 1% of all computers become infected during any given year. The world's computer population has been inconvenienced, but despite dire predictions [Tippett, 1991] it has not been incapacitated. Most of the anti-virus products in common usage have been reasonably effective in detecting and removing viruses. Within our sample population, only 10% of all known viruses (about 360 of 4000 at the time of writing) have been observed "in the wild" — in real incidents. Several viruses that used to be relatively common now qualify for inclusion on an endangered species list. Today, computer viruses are a manageable nuisance.

Several worrisome trends threaten to turn the balance in the favor of computer virus authors. First, the rate at which new viruses are created, already on the verge of overwhelming human experts, has the potential to increase substantially. Second, continued increases in interconnectivity and interoperability among the world's computers, designed to benefit computer users, are likely to be a boon to DOS and Macintosh viruses as well. Theoretical epidemiological studies indicate that the rate at which computer viruses spread on a global scale can be very sensitive to the rate and the degree of promiscuity of software exchange [Kephart and White, 1991; 1993; Kephart *et al.*, 1993; Kephart, 1994b]. Anticipated increases in both factors threaten to increase substantially the speed of spread and the pervasiveness of these traditional types of virus. In addition, mobile intelligent agents [Chess *et al.*, 1995; Harrison *et al.*, 1994] will soon navigate the global network, potentially serving as a fertile medium for a new breed of rapidly-spreading virus that exploits the itinerancy of its host by leaving behind copies of itself wherever its host goes. Traditional methods of detecting and removing viruses, which rely upon expert analysis by humans and subsequent distribution of the cure to users, would be orders of magnitude too slow to deal with viruses that spread globally within days or hours.

To address these problems, we have developed a variety of biologically inspired anti-virus algorithms and techniques that replace many of the tasks traditionally performed by human virus experts, thus permitting much faster, automatic response to new viruses.

The term "computer virus", coined by Adleman in the early 1980's [Cohen, 1987], is suggestive of strong analogies between computer viruses and their biological namesakes. Both attach themselves to a small functional unit (cell or program) of the host individual (organism or computer), and co-opt the resources of that unit for the purpose of creating more copies of the virus. By us-

[1] This figure is based on the number of distinct new viruses that have been received by us during the last year.

ing up materials (memory[2]) and energy (CPU[3]), viruses can cause a wide spectrum of malfunctions in their hosts. Even worse, viruses can be toxic. In humans, diptheria is caused by a toxin produced by virally-infected bacteria [Levine, 1992]. Some computer viruses are similarly toxic, being deliberately programmed to cause severe harm to their hosts. One notorious example, the Michelangelo virus, destroys data on a user's hard disk whenever it is booted on March 6th.

It is therefore natural to seek inspiration from defense mechanisms that biological organisms have evolved against diseases. The idea that biological analogies might be helpful in defending computers from computer viruses is not original to us [Murray, 1988]. But to our knowledge we are the first to take these analogies seriously, to deliberately design and implement anti-virus technology that is inspired by biology, and incorporate it into a commercial anti-virus product.

First, we will briefly describe what computer viruses are, how they replicate themselves, and why their presence in a system is undesirable. Then, we shall describe the typical procedures used by human experts to analyze computer viruses, and explain why these methods are unlikely to remain viable a few years from now. Then, we shall discuss two complementary anti-virus techniques that are inspired by biological systems that learn: a neural-network virus detector and a computer immune system.

2 Background

2.1 Computer viruses and worms

Computer viruses are self-replicating software entities that attach themselves parasitically to existing programs. They are endemic to DOS, Macintosh, and other microcomputer systems. When a user executes an infected program (an executable file or boot sector), the viral portion of the code typically executes first. The virus looks for one or more victim programs to which it has write access (typically the same set of programs to which the user has access), and attaches a copy of itself (perhaps a deliberately modified copy) to each victim. Under some circumstances, it may then execute a payload, such as printing a weird message, playing music, destroying data, *etc.* Eventually, a typical virus returns control to the original program, which executes normally. Unless the virus executes an obvious payload, the user is unlikely to notice that anything is amiss, and will be completely unaware of having helped a virus to replicate. Viruses often enhance their ability to spread by establishing themselves as resident processes in memory, persisting long after the infected host finishes its execution (terminating only when the machine is shut down). As resident processes, they can monitor system activity continually, and identify and infect executables and boot sectors as they become available.

Over a period of time, this scenario is repeated, and the infection may spread to several programs on the user's system. Eventually, an infected program may be copied and transported to another system electronically or via diskette. If this program is executed on the new system, the cycle of infection will begin anew. In this manner, computer viruses spread from program to program, and (more slowly) from machine to machine. The most successful PC DOS viruses spread worldwide on a time scale of months [Kephart and White, 1993].

Worms are another form of self-replicating software that are sometimes distinguished from viruses. They are self-sufficient programs that remain active in memory in multi-tasking environments, and they replicate by spawning copies of themselves. Since they can determine when to replicate (rather than relying on a human to execute an infected program), they have the potential to spread much faster than viruses. The Internet worm of 1988 is said to have spread to several thousand machines across the United States in less than 24 hours [Eichin, 1989; Spafford, 1989].

2.2 Virus detection, removal and analysis

Anti-virus software seeks to detect all viral infections on a given computer system and to restore each infected program to its original uninfected state, if possible.

There are a variety of complementary anti-virus techniques in common usage; taxonomies are given in [Spafford, 1991; Kephart *et al.*, 1993]. *Activity monitors* alert users to system activity that is commonly associated with viruses, but only rarely associated with the behavior of normal, legitimate programs. *Integrity management systems* warn the user of suspicious changes that have been made to files. These two methods are quite general, and can be used to detect the presence of hitherto unknown viruses in the system. However, they are not often able to pinpoint the nature or even the location of the infecting agent, and they can sometimes flag or prevent legitimate activity, disrupting normal work or leading the user to ignore their warnings altogether.

Virus scanners search files, boot records, memory, and other locations where executable code can be stored for characteristic byte patterns (called "signatures") that occur in one or more known viruses. Providing much more specific detection than activity monitors and integrity management systems, scanners are essential for establishing the identity and location of a virus. Armed with this very specific knowledge, *disinfectors*, which restore infected programs to their original uninfected state, can be brought into play. The drawback of scanning and repair mechanisms is that they can be applied only to known viruses, or variants of them. Furthermore, each individual virus strain must be analyzed in order to extract both a signature and information that permits a disinfector to remove the virus. Scanners and disinfectors require frequent updates as new viruses are discovered, and the analysis can entail a significant amount of effort on the part of human virus experts.

Whenever a new virus is discovered, it is quickly dis-

[2]The Jerusalem virus increases the size of an executable by 1813 bytes each time it infects it, eventually causing it to be too large to be loaded into memory [Highland, 1990].

[3]The Internet worm caused the loads on some Unix machines to increase by two orders of magnitude [Eichin, 1989; Spafford, 1989].

tributed among an informal, international group of anti-virus experts. Upon obtaining a sample, a human expert disassembles the virus and then analyzes the assembler code to determine the virus's behavior and the method that it uses to attach itself to host programs. Then, the expert selects a "signature" (a sequence of perhaps 16 to 32 bytes) that represents a sequence of instructions that is guaranteed to be found in each instance of the virus, and which (in the expert's estimation) is unlikely to be found in legitimate programs. This signature can then be encoded into the scanner. The attachment method and a description of the machine code of the virus can be encoded into a verifier, which verifies the identity of a virus that has been found by the scanner. Finally, a reversal of the attachment method can be encoded into a disinfector.

Virus analysis is tedious and time-consuming, sometimes taking several hours or days, and even the best experts have been known to select poor signatures — ones that cause the scanner to report false positives on legitimate programs. Alleviation of this burden is by itself enough to warrant a serious attempt to automate virus analysis. The anticipated speed with which viruses of the future may spread is an even stronger argument in favor of endowing anti-virus software with the ability to deal with new viruses on its own.[4] The rest of this paper describes two techniques for achieving this goal.

3 Generic Detection of Viruses

Two methods of computer virus identification have already been introduced: the overly broad, *ex post facto* detection provided by activity monitors and integrity management systems, and the overly specific detection offered by virus scanners. Somewhere in between is the ideal "generic detector": taking a program's code as input, it determines whether the program is viral or non-viral. Perfect generic detection is an algorithmically "undecidable" problem: as observed by [Cohen, 1987], it is reducible to the halting problem. However, imperfect generic detection that is good in practice is possible, and is naturally viewed as a problem in automatic pattern classification. Standard classification techniques encompass linear methods and non-linear ones such as nearest-neighbor classification, decision trees, and multi-layer neural networks.

Within the problem of the generic detection of viruses, detection of "boot sector viruses" is both an important and relatively tractable sub-problem. A boot sector is a small sequence of code that tells the computer how to "pick itself up by its bootstraps". For IBM-compatible PC's, boot sectors are exactly 512 bytes long; their main function is to load and execute additional code stored elsewhere.

[4] At the very least, anti-virus software must handle a majority of viruses well enough to prevent them from spreading. For the foreseeable future, it will continue to be important for human virus experts to analyze carefully any viruses that appear in the wild to corroborate the results of the automated analysis and to determine any side effects that the virus may cause in infected systems.

Although there are over 4,000 different file-infecting viruses and only about 250 boot-sector viruses, of the 20 viruses most commonly seen 19 are boot viruses, and account for over 80% of all virus incidents. Boot viruses similarly dominate the rolls of newly observed viruses, so an ability to detect new boot sector viruses is significant in the war against viruses.

Detecting boot viruses is a relatively limited pattern classification task. Legitimate boot sectors all perform similar functions. Viral boot sectors also all perform similar functions, before passing control to a legitimate boot sector loaded from elsewhere.

For this application, false positives are critical. False negatives mean missed viruses, and since viruses occur fairly rarely, so will false negatives. Also, if a classifier does let a virus slip by, the outcome is no worse than if no virus protection were in place. On the other hand, false positives can occur any time, and will leave a user worse off than she would have been without virus protection. Moreover, a false positive on one legitimate boot sector will mean false-positive events on thousands of computers. False positives are not tolerable.

Nearest-neighbor classification might seem to be a simple, attractive approach to the classification of legitimate and viral boot sectors. Natural measures of the difference between two boot sectors include the Hamming distance between them (considered as 512-element vectors), or the edit distance [Crochemore, 1994] between them (considered as text strings). To classify a new boot sector, the procedure would find the "nearest" of the 250 known boot sector viruses and 100 legitimate boot sectors (a representative if not comprehensive set that we have collected), and classify the new boot sector as viral if its nearest neighbor is viral, legitimate if its nearest neighbor is legitimate.

Unfortunately, nearest-neighbor classification performs poorly for this problem. A viral boot sector can be just a short string of viral code written over a legitimate boot sector, so in any overall comparison, the virus will be more similar to the legitimate boot sector it happened to overwrite than to any other virus. This says that what makes viral boot sectors viral is not any overall quality but the presence of specific viral functions.

These functions can be used to construct a virus classifier. For example, one common action is for a virus to reduce the apparent size of available memory so that space taken up by the virus will not be noticed. Although this action may be variously implemented in machine code, most machine code implementations match one of a few simple patterns. (A fictitious pattern typifying the form is `C31B****AC348F**90D3D217` — about 10 fixed bytes and some wildcards.) Of the viruses that lower memory in less conventional ways, most still contain a 2-byte pattern weakly indicative of the same function, but more prone to false positives. Similar strong and weak patterns describe other common viral functions.

Using expert knowledge of viral and non-viral boot sectors and several days of extensive experimentation, we hand-crafted an *ad hoc* classifier (see Figure 1). The classifier scans a boot sector for the presence of patterns that provide strong or weak evidence for any of four viral

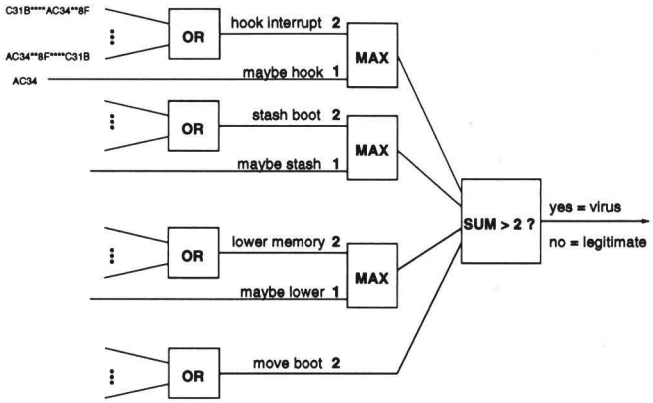

Figure 1: A hand-crafted multi-level classifier network. Eliminating the "MAX" boxes produces a more conventional neural network, but it is inferior, even when the seven weights are optimized.

functions. One point is credited for weak evidence, and two points for strong evidence. A boot sector is classified as viral if its total score is 3 or higher. This classifier performed well on the 350 examples, with a false-negative rate of about 18% and a false-positive rate too small to measure over the 100 negative examples. That is, 82% of viruses were detected, and no legitimate boot sector was classified as viral.

We hoped to develop a procedure for automatically constructing a virus classifier, using similar features as inputs to a neural network. Since the *ad hoc* classifier incorporated knowledge of all of the available boot sectors, there was a possibility that it suffered from overfitting, in which case it would generalize poorly on new data. It would be much easier to assess the generalization performance of an automatically constructed classifier. Also, we hoped that algorithmic extraction of features and optimization of network weights might give even better classification performance, especially in the false-positive measure. Finally, we believed that an automated procedure would adapt much more readily to new trends in boot sector viruses. If substantially new types of boot sector viruses became common, we could simply retrain the classifier — a much easier task than hacking on an *ad hoc* classifier, or re-writing it from scratch.

Essentially, what we did was this. We extracted a set of 3-byte strings, or "trigrams", appearing frequently in viral boot sectors but infrequently in legitimate ones. The presence (1) or absence (0) of the strings defined the input vector to a single-layer neural network. (See Figure 2.) Its weights were "trained" over about half the examples, and the resulting network's performance tested on the other half. During the development of the automatic classifier, we encountered novel challenges in *feature pruning* and *ill-defined learning* that we think represent interesting general issues in learning. These will be introduced in the course of a more detailed description of the classifier's construction.

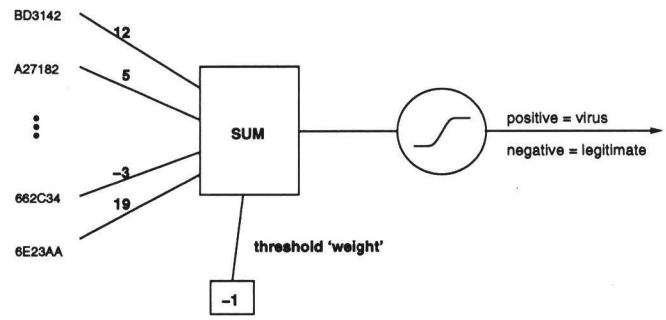

Figure 2: Single-layer neural classifier with about 50 input features and weights determined algorithmically. The trigrams and weights are fictitious.

3.1 Feature selection

The first step in the construction was the selection of byte strings to act as features. Where a human expert is able to use high-level understanding of viruses, knowledge of machine code, and natural intelligence to select complex feature patterns containing wildcards, for algorithmic feature generation we contented ourselves with simple 3-byte features. A training set with 150 512-byte viral boot sectors includes 76,500 "trigrams", of which typically 25,000 are distinct.

This is where the first challenge, *feature pruning*, comes in. A well known principle in machine learning states that the number of training examples must be considerably larger than the number of adjustable parameters to reliably give good generalization to test examples [Hertz *et al.*, 1991]. With 150 viral and 45 non-viral training examples, a network must have well fewer than 195 weights — say about 50 — implying a lesser or equal number of inputs. Somehow the 25,000 trigrams must be winnowed down to 50.

Since what is desired are trigrams that are indicative of viral as opposed to legitimate behavior, it is natural to remove trigrams appearing too frequently in legitimate boot sectors. Eliminating all trigrams which appear even once in the 45 legitimate training examples reduces the 25,000 candidate trigrams by only about 8%. On the reasoning that trigrams occurring frequently in PC programs in general are analogous to the English word "the" and not salient features, further winnowing can be done. Eliminating trigrams with frequency over 1/200,000 (occurring on average more than once in 200,000 bytes) again reduces the number about 8%, leaving about 21,000 of the original 25,000 candidate features. Much more drastic pruning is required.

It is provided by selecting trigram features which figure importantly in the viral training set. One way to do this would be to select trigrams occurring at least some number of times in the viral training set, but this leaves some viral samples unrepresented by any trigrams. A better approach comes from selecting a "cover" of trigrams: a set of trigrams with at least one trigram representing each of the viral samples. In fact, we can afford something close to a 4-cover, so that each viral sample is represented by 4 different trigrams in the set. (A few

samples have fewer than 4 representatives in the full set of 21,000 trigrams, in which case they are only required to be 3-covered, 2-covered, or singly covered, as possible.) Four-covering produces a set of about 50 trigram features, few enough to be used as input to a neural net.

(Even so, a complete two-layer network with h hidden nodes would have h times as many weights as inputs, which here is prohibitive even for an h of 2 or 3; this is why we used a single-layer network.)

Reassuringly, most of the trigrams were substrings of or otherwise similar to the more complex patterns of the *ad hoc* classifier. However, there were a few trigrams that could not be related to any of these patterns, and on expert inspection they turned out to define a meaningful new feature class.

3.2 Classifier training and performance

By construction, the selected trigrams are very good features: within the training set, no legitimate boot sector contains any of them, and most of the viral boot sectors contain at least 4. Paradoxically, the high quality of the features poses the second challenge, what we have called the problem of *ill-defined learning*. Since no negative example contains any of the features, any "positive" use of the features gives a perfect classifier.

Specifically, the neural network classifier of Figure 2 with a threshold of 0 and any positive weights will give perfect classification on the training examples, but since even a single feature can trigger a positive, it may be susceptible to false positives on the test set and in real-world use. The same problem shows up as an instability when the usual back-propagation [Rumelhart *et al.*, 1986] training procedure is used to optimize the weights: larger weights are always better, because they drive the sigmoid function's outputs closer to the asymptotic ideal values of -1 and 1.

In fact all that will keep a feature's ideal weighting from being infinite is the feature's presence in some negative example. Since none of the features were present in any negative example, our solution was to introduce new examples. One way is to add a set of examples defined by an identity matrix. That is, for each feature in turn, an artificial negative example is generated in which that feature's input value is 1 and all other inputs are 0. This adds one artificial example for each trigram feature; it might be better to emphasize features which are more likely to appear by chance.

To do so, we used 512 bytes of code taken from the initial "entry point" portions of many PC programs to stand in as artificial legitimate boot sectors; the thought was that these sections of code, like real boot sectors, might be oriented to machine setup rather than performance of applications. Of 5,000 such artificial legitimate boot sectors, 100 contained some viral feature. (This is about as expected. Each selected trigram had general-code frequency of under 1/200,000, implying that the chance of finding any of 50 trigrams among 512 bytes is at most 13%; the observed rate for the artificial boot sectors was 5%.) Since not all of the 50 trigrams occurred in any artificial boot sector, we used this approach in combination with the "identity matrix" one.

At this point the problem is finally in the form of the most standard sort of (single-layer) feed-forward neural network training, which can be done by back-propagation. In typical training and testing runs, we find that the network has a false-negative rate of 10–15%, and a false-positive rate of 0.02% as measured on artificial boot sectors.[5] (Given the trigrams' frequencies of under 1/200,000, if their occurrences were statistically independent, the probability of finding two within some 512 bytes would be at most 0.8%.) Consistent with the 0.02% false-positive rate, there were no false positives on any of the 100 genuine legitimate boot sectors.

There was one eccentricity in the network's learning. Even though all the features are indicative of viral behavior, most training runs produced one or two slightly negative weights. We are not completely sure why this is so, but the simplest explanation is that if two features were perfectly correlated (and some are imperfectly correlated), only their total weight is important, so one may randomly acquire a negative weight and the other a correspondingly larger positive weight.

For practical boot virus detection, the false-negative rate of 15% or less and false-positive rate of 0.02% are an excellent result: 85% of new boot sector viruses will be detected, with a tiny chance of false positives on legitimate boot sectors. In fact the classifier, incorporated into IBM AntiVirus, has caught several new viruses. There has also been at least one false positive, on a "security" boot sector with virus-like qualities, and not fitting the probabilistic model of typical code. Rather than specifically allowing that boot sector, less than an hour of re-training convinced the neural network to classify it negatively; this may help to reduce similar false positives.

Of the 10 or 15% of viruses that escape detection, most do so not because they fail to contain the feature trigrams, but because the code sections containing them are obscured in various ways. If the obscured code is captured by independent means, the trigrams can be passed on to the classifier and these viruses too will be detected.

4 A Computer Immune System

Although generic virus detection works well for boot-sector viruses, and may eventually prove useful for file infectors as well, at least two drawbacks are inherent in the technique:

1. New viruses can be detected only if they have a sufficient amount of code in common with known viruses.
2. The method is appropriate for viral detection only; it is incapable of aiding in the removal of a virus from an infected boot sector or file. The only way

[5] Comparison of this classifier's 85% detection rate on test data with the 82% rate of the hand-crafted one is more favorable than the numbers suggest. The rate for the neural net was measured over an independent test set, where for the hand-crafted detector there was no training–testing division. Measured over all examples (and especially if trained over all examples), the network's detection rate exceeds 90%.

to eliminate the infection is to erase or replace the infected boot sector or file.

The generic classifier could be viewed as an analog of the "innate", or non-adaptive, non-specific immune system that is present in both vertebrates and lower animals. One important component of this innate immunity can be viewed as a sort of generic classifier system, in which the features on which recognition is based include:

1. the presence of certain proteins that are always present on self-cells, but usually not on foreign cells,[6]
2. the presence of double-strand RNA, which appears in much larger concentrations in a particular class of viruses than it does in mammalian cells [Marrack, 1993], and
3. the presence of a peptide that begins with an unusual amino acid (formyl methionine) that is produced copiously by bacteria, but only in minute amounts by mammals [Marrack, 1993].

This generic classification is coupled with a generic response to a pathogen that either disables it or kills it.

However, vertebrates have evolved a more sophisticated, adaptive immune system that works in concert with the innate immune system, and is based on recognition of *specific* pathogens.[7] It exhibits the remarkable ability to detect and respond to previously unencountered pathogens, regardless of their degree of similarity to known pathogens. This is precisely the sort of defensive capability that we seek against computer viruses.

Figure 3 provides an overview of our design for an adaptive computer immune system. The immune system responds to virus-like anomalies (as identified by various activity and integrity monitors) by capturing and analyzing viral samples. From its analysis, it derives the means for detecting and removing the virus. Many components of the computer immune system are working in the laboratory, and are providing useful data that is incorporated into IBM AntiVirus, IBM's commercial anti-virus product.

The remainder of this section will be devoted to a discussion of the various components of the immune system design, along with their relationship to analogous biological principles. Further exploration of some biological analogies can be found in [Kephart, 1994a]. First, we shall consider the set of components that are labeled as being currently in IBM AntiVirus: anomaly detection, scanning for known viruses, and removal of known viruses. Then, we shall discuss some of the components that are labeled as being currently in the virus lab: sample capture using decoys, algorithmic virus analysis, and signature extraction. These components are all functioning prototypes. Finally, we shall discuss a mechanism by which one machine can inform its neighbors about viral infections.

4.1 Anomaly detection

The fundamental problem faced by both biological and computer immune systems is to distinguish between malignant and benign entities that enter the individual. Due to the high degree of stability of body chemistry in individual vertebrates during their lifetimes, their immune systems can replace this difficult task with the much simpler one of distinguishing self from non-self. This is a nice hack, because "self" is much easier to define and recognize than "benign". The biological immune system can simply implement the xenophobic strategy: "Know thyself (and reject all else)." This strategy errs on the side of false positives (*i.e.* false rejection of benign entities), but except in cases of blood transfusions and organ transplants, these mistakes are of little consequence.[8]

In computers, the same xenophobic strategy is an important component of anomaly detection. Integrity monitors can use checksums or other methods[9] to determine whether an existing executable has changed. However, this is only a partial solution. The nature of "self", *i.e.* the collection of software on an individual computer, is continually shifting over time — much more so than in biological organisms. People continually add new software to their system, and update existing software by buying new versions or compiling new source code. The fact that an executable is new or has changed is not nearly enough to warrant suspicion. An array of other monitors and heuristics employ a complementary "Know thine enemy" strategy: the nature of the anomaly must be strongly indicative of a virus. Some components of the anomaly detector trigger on suspicious dynamical behaviors (such as one process writing to an executable or boot record, or unusual sequences of operating system calls, perhaps involving interception of particular interrupts); others trigger on static properties having to do with the exact nature of a change that has been identified by the integrity monitor.

4.2 Scanning for known viruses

If the anomaly detector has been triggered, the system is scanned for all known viruses. Since there are currently at least 4000 known PC DOS viruses, this means that exact or slightly inexact matches to approximately 4000 signatures, each in the range of roughly 16 to 32 bytes long, are searched in parallel. This is in itself an interesting string matching problem, and efficient search methods are an active area of research for us. Much

[6] These proteins inactivate complement, a class of proteins that bind to cells, and attract the attention of other components of the immune system, which kill the cell [Janeway, 1993].

[7] This extra sophistication pits the quick adaptability of the immune system, which occurs within a single individual over the course of a few days, against the similarly quick evolutionary adaptability of pathogens (due to their short life-cycles). Due to their much slower life-cycles, it is doubtful that vertebrates could hold their own if their immune systems had to rely on evolution alone.

[8] Another important class of false positives are autoimmune reactions, which are sometimes induced by biochemical changes that occur at puberty (thus changing the nature of "self").

[9] A novel method for integrity monitoring that is based on a close analogy to T cells is described in [Forrest *et al.*, 1994].

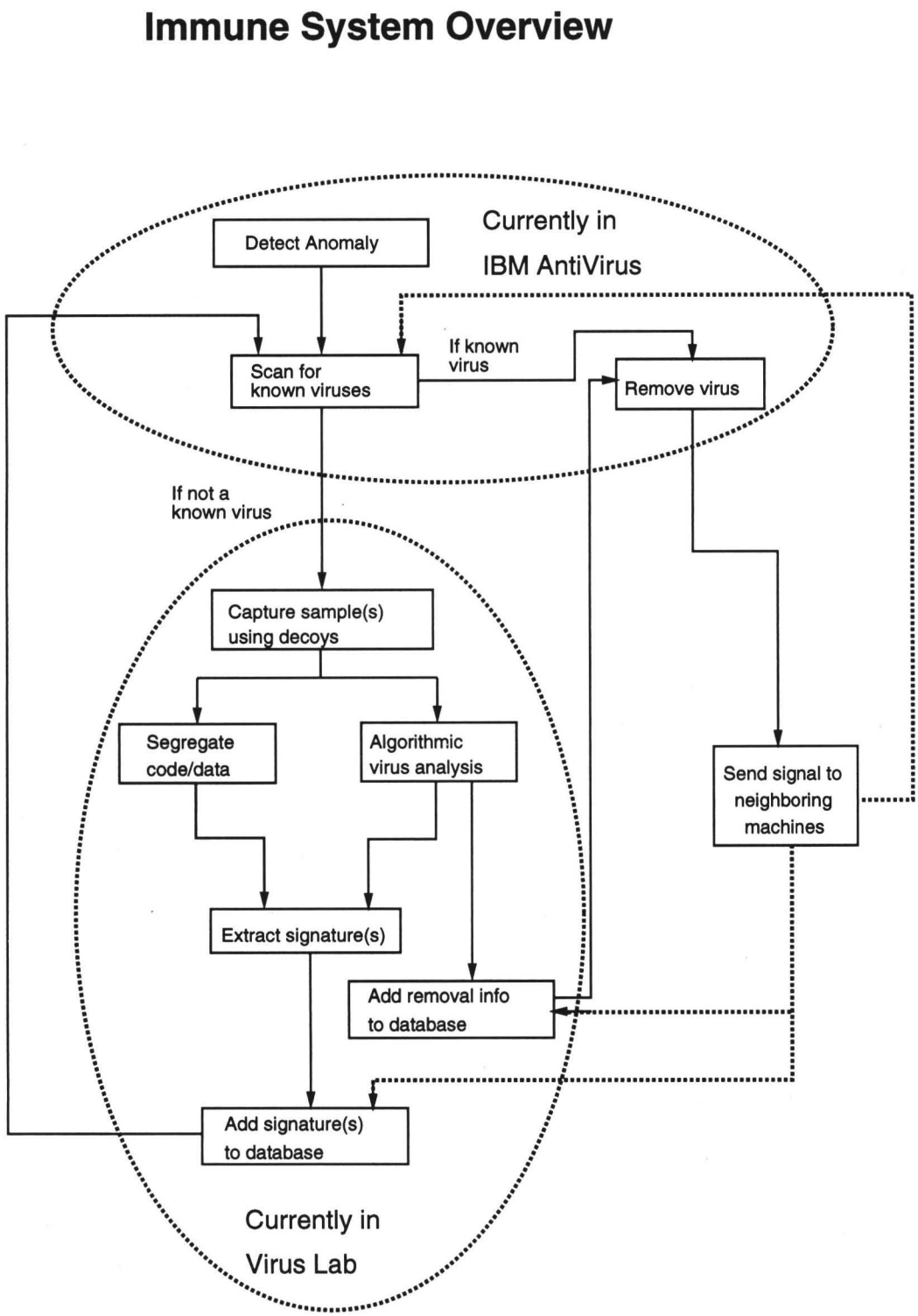

Figure 3: The main components of the proposed immune system for computers and their relationship to one another.

more impressive than any string matching algorithm we could ever hope to devise, however, is the parallel search carried out by the vertebrate immune system, in which roughly 10 million different types of T-cell receptors and 100 million different types of antibodies and B-cell receptors are continually patrolling the body in search of antigen [Janeway, 1993]. Just as a computer virus scanner recognizes viruses on the basis of (perhaps inexact) matches to a fragment of the virus (the signature), T-cell and B-cell receptors and antibodies recognize antigen by binding (strongly or weakly, depending on the exactness of the match) to fragments of the antigen (consisting of linear sequences of 8 to 15 amino acids, in the case of T cells [Janeway, 1993]).

Matching to fragments rather than the entire antigen is a physical necessity in the biological immune system; in computers, this strategy is not absolutely necessary, but it has some important advantages. Matching to fragments is more efficient in time and memory, and permits the system to recognize slight variants, particularly when some mismatches are tolerated. These issues of efficiency and variant recognition are relevant for biology as well.

For both biological and computer immune systems, an ability to recognize variants is essential because viruses tend to mutate frequently. If an exact match were required, immunity to one variant of a virus would confer no protection against a slightly different variant. Similarly, vaccines would not work, because they rely on the biological immune system's ability to synthesize antibodies to tamed or killed viruses that are similar in form to the more virulent one that the individual is being immunized against.

4.3 Virus removal

In the biological immune system, if an antibody encounters antigen, they bind together, and the antigen is effectively neutralized. Thus recognition and neutralization of the intruder occur simultaneously. Alternatively, a killer T cell may encounter a cell that exhibits signs of being infected with a particular infecting agent, whereupon it kills the host cell. This is a perfectly sensible course of action, because an infected host cell is slated to die anyway, and its assassination by the killer T cell prevents the viral particles from reaching maturation.

A computer immune system can take the same basic approach to virus removal: it can erase or otherwise inactivate an infected program. However, an important difference between computer viruses and biological viruses raises the possibility of a much gentler alternative.

In biological organisms, most infected cells would not be worth the trouble of saving even if this were possible, because cells are an easily-replenished resource.[10]

In contrast, each of the applications run by a typical computer user is unique in function and irreplaceable (unless backups have been kept, of course). Since a user would be likely to notice any malfunction, all but the most ill-conceived computer viruses attach themselves to their host in such a way that they do not destroy its function. Viruses tend to merely rearrange or reversibly transform their hosts. Thus an infected program is usually expressible as a reversible transformation of the uninfected original.

When the scanner identifies a particular program as being infected with a particular virus, the first step in our removal procedure is to verify that the virus is identical to a known strain. Verification is based upon checksums of regions of viral code that are known to be invariant (perhaps after an appropriate decryption operation) across different instances of the virus. The exact location and structure of the virus must have been derived beforehand, and expressed in terms of a language understood by the verification algorithm. If the verification does not succeed, an attempt to remove the virus by this means is considered too risky, and another more generic virus removal method (beyond the scope of this paper) is brought into play. If the verification succeeds, a repair algorithm carries out the appropriate sequence of steps required for removing that virus, expressed in a simple repair language. The sequence of steps is easily derived from an analysis of the locations (and transformations, if any) of all of the portions of the original host.

Although the analysis required to extract verification and removal information has traditionally been performed by human experts, we shall discuss in a later subsection an automated technique for obtaining this information.

4.4 Decoys

Suppose that the anomaly detector has found evidence of a virus, but that the scanner cannot identify it as any of the known strains. Most current anti-virus software will not be able to recover the host program unless it was deliberately stored or analyzed[11] prior to becoming infected. Ideally, one would like to have stronger evidence that the system really is infected, and to know more about the nature of the virus, so that all instances of it (not just the one discovered by the anomaly detector) can be found and eliminated from the system.

In the computer immune system, the presence of a previously unknown virus in the system can be established with much greater certainty than can be provided by the anomaly detector. The idea is to lure the virus into infecting one or more members of a diverse suite of "decoy" programs. Decoys are designed to be as attractive as possible to those types of viruses that spread most successfully. A good strategy for a virus to follow is to infect programs that are touched by the operating system in some way. Such programs are most likely to be executed by the user, and thus serve as the most successful vehicle for further spread. Therefore, the immune system entices a putative virus to infect the decoy programs by executing, reading, writing to, copying, or otherwise manipulating them. Such activity attracts the attention of many viruses that remain active in memory even after they have returned control to their host. To

[10] Neurons are a notable exception, but they are protected from most infections by the blood-brain barrier [Seiden, 1995].

[11] Generic disinfection methods can store a small amount of information about an uninfected program, and use this information to help reconstruct it if it subsequently becomes infected.

catch viruses that do not remain active in memory, the decoys are placed in places where the most commonly used programs in the system are typically located, such as the root directory, the current directory, and other directories in the path. The next time the infected file is run, it is likely to select one of the decoys as its victim. From time to time, each of the decoy programs is examined to see if it has been modified. If any have been modified, it is almost certain that an unknown virus is loose in the system, and each of the modified decoys contains a sample of that virus. These virus samples are stored in such a way that they will not be executed accidentally. Now they are ready to be analyzed by other components of the immune system.

The capture of a virus sample by the decoy programs is somewhat analogous to the ingestion of antigen by macrophages [Paul, 1991]. Macrophages and other types of cells break antigen into small peptide fragments and present them on their surfaces, where they are subsequently bound by T cells with matching receptors. A variety of further events can ensue from this act of binding, which in one way or another play essential roles in recognizing and removing the pathogen. Capture of an intruder by computer decoys or biological macrophages allows it to be processed into a standard format that can be interpreted by other components of the immune system, provides a standard location where those components can obtain information about the intruder, and primes other parts of the immune system for action.

4.5 Automatic virus analysis

Typically, a human expert applies a deep understanding of machine instruction sequences to virus analysis. Sometimes, this is combined with observation of the effects of the virus on a program.

Our automatic virus analysis algorithm is much less sophisticated in its knowledge of machine code, but makes up for this deficiency by making use of more data: specifically, several samples of the virus. Once a few samples of the virus have been captured, the algorithm compares the infected decoys with one another and with the uninfected decoys to yield a precise description of how the virus attaches to any host. The description is completely independent of the length and contents of the host, and to some extent can accommodate self-encrypting viruses. A pictorial representation of one particularly simple infection pattern is presented in Fig. 4.

Automatic virus analysis provides several useful types of information:

1. The location of all of the pieces of the original host within an infected file, independent of the content and length of the original host. This information is automatically converted into the repair language used by the virus removal component of IBM AntiVirus.
2. The location and structure of all components of the virus. Structural information includes the contents of all regions of the virus that are invariant across different samples. This information has two purposes:

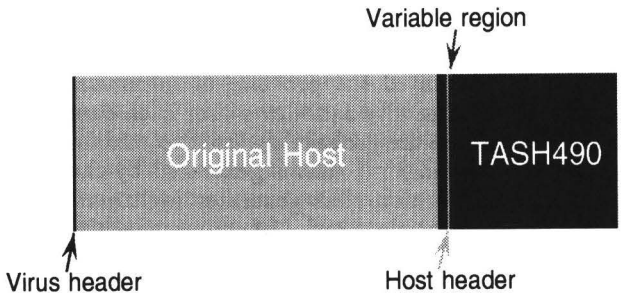

Figure 4: Pictorial representation of attachment pattern and structure of the TASH490 virus, derived completely automatically.

   (a) It is automatically converted into the verification language used by the verification component of IBM AntiVirus.
   (b) It is passed to the automatic signature extraction component for further processing.

4.6 Automatic signature extraction

The basic goal of automatic signature extraction is to choose a signature that is very likely to be found in all instances of the virus, and very unlikely to be found accidentally in uninfected programs. In other words, we wish to minimize false negatives and false positives. False negatives are dangerous because they leave the user vulnerable to attack. False positives are extremely annoying to customers, and so infuriating to vendors of falsely-accused software that they have led to at least one lawsuit.

To minimize false negatives, we first start with the contents of the invariant regions that have been identified by the automatic virus analysis procedure. However, it is quite conceivable that not all of the potential variation has been captured within the samples. As a general rule, non-executable "data" portions of programs, which can include representations of numerical constants, character strings, work areas for computations, *etc.*, are inherently more likely to vary from one instance of the virus to another than are "code" portions, which represent machine instructions. The origin of the variation may be internal to the virus, or a virus hacker might deliberately change a few data bytes in an effort to elude virus scanners. To be conservative, "data" areas are excluded from further consideration as possible signatures. Although the task of separating code from data is in principle somewhat ill-defined, there are a variety of methods, such as running the virus through a debugger or virtual interpreter, which perform reasonably well.

At this point, there are one or more sequences of invariant machine code bytes from which viral signatures could be selected. We take the set of candidate signatures to be all possible contiguous blocks of S bytes found in these byte sequences, where S is a signature length that is predetermined or determined by the algorithm itself. (Typically, S ranges between approximately 12 and 36.) The remaining goal is to select from among the

candidates one or perhaps a few signatures that are least likely to lead to false positives.

We have formulated the problem of minimizing the false positive probability as follows. For each candidate signature, estimate the probability for it to match a random sequence of length S that is generated by the same probability distribution that generates legitimate software on the relevant platform. (Of course, machine code is written by people or compilers, not probability distributions, so such a probability distribution is a theoretical and somewhat ill-defined construct, but we estimate its statistics from a set of over 10,000 DOS and OS/2 programs, constituting half a gigabyte of code.) Then, we select the candidate signature for which the estimated probability is the smallest.

In slightly more detail, the key steps of the algorithm are as follows:

1. Form a list of all n-grams (sequences of n bytes; $1 \leq n \leq n_{max}$) contained in the input data. (n_{max} is typically 5 or 8.)
2. Calculate the frequency of each such n-gram in the "self" collection.
3. Use a simple formula that chains together conditional probabilities based on the measured n-gram frequencies to form a "false-positive" probability estimate for each candidate signature, *i.e.* the probability that it matches a random S-byte sequence chosen from code that is statistically similar to "self".
4. Select the signature with the lowest estimated false-positive probability.

Characterizations of this method [Kephart and Arnold, 1994] show that the probability estimates are poor on an absolute scale, due to the fact that code tends to be correlated on a longer scale than 5 or 8 bytes. However, the relative ordering of candidate signatures is rather good, so the method generally selects one of the best possible signatures. In fact, judging from the relatively low false-positive rate of the IBM AntiVirus signatures (compared with that of other anti-virus vendors), the algorithm's ability to select good signatures may be *better* than that achieved by human experts.

In a sense, the signature extraction algorithm combines elements of outmoded and current theories of how the vertebrate immune system develops antibodies and immune-cell receptors to newly encountered antigen. The *template* theory, which held sway from the mid-1930's until the early 1960's, was that antibodies and receptors molded themselves around the antigen. The *clonal selection* theory holds that a vast, random repertoire of antibodies and receptors is generated, and those that recognize self are eliminated during the maturation phase. Of the remaining antibodies and receptors, at least a few will match any foreign antigen that is encountered. The clonal selection theory gained favor in the 1960's, and is currently accepted [Paul, 1991].

Our automatic signature extraction method starts out looking like the template theory. Instead of generating a large random collection of signatures that might turn out to be useful someday, we take the collection of code for a particular virus as our starting point in choosing a signature. However, we do share one important element with the clonal selection theory: elimination of self-recognizing signatures. In fact, the automatic signature extraction method is even more zealous in this endeavor than clonal selection, in that it only retains the "best" signature.

4.7 Immunological memory

The mechanisms by which the vertebrate immune system retains a lifelong memory of viruses to which it has been exposed are quite complex, and are still the subject of study and debate.

By contrast, immunological memory is absolutely trivial to implement in computers. During its first encounter with a new virus, a computer system may be "ill", *i.e.* it will devote a fair amount of time and energy (or CPU cycles) to virus analysis. After the analysis is complete, the extracted signature and verification/repair information can be added to the appropriate known-virus databases. During any subsequent encounter, detection and elimination of the virus will occur very quickly. In such a case the computer can be thought of as "immune" to the virus.

4.8 Fighting self-replication with self-replication

In the biological immune system, immune cells with receptors that happen to match a given antigen reasonably well are stimulated to reproduce themselves. This provides a very strong selective pressure for good recognizers, and by bringing a degree of mutation into play, the immune cell is generally able to come up with immune cells that are extremely well-matched to the antigen in question.

One can view this as a case in which self-replication is being used to fight a self-replicator (the virus) in a very effective manner. One can cite a number of other examples in nature and medical history where this strategy has been employed, such as the deliberate use of the myxoma virus in the 1950's to curtail an exploding rabbit population in Australia [McNeill, 1976; Levine, 1992].

The self-replicator need not itself be a virus. In the case of the worldwide campaign against smallpox, launched by the World Health Organization in 1966, those who were in close contact with an infected individual were all immunized against the disease. Thus immunization spread as a sort of anti-disease among smallpox victims. This strategy was amazingly successful: the last naturally occurring case of smallpox occurred in Somalia in 1977 [Bailey, 1975].

We propose to use a similar mechanism, which we call the "kill signal", to quell viral spread in computer networks. When a computer discovers that it is infected, it can send a signal to neighboring machines. The signal conveys to the recipient the fact that the transmitter was infected, plus any signature or repair information that might be of use in detecting and eradicating the virus. If the recipient finds that it is infected, it sends the signal to *its* neighbors, and so on. If the recipient

is not infected, it does not pass along the signal, but at least it has received the database updates, effectively immunizing it against that virus.

Theoretical modeling has shown the kill signal to be extremely effective, particularly in topologies that are highly localized or sparsely connected [Kephart and White, 1993; Kephart, 1994b].

5 Conclusion and Perspective

The development of the generic virus detector and the computer immune system were primarily motivated by practical concerns: human virus experts are on the verge of being overwhelmed, and we need to automate as much of what they do as possible.

The generic virus detector was incorporated into IBM AntiVirus in May, 1994, and since that time it has successfully identified several new boot viruses. It is the subject of a pending patent. Most of the components of the computer immune system are functioning as very useful prototypes in our virus isolation laboratory; we use them every day to process the large sets of new viruses that arrive in the mail from other virus experts around the world. The immune system itself is the subject of a pending patent, as are several of its components, including automatic virus analysis and automatic signature extraction.

Our eventual goal is to incorporate the immune system into IBM AntiVirus and, a few years from now, in networks inhabited by itinerant software agents. More implementation and more invention, guided in part by the biological metaphor, lie ahead.

Although our primary motivation for developing a computer immune system is practical, it is interesting to adopt a more philosophical perspective.

Consider the history of how humans have handled disease. For millions of years, our sole defense against infectious disease was our immune system, and it has done a good job of defending us from most infectious diseases. When we are suffering from the common cold, we may experience a few days of discomfort while the immune system figures out how to recognize and eradicate the virus, but we usually survive the attack. However, a minority of diseases, like smallpox or AIDS, are not handled effectively by the immune system. Fortunately, during the last few centuries, we have made tremendous advances in our understanding of infectious diseases at both the macroscopic and microscopic levels, and medical practices based on this understanding now augment the capabilities of our natural immune system.

A few hundred years ago, disease began to be understood at the macroscopic level. In 1760, Daniel Bernoulli, the founder of mathematical physics, was interested in determining whether a particular form of inoculation against smallpox would be generally beneficial or harmful to society. Formulating and solving a mathematical model, he found that inoculation could be expected to increase the average life expectancy by three years. His work founded the field of mathematical epidemiology [Bailey, 1975]. Observational epidemiology received a major boost from John Snow, who in 1854 was able to deduce the origin of a severe cholera outbreak in London by plotting the addresses of victims on a city map [Bailey, 1975].

The macroscopic approaches of Snow and Bernoulli proved fruitful even before bacteria and viruses were identified as the underlying cause of infectious disease in the late 19th century. During the 20th century, research at the microscopic level has supplemented epidemiology. Electron microscopy and X-ray crystallography brought the structure of viruses into view in the 1930's, and the fascinating complexities of their life cycle and biochemistry began to be studied intensively in the mid-1940's. These advances established *terra firma* on which mathematical epidemiologists could build their models.

Today, epidemiologists, in the detective role pioneered by John Snow, discover new viruses [Garrett , 1994]. Biochemists, molecular biologists, and geneticists work to elucidate the secrets of viruses, and to create safe and effective vaccines for them. Epidemiologists use intuition and mathematics to develop plans for immunizing populations with these vaccines. The eradication of smallpox from the planet in 1977 is probably the greatest triumph of this multi-disciplinary collaboration.

Interestingly, the history of man's defense against computer viruses is almost exactly reversed. Computer viruses were first understood at the microscopic level, thanks to the pioneering work of Fred Cohen in the early 1980's [Cohen, 1987]. As soon as the first DOS viruses began to appear in 1987 [Highland, 1990], they were dissected in great detail, and the first primitive anti-virus software was written. It was not until 1990 that the first real attempts were made to understand the spread of computer viruses from a macroscopic perspective [Kephart and White, 1991; 1993; Tippett, 1990; 1991]. Finally, in the mid-1990's, we are proposing to give computers what humans and other vertebrates have always relied upon as a first line of defense against disease: an immune system.

The Center for Disease Control does not get worked up when a new strain of the common cold sweeps through a population. Instead, they concentrate their limited resources on finding cures for horrible diseases such as AIDS. Currently, the world community of anti-virus researchers (the computer equivalent of the CDC) squanders lots of time analyzing the computer equivalents of the common cold. Our hope is that a computer immune system will deal with most of the standard, run-of-the-mill viruses quietly and effectively, leaving just a small percentage of especially problematic viruses for human experts to analyze.

References

[Bailey, 1975] Norman T.J. Bailey. *The Mathematical Theory of Infectious Diseases and Its Applications.* Oxford University Press, second edition, 1975.

[Chess *et al.*, 1995] David Chess, Benjamin Grosof, Colin Harrison, David Levine, and Colin Parris. Itinerant agents for mobile computing. *IEEE Personal Communications Magazine*, 1995. Submitted.

[Cohen, 1987] Fred Cohen. Computer viruses, theory and experiments. In *Computers and Security*, volume 6, pages 22–35, 1987.

[Crochemore, 1994] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

[Eichin, 1989] M.W. Eichin and J.A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 326–343, 1989.

[Forrest *et al.*, 1994] Stephanie Forrest, Alan S. Perelson, Lawrence Allen, and Rajesh Cherukuri. Self – nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, May 1994.

[Garrett , 1994] Laurie Garrett. *The Coming Plague: Newly Emerging Diseases in a World Out of Balance*. Farrar, Straus and Giroux, 1994.

[Harrison *et al.*, 1994] Colin Harrison, David Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? Technical Report 19887, IBM Research Report, 1994. http://www.research.ibm.com/xw-d953-mobag-ps.

[Hertz *et al.*, 1991] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[Highland, 1990] Harold J. Highland. *Computers and Security's Computer Virus Handbook*. Elsevier, 1990.

[Janeway, 1993] Charles A. Janeway, Jr. How the immune system recognizes invaders. *Scientific American*, 269(3):72–79, September 1993.

[Kephart and Arnold, 1994] Jeffrey O. Kephart and William C. Arnold. Automatic extraction of computer virus signatures. In R. Ford, editor, *Proceedings of the Fourth International Virus Bulletin Conference*, pages 179–194. Virus Bulletin, Ltd., September 1994.

[Kephart and White, 1991] Jeffrey O. Kephart and Steve R. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 343–359, May 1991.

[Kephart and White, 1993] Jeffrey O. Kephart and Steve R. White. Measuring and modeling computer virus prevalence. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–15, May 1993.

[Kephart *et al.*, 1993] Jeffrey O. Kephart, Steve R. White, and David M. Chess. Computers and epidemiology. *IEEE Spectrum*, 30(5):20–26, May 1993.

[Kephart, 1994a] Jeffrey O. Kephart. A biologically inspired immune system for computers. In R. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, 1994.

[Kephart, 1994b] Jeffrey O. Kephart. How topology affects population dynamics. In C. Langton, editor, *Artificial Life III: Studies in the Sciences of Complexity*, pages 447–463. Addison-Wesley, 1994.

[Levine, 1992] Arnold J. Levine. *Viruses*. Scientific American Library. Freeman, 1992.

[Marrack, 1993] Philippa Marrack and John W. Kappler. How the immune system recognizes the body. *Scientific American*, 269(3):81–89, September 1993.

[McNeill, 1976] W.H. McNeill. *Plagues and Peoples*. Doubleday, 1976.

[Murray, 1988] W.H. Murray. The application of epidemiology to computer viruses. In *Computers and Security*, volume 7, pages 130–150, 1988.

[Paul, 1991] William E. Paul, editor. *Immunology: Recognition and Response ... Readings from Scientific American*. Freeman, 1991.

[Rumelhart *et al.*, 1986] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

[Seiden, 1995] Philip E. Seiden. Note on auto-immunity. Private communication, 1995.

[Spafford, 1989] E.H. Spafford. The internet worm program: An analysis. *Computer Comm. Review*, 19, 1989.

[Spafford, 1991] E.H. Spafford. Computer viruses: A form of artificial life? In D. Farmer, C. Langton, S. Rasmussen, and C. Taylor, editors, *Artificial Life II: Studies in the Sciences of Complexity*, pages 727–747. Addison-Wesley, 1991.

[Tippett, 1990] Peter S. Tippett. Computer virus replication. *Comput. Syst. Eur.*, 10:33–36, 1990.

[Tippett, 1991] Peter S. Tippett. The kinetics of computer virus replication: A theory and preliminary survey. In *Safe Computing: Proceedings of the Fourth Annual Computer Virus and Security Conference*, pages 66–87, March 1991.