Self-Nonself Discrimination in a Computer*

Stephanie Forrest
Dept. of Computer Science
University of New Mexico
Albuquerque, N.M. 87131-1386
forrest@cs.unm.edu

Alan S. Perelson
820 Los Arboles Ln.
Santa Fe, N.M. 87501
asp@santafe.edu

Lawrence Allen
Dept. of Computer Science
University of New Mexico
Albuquerque, N.M. 87131-1386

Rajesh Cherukuri
Dept. of Computer Science
University of New Mexico
Albuquerque, N.M. 87131-1386
raj@cs.unm.edu

Abstract

The problem of protecting computer systems can be viewed generally as the problem of learning to distinguish *self* from *other*. We describe a method for change detection which is based on the generation of T cells in the immune system. Mathematical analysis reveals computational costs of the system, and preliminary experiments illustrate how the method might be applied to the problem of computer viruses.

1 Introduction

The problem of ensuring the security of computer systems includes such activities as detecting unauthorized use of computer facilities, guaranteeing the integrity of data files, and preventing the spread of computer viruses. In this paper, we view these protection problems as instances of the more general problem of distinguishing *self* (legitimate users, corrupted data, etc.) from *other* (unauthorized users, viruses, etc.). We introduce a change-detection algorithm that is based on the way that natural immune systems distinguish self from other. Mathematical analysis of the expected behavior of the algorithm allows us to predict the conditions under which it is likely to perform reasonably. Based on this analysis, we also report preliminary results illustrating the feasibility of the approach on the problem of detecting computer viruses (demonstrating that the algorithm can be practically applied remains an open problem), and finally, we suggest that the general principles can be readily applied to other computer security problems.

Current commercial virus detectors are based on three distinct technologies: activity monitors, signature scanners, and file authentication programs. The system that we describe is essentially a file authentication method, or change detector. Although our initial testing has been in a virus detection setting, the algorithm may be more applicable to other change-detection problems. There are several significant differences between the algorithm described here and more conventional approaches to change detection, such as checksums and message-digest algorithms: (1) the checking activity can be distributed over many sites with each site having a unique signature, (2) the quality of the check can be traded off against the cost of performing check, (3) protection is symmetric in the sense that the change detector and protected data set are mutually protective, and (4) the algorithm for generating the change detectors is computationally expensive, although checking is cheap, so it would be difficult to modify a protected file and then alter the detectors in such a way that the modification could not be detected. As with other authentication methods, our method relies on the guarantee that the data to be protected are uncorrupted at the time that the detectors are generated.

There are several change-detection tools available which employ a variety of change-detection methods and signature functions, e.g., Tripwire [6]. Tools such as Tripwire devote considerable attention to the important problems of administration, portability, and

[0]In Proceedings of 1994 IEEE Symposium on Research in Security and Privacy (in press).

reporting. Our work is properly viewed as an algorithm, comparable in nature to a signature function, which might be incorporated into a tool like Tripwire. As we mentioned above, there are several features which distinguish our algorithm from conventional signature methods, in particular, our "signatures" are expensive to generate (although cheap to check, especially if the checking activity is distributed across multiple sites) and multiple signatures exist for each data set. These distinguishing features have advantages and disadvantages, and it remains to be seen what setting is most appropriate for an algorithm with these features.

Our approach relies on three important principles:

- Each copy of the detection algorithm is unique. Most protection schemes need to protect multiple sites (e.g., multiple copies of software, multiple computers on a network, etc.). In these environments, we believe that any single protection scheme is unlikely to be effective, because once a way is found to avoid detection at one site, then all sites are vulnerable. Our idea is to provide each protected location with a unique set of detectors. This implies that even if one site is compromised, other sites will remain protected.

- Detection is probabilistic. One consequence of using different sets of detectors to protect each entity is that probabilistic detection methods are feasible. This is because an intrusion at one site is unlikely to be successful at multiple sites. By using probabilistic methods our system can achieve high system-wide reliability at relatively low cost (time and space). The price, of course, is a somewhat higher chance of intrusion at any one site.

- A robust system should detect (probabilistically) any foreign activity rather than looking for specific known patterns of intrusion. Most virus detection programs work by scanning for unique patterns (e.g., digital signatures) that are known at the time the detection software is distributed. This leaves systems vulnerable to attack by novel means. Like other change detectors, our algorithm learns what self is and notices (probabilistically) any deviation from self.

1.1 System Overview

The algorithm has two phases:

1. Generate a set of detectors. Each detector is a string that does not match any of the protected data (see below for a careful definition of "match"). This "censoring" phase is illustrated in Figure 1.

2. Monitor the protected data by comparing them with the detectors. If a detector is ever activated, a change is known to have occurred (shown in Figure 2).

This might seem to be an unpromising approach. If we view the set of data being protected as a set of strings over a finite alphabet, and a change to that data as any string not in the original set, then we are proposing to generate detectors for (almost) all strings not in the original data set. Surprisingly, this algorithm turns out to be feasible mathematically—that is, a fairly small set of detector strings has a very high probability of noticing a random change to the original data. Further, the number of detectors can remain constant as the size of the protected data grows. Figures 1 and 2 illustrate how the algorithm works. Each copy of the detection system generates its own unique valid set of detectors once, and then runs the monitoring program regularly (for example, as a background process) to check for changes.

Before describing the procedure in detail, we need to describe what it is that we are trying to detect. We reduce the detection problem to the problem of detecting whether or not a string has been changed, where a change could be a modification to an existing string or a new string being added to self. The algorithm will fail to notice deletions. The string could be a string of bits (and hence, anything that can be represented in a digital computer), a string of assembler instructions, a string of data, etc. However, as will become apparent later, the method appears to be most relevant for strings that do not change over time, that is, the protected strings need to be fairly stable. We define *self* to be the string to be protected, and *other* to be any other string. Note that it will sometimes be convenient to view self as an unordered collection of substrings, and other times as one long string that is the concatenation of the substrings. We use the term "collection" instead of "set" because we do not remove or check for duplicates. Duplicates appear with extremely low frequency, but technically, the collections both of protected strings and detectors are "multisets."

To generate valid detectors, we first split (logically) the self string into equal-size segments. Originally, we chose to split the strings to facilitate our mathematical analysis of the system, which allows us to predict the probability of detection. However, it has turned out to have other advantages, such as making it much easier to detect certain kinds of computer viruses, and suggesting extensions to the system. As an example,

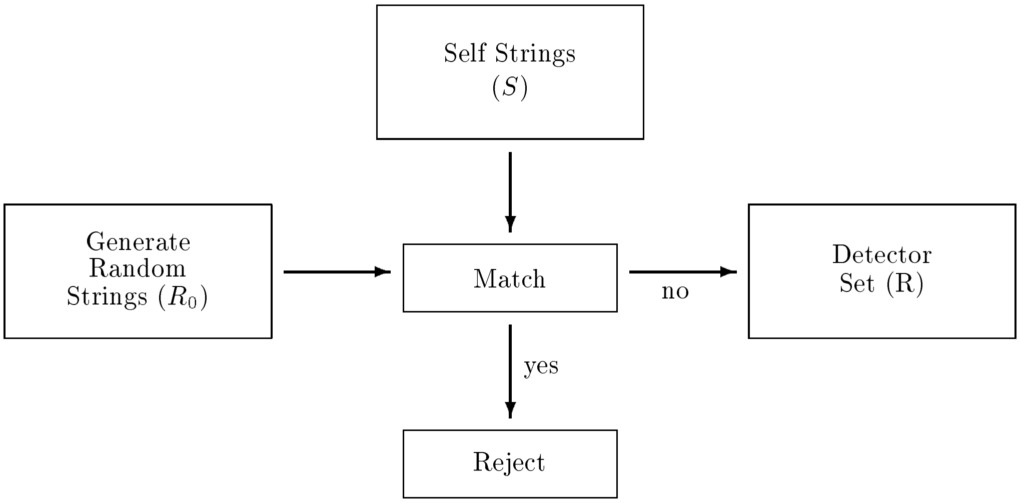

Figure 1: Generation of Valid Detector Set (Censoring) .

we might break the following 32-bit string into eight substrings, each of length four:

0010 1000 1001 0000 0100 0010 1001 0011

This produces the collection S of self (sub)strings to be protected (S contains all of the substrings). The second step is to generate random strings (call this collection R_0), and then match the strings of R_0 against the strings in S. Strings from R_0 that match self (see Section 1.2) are eliminated. Strings that do not match any of the strings in S become members of the detector collection (R), also called the *repertoire*. This procedure is called *censoring*. Continuing the example, suppose R_0 contains the following four random strings: 0111, 1000, 0101, 1001. Then, R will consist of two strings, 0111 and 0101, the strings 1000 and 1001 being eliminated because they each match a string in S.[1] The censoring procedure is illustrated in Figure 1.

Once a collection R of detector strings has been produced, the state of *self* can be monitored by continually matching strings in S against strings in R. This is achieved by choosing one string from S and one string from R and testing to see if they match. In our implementation, the pairings are made deterministically—each string is chosen for matching in a fixed order. The detectors are checked in the order they were produced. For the self strings, the order is determined, for example, by the order of instructions in the program. Alternatively, the procedure could be randomized. If ever a match is found, then it is concluded that S has changed.

In the example, suppose that one bit of the last self string (0011) is changed to produce 0111. Then, at some point in the monitoring process, it would be noticed that the "self" string (0111) matches one of the detector strings (the string 0111), and a change would be reported.

[1] In practice, the procedure is to generate random strings sequentially, and to continue generating them until R has a sufficient number of elements. R_0 is useful conceptually for predicting how many strings must be generated to produce a R of a certain size.

1.2 Matching

A perfect *match* between two strings of equal length means that at each location in the string, the symbols are identical. The example in Section 1 shows perfect matching between strings defined over the alphabet $\{0, 1\}$. Since perfect matching is extremely rare between strings of any reasonable length, a partial matching rule is needed. We relax the matching requirement by using a matching rule that looks for r contiguous matches between symbols in corresponding positions. Thus, for any two strings x and y, we say that $match(x, y)$ is true if x and y agree (match) at at least r contiguous locations. See Figure 3 for an example.

The matching rule can be applied to strings defined over any alphabet of symbols. In the most general case, the strings will be over the alphabet $\{0, 1\}$, representing any bit pattern that can be stored in a computer. At a higher level, strings might be defined over a particular machine instruction set. Figure 4 shows an example of censoring with $r = 2$.

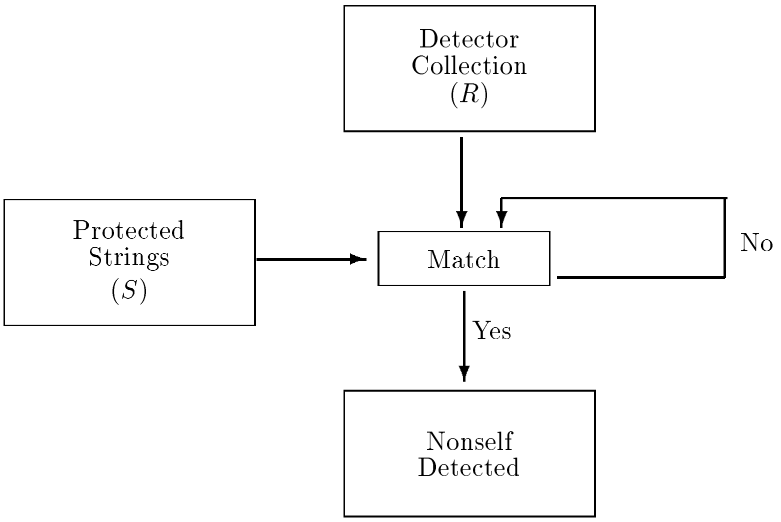

Figure 2: Monitor Protected Strings for Changes.

X	ABADCBAB
Y	CAGDCBBA

Figure 3: Example Matching Rule. The two strings, x and y defined over the four-letter alphabet $\{A, B, C, D\}$ match at three contiguous locations (underlined). Thus, $match(x, y)$ is false for $r = 4$ or greater, since x and y agree at 3 contiguous locations. $match(x, y)$ is true for $r = 3$ or less.

It is useful to know the probability P_M that two random strings match at at least r contiguous locations. If:

$m =$ the number of alphabet symbols,
$l =$ the number of symbols in a string (length of the string), and
$r =$the number of contiguous matches required for a match,

then[9, 8],

$$P_M \approx m^{-r}[(l-r)(m-1)/m+1].$$

The approximation is only good if $m^{-r} << 1$, so we use the exact formula for the cases in which the approximation fails [12]. Table 1 illustrates the effect of varying r and l on P_M for different values of m. The first row shows the configuration we have used in most of our experiments. Setting $r = 8$ corresponds to a one-byte change. The first four rows of the table show the linear increase in P_M as the length of the string (l) increases. Rows one and five show the exponential decrease in P_M as r increases. Finally, the last eight rows show the dramatic effect on P_M of increasing the alphabet size.

m	r	l	P_M
2	8	32	0.0502023
2	8	64	0.108697
2	8	128	0.2151
2	8	256	0.391316
2	16	32	0.000137329
2	16	64	0.000381437
2	16	128	0.000869474
2	16	256	0.00184483
128	8	32	$3.33067 * 10^{-16}$
128	8	64	$7.77156 * 10^{-16}$
128	8	128	$1.66533 * 10^{-15}$
128	8	256	$3.44169 * 10^{-15}$
128	16	32	~ 0.0
128	16	64	~ 0.0
128	16	128	~ 0.0
128	16	256	~ 0.0

Table 1: Example values of P_M for varying values of m (alphabet size), r (number of contiguous matches required for a $match$), and l (string length).

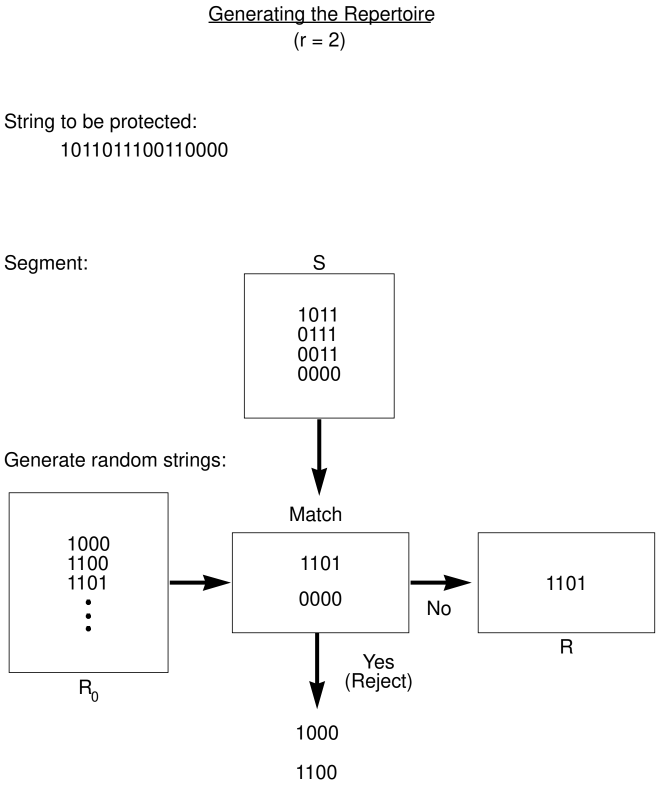

Figure 4: Generating the repertoire. The string to be protected is logically segmented into four equal-length "self" strings (stored in S). To generate the repertoire, random strings are produced in the box labeled R_0 and matched against each of the self strings. The first two strings, 1000 and 1100, are eliminated because they both match self string 0000 at at least two contiguous positions. The string 1101 fails to match any string in self at at least two contiguous positions, so it is accepted into the repertoire (box labeled "R").

2 Probability of Detection

Since detection is probabilistic, we need to make accurate estimates of these probabilities for different configurations of the change-detection system. This section describes how we make our predictions. The following analysis is taken from [1].

Suppose that we have some string that we want to protect. As we mentioned before, this string could be an application program, some data, or any other element of a computer system that is stored in memory. Using the algorithm described in Section 1.1, we would like to estimate the number and size of detector strings that will be required to ensure that an arbitrary change to the protected string is detected with some fixed probability.

We make the following definitions and calculations:

N_{R_0} = The number of initial detector strings (before censoring).

N_R = The number of detector strings after censoring (size of the repertoire).

N_S = The number of self strings.

P_M = The probability of a match between 2 random strings.

f = The probability of a random string not matching any of the N_S self strings.
$= (1 - P_M)^{N_S}$.

P_f = The probability that N_R detectors fail to detect an intrusion.

If P_M is small and N_S is large, then

$$f \approx e^{-P_M N_S}$$

and,

$$N_R = N_{R_0} \times f \tag{1}$$

$$P_f = (1 - P_M)^{N_R}. \tag{2}$$

(3)

If P_M is small and N_R is large, then

$$P_f \quad \approx \quad e^{-P_M N_R}. \tag{4}$$

Thus,

$$N_R = N_{R_0} \times f = \frac{-lnP_f}{P_M}.$$

Solving 1 and 2 for N_{R_0}, we get the following:

$$N_{R_0} \quad = \quad \frac{-lnP_f}{P_M \times (1-P_M)^{N_S}}. \tag{5}$$

This formula allows us to predict the number of initial strings (N_{R_0}) that will be required to detect a random change, as a function of the probability of detection ($1 - P_f$), the number of self strings being protected (N_S), and the matching rule (P_M). N_{R_0} is minimized by choosing a matching rule such that

$$P_M \approx \frac{1}{N_S}. \tag{6}$$

The foregoing analysis allows us to estimate the computational costs of the algorithm in the following way. First, we observe that the method relies on two basic operations: (1) generating a random string of fixed length and (2) comparing two strings to see if they meet the matching criterion (more than r contiguous matches). We assume that these operations take constant time. Then, the time complexity of Phase I will be proportional both to the number of strings in R_0 (i.e., N_{R_0}) and the number of strings in S (i.e., N_S). Equation 5 estimates N_{R_0} based on the size of the protected data set (N_s), the reliability of detection that is required (P_f), and the particular matching rule that is being used (P_M). The cost of complete checking in Phase II will be proportional to the number of strings in R (i.e., N_R) and the number of strings in S (i.e., N_S).

Based on the above analysis, we can make several observations about the algorithm:

1. It is tunable: we can choose a desired probability of detection (P_f), and then estimate the number of detector strings required as a function of the size of N_S (the strings to be protected) by using Equations 2 and 6.

   Since an increased probability of detection results in increased computational expense (due to the increased size of R_0 and R), one can choose a desired probability of detection by determining (a) how fatal a single intrusion would be, and (b) how much redundancy exists in the system (see Item 5 below).

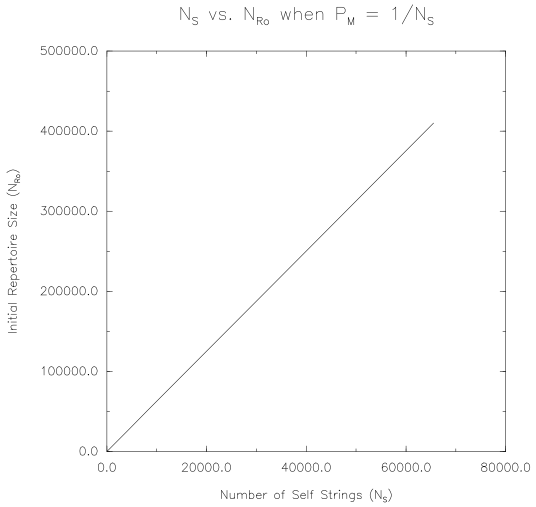

Figure 5: N_S vs. N_{R_0} when $P_M = \frac{1}{N_S}$. X-axis shows the number of self strings (N_S) and the Y-axis predicts the initial repertoire size (N_{R_0}) using the formula $N_{R_0} = \frac{-lnP_f}{P_M \times (1-P_M)^{N_S}}$.

2. N_R is independent of N_S for fixed P_M and P_f (Equation 2). That is, the size of the detector set does not necessarily grow with the number of strings being protected. This implies that it is possible to protect very large data sets efficiently.

3. If N_R, P_f, and P_M are fixed, then N_{R_0} grows exponentially with N_S. This exponential factor is unfortunate in one respect, but it does imply that once a set of detectors has been produced (say, using a supercomputer) that it would be virtually impossible for a malicious agent to change self and then change the detector set so that the change was unnoticed. N_{R_0} can be controlled by choosing $P_M = \frac{1}{N_S}$. Figure 5 illustrates the dramatic effect of minimizing N_{R_0} for each different value of N_S. We are currently studying several promising methods for reducing the size of N_{R_0} which do not rely on changing the matching rule for each new N_S.

4. The probability of detection increases exponentially with the number of independent detection algorithms.
   If N_t = the number of copies of the algorithm, then
   $$P_{\text{system fails to detect}} = (P_f)^{N_t}.$$
   This feature is the primary advantage of the algorithm. For example, Table 2 shows that with

one copy of the detection algorithm ($N_t = 1$), 46 detectors can protect a data set (of any size) with 90.6% reliability. With only ten different sites ($N_t = 10$), the same system-level reliability can be obtained with less than four detectors per site.

5. Detection is symmetric. Changes to the detector set are detected by the same matching process that notices changes to self. This implies that when a change is detected there is no *a priori* way to decide if the change was to self or to the detectors. The advantage is that self confers the same protection to the detector set that the detector set provides to self.

3 Experiments

Based on the above analysis, it is possible to design a wide variety of detection systems, each with different properties. In this section, we report some preliminary results based on our investigations of different parameter settings. We report three classes of experiments: experiments using random binary strings, experiments on SPARC instructions generated by compiling C programs, and in the DOS environment, experiments on COM files infected with actual computer viruses.

The first set of experiments show some of the implications of Equation 5 and confirms the estimates provided by the theoretical analysis. The remaining two sets of experiments illustrate how the method might be applied to the problem of detecting computer viruses.

Table 2 compares the theoretical and experimental probabilities (P_f) that a fixed number (N_R) of detectors will fail to detect a random change to self. It also compares the theoretical and experimental values for the initial repertoire size (N_{R_0}), thus providing an estimate of how costly Phase I of the algorithm is. The repertoire size (N_R) is set to 46 (i.e., 46 detectors, each consisting of 32 bits) and the target failure rate (P_f) is set to 0.1. The experimental procedure was as follows:

1. Fix P_f to 0.1.

2. Compute P_M using $m = 2$, $l = 32$, $r = 8$. Setting $r = 8$ corresponds to a matching rule that notices 1-byte changes.

3. Compute N_R based on P_f and P_M (using Equation 2) and round to next largest integer.

4. Repeat the following 1000 times:
   (a) Generate N_S random binary strings ($l = 32$).
   (b) Determine N_{R_0} experimentally by generating random strings until N_R valid detectors are found.
   (c) Test the detectors:
      i. Replace a string in self with one random string.
      ii. Compare the detector strings with the modified self strings.
      iii. If any of the detectors matches the new string (using the partial matching rule described earlier), report a modification.

5. Compare the mean P_f obtained over the 1000 trials with the P_f of (1).

Using this procedure, we obtained close agreement between the theoretical predictions and the observed results, as shown in Table 2. This experiment establishes a worst-case baseline for the algorithm. It is "worst case" in the sense that there is only one set of detectors (no advantage from distributing the detection task), the self strings are generated randomly, and changes to self consist of replacing a single string (changing at most 32 bits). For example, 128 self strings can be protected by a repertoire consisting of 46 detectors. These detectors detect one random change to self 84.3% of the time. Additionally, we can see that the exponential cost of generating the detector set is already significant (34,915), even for 128 self strings (a modest amount of data to protect). However, if the detection task were distributed over 100 sites, then each site would need to generate only one valid detector, would use an initial repertoire of 269 and would achieve at least a 98% detection rate.

These and other similar experiments indicate that there is good agreement between experimental and predicted values for P_f. The desired P_f can be achieved either by fixing N_{R_o} or N_R.

We also conducted several tests using C programs compiled for a SPARC processor. These experiments are interesting because they illustrate the effect of using a larger alphabet. The tests differed in the method used to generate the infected program. In all of the tests a C source code file was first compiled. The resulting object file was then disassembled into SPARC instructions which were each mapped to a single ASCII character. This produced one long string in which each symbol represented a single op-code. The string was then split into substrings, each 32 ASCII symbols long, representing the collection S of self strings. Thus, each string was defined over an alphabet of size 104 ($m = 104$) and of length 32 ($l = 32$).[2]

[2] The full SPARC instruction set is larger than 104, but we

N_S	Experimental N_{Ro}	Theory N_{Ro}	Experimental P_f
8	69(6.06)	69	0.085(0.009)
16	105(11.99)	105	0.104(0.010)
24	156(20.09)	158	0.110(0.011)
32	240(32.49)	239	0.099(0.009)
40	360(53.49)	361	0.107(0.010)
48	549(82.98)	545	0.133(0.011)
56	829(133.06)	823	0.109(0.010)
64	1253(218.77)	1243	0.124(0.010)
72	1876(318.55)	1876	0.112(0.010)
80	2872(495.61)	2833	0.116(0.010)
88	4327(781.78)	4277	0.130(0.011)
96	6618(1343.56)	6458	0.130(0.011)
104	9903(2082.86)	9750	0.135(0.011)
112	15074(3140.29)	14722	0.124(0.010)
120	22878(5283.80)	22228	0.154(0.011)
128	34915(8513.26)	33561	0.157(0.012)

Table 2: Theoretical and Experimental P_f with fixed N_R. Numbers reported are the mean of 1000 trials. Numbers in parentheses are standard deviations. N_R is set to 46 and the theoretical P_f is 0.094 for all entries. The theoretical N_{Ro} is calculated using the formula $N_{Ro} = \frac{N_R}{(1-P_M)^{N_S}}$. Theoretical P_f is calculated using the formula $P_f = (1\text{-}P_M)^{N_R}$ with $P_M = 0.0502$. It differs from 0.1 because N_R has been rounded in Step 3 of the algorithm.

Next, we constructed the collections R_0 (implicitly) and R (explicitly), defined over the same alphabet and also 32 symbols long. The detectors were selected by randomly generating strings, and comparing them to the program strings. If more than the specified number (r) of contiguous characters in the same positions matched those in a program string, the detector was rejected. The generate and match procedure continued until the specified number of detectors was generated.

To test for a modified file, a new file was constructed and the detectors were then compared to it. If any of the detectors matched the strings in the new file by more than the specified allowed maximum number of matches, a modification was reported. We used several methods to modify the program file: changing the source code and recompiling (say, to add a loop), changing a single character in the protected file (the minimal change possible), and changing 24 characters at the end of the code segment of the source file. These methods are labeled "Loops," "Single Mutation," and "Data Segment" respectively in Table 3. The last method is intermediate in difficulty and is probably the most realistic from a virus-detection viewpoint. It was the one used to generate the example shown in Table 5.

Method of Infection	Num. of Detectors	P_F	$(P_F)^5$
Loop	2	0.26	0.001
Single Mutation	50	0.62	0.092
Single Mutation	100	0.24	7.96e-4
Data Segment	8	0.18	1.89e-4
Data Segment	16	0.06	7.776e-7
Data Segment	32	0.00	0.00

Table 3: System performance on high-cardinality alphabet (experimental results). Each experiment was run with 37 self strings, each of length 32 characters, and with $r = 1$. Each reported number is based on multiple repetitions of the detection system using a different R_0, but leaving the modification constant. The Loop and Single Mutation experiments were repeated 50 times, Data Segment experiments were repeated 100 times.

The first series of tests involved inserting a short loop into the source code of a program and recompiling the code (called Loop in the table). Although the Data Segment change inserts only 24 additional instructions into the compiled file, the insertion shifts the order of the instructions from the point of insertion forward. This has the effect of altering all of the character strings following the new code. For this type of change, tests run with only two detectors constructed with $r = 1$ (one matching symbol), could detect a change 74% of the time. We conclude that detecting viruses of this form is easy for our algorithm.

It may seem surprising that the algorithm works with $r = 1$. In fact, there are two constraints on the matching process: (1) the detector must have the correct character (out of a possible 104), and (2) the character must be in the correct position. It turns out that these constraints are restrictive enough that we do not need to require any contiguous matching.

The second series of tests created an infected file by changing only one character in the original file (labeled "Single Mutation" in the table). This represents the opposite extreme from the first case. In these tests, we performed two types of experiments, one with 100 detectors and one with 50. Each detector was 32 characters long. Again, r was set to one. In the test runs

collapsed some variants into one symbol, e.g., different versions of certain floating point operations were treated uniformly.

r	N_R	N_{Ro}	P_f
9	2	1435(1150.53)	0.270(0.044)
9	5	3229(1104.72)	0.111(0.074)
9	8	5910(1864.25)	0.010(0.010)
9	10	7274(2580.88)	0.000(0.000)
10	5	182 (71.67)	0.150(0.036)
10	8	315 (126.09)	0.040(0.020)
10	10	382 (111.66)	0.020(0.014)
10	15	598 (161.29)	0.020(0.014)
10	25	996 (211.11)	0.000(0.000)
13	25	54 (7.18)	0.140(0.035)
13	50	86 (8.36)	0.110(0.031)
13	100	170 (12.35)	0.010(0.010)
13	125	205 (0.00)	0.000(0.000)

Table 4: Probability of failing to detect modification to more.com when infected by timid virus. $N_S = 655$. String Length = 32. r is match threshold. N_R is the number of detectors. N_{Ro} is the Initial repertoire size. N_M is the number of non-self strings. $N_M \approx 76$. P_f is the observed probability of failing to detect the virus.

with 50 detectors, the modification was detected 38% of the time. With 100 detectors the modification was detected 76% of the time. Table 3 shows results from all three types of experiments. The rightmost column, labeled $(P_F)^5$, shows the dramatic improvement that is obtained if only five copies of the detection algorithm are present (i.e., five independent sites).

A final set of experiments tested a simple file-infector virus. As an example, consider the TIMID virus, described in [7]. This virus modifies the first five bytes of a COM file and appends 300 bytes of code to the end of the file. Viruses such as these turn out to be extremely easy for our algorithm to detect, for the same reason as the data segment change in the SPARC test. The testing method was as follows:

1. Generate detectors for a standard .com file, supplied with DOS 5.0.
2. Copy the virus into a directory containing the .com file.
3. Execute the virus, causing the original .com file to be infected.
4. Test modified .com file with detectors to see if a modification is detected.

Tests were conducted on three different files, more.com, loadfix.com (not shown), and edit.com (not shown). Table 4 shows the results for more.com, a file with 655 binary self strings, each of length 32. Values shown are the average of 100 trials (numbers in parentheses are the standard deviations). Considering the first four lines of the table, 73% reliability can be attained with only two detectors, and essentially 100% reliability is attained with ten detectors.

Params	Binary Strings	SPARC ($m = 104$)	COM ($m = 2$)	Timid ($m = 2$)
l	32	32	32	32
r	8	1	8	9
N_S	128	37	128	655
P_f	~ 0.0	~ 0.0	~ 0.0	~ 0.0
N_{R_0}	24081	68	1861	6576
N_R	46	8	25	10

Table 5: Typical experimental results on randomly generated binary strings (column 1), strings of SPARC instructions generated from a C program (column 2), .com files (column 3), and a file infected by the Timid virus (column 4). In each case, m, l, r, and N_S are predetermined. In the binary string and .com cases, P_f was fixed and N_{R_0} and N_R were determined experimentally. In the SPARC and Timid cases, N_R was fixed and P_f observed experimentally.

Finally, in order to compare the results from each of the preceding experiments, Table 5 displays typical results from each of the preceding experiments.

The most notable observation about the data in this table is that the algorithm performs much better in practice than in theory. The most obvious explanation for this discrepancy is that real programs are not collections of completely random strings. For example, our analysis assumed that each symbol of the alphabet occurs with equal probability, and this is certainly not the case with the SPARC instruction set. It is fairly straightforward to modify the analysis to account for different occurrence frequencies of symbols. However, there are other ways in which actual programs might deviate from random strings. For example, certain sequences of symbols may occur with some regularity, perhaps related to the particular compiler that was used. We have not yet investigated what these patterns are or how to extend the theory to account for them. A second way in which our experiments deviate from the analysis is in the method used to generate a modified string. The theory assumes that only one random string is added to self, possibly replacing an existing string, but the viruses and modifications we reported (except for single mutation in the SPARC case) all involve larger changes to self.

4 Discussion

The algorithm we have just presented takes its inspiration from the generation of T cells in the immune system. The immune system is capable of recognizing virtually any foreign cell or molecule. To do this, it must distinguish the body's own cells and molecules which are created and circulated internally (estimated to consist of on the order of 10^5 different proteins) from those that are foreign. T cells have receptors on their surface that can detect foreign proteins (called antigens). These receptors are made by a pseudo-random genetic process, and it is highly likely that some receptors will detect self molecules. T cells undergo a censoring process in the thymus, called negative selection, in which T cells that recognize self proteins are destroyed and not allowed to leave the thymus.[3] T cells that do not bind self peptides leave the thymus, and provide the basis for our immune protection against foreign antigens. Our artificial immune system works on similar principles, generating detectors randomly, and eliminating (censoring) the ones that detect self. We refer to the detectors as *antibodies*, even though our model was inspired more by the deletion of self reactive T cells than by the deletion of antibodies.

The algorithm presented here is related to earlier immune-system models based on a universe in which antigens (foreign material) and antibodies (the cells that perform the recognition) are represented by binary strings [2, 11, 4, 3]. The complex chemistry of antibody/antigen recognition is highly simplified in these binary immune systems, being modeled as string matching. These binary models have been used to study several different aspects of the immune system, including its ability to detect common patterns in noisy environments [3], its ability to discover and maintain coverage of diverse pattern classes [10], and its ability to learn effectively, even when not all antibodies are expressed and not all antigens are presented [5]. In the current algorithm, we logically split the self string into equal-size segments to generate valid antibodies (detectors), providing a collection of strings analogous to internal cells and molecules in the body.

The distributed nature of the algorithm was also inspired by the immune system in which each individual generates its own unique set of protective antibodies. This analogy is reflected in the change-detection algorithm because each copy of the detection system generates its own unique valid set of detectors.

[3] Just as our algorithm splits up a self string into smaller substrings, proteins are broken up into smaller subunits, called peptides, before recognition by T cells.

5 Conclusions and Future Directions

We have described a general method for distinguishing self from other in the context of computational systems, and we have illustrated its feasibility as a change-detection method on the problem of computer virus detection. The major limitation appears to be the computational difficulty of generating the initial repertoire. Although this is potentially an advantage in that it protects the antibodies from being modified to conform to a modified form of self, we are currently investigating several possible ways to reduce this complexity. Our current investigations in this direction are based on ideas from immunology, although we believe that it may also be possible to apply more conventional algorithms, especially to take account of regularities in self. It should be noted, however, that any nonrandom method of generating detectors is likely to produce some regularities in the detector set. These regularities might be exploited by a malicious agent, thus compromising the security of the system.

One way to defeat our algorithm would be to design a virus that was composed from a subset of self (presumably in a different order). That is, if one could design a virus that used the same logical segments of the program as our method uses for its checking, we would be unable to detect it. Although we believe that this would be very difficult to do in practice, a slight modification to our method protects against this vulnerability. By simply choosing a different segment length (l) for each site, the number of common substrings available for the virus quickly diminishes. Interestingly, natural immune systems use a similar strategy. Proteins are broken up into a large pseudorandom collection of peptides. Major histocompatibility complex (MHC) molecules bind a subset of these peptides and present them to T cells for recognition. The genes that code for the MHC molecules are highly polymorphic and thus each individual may present a different set of peptides for recognition.

The details of our partial matching rule and the segmentation of self into equal-size segments were arbitrary decisions. We designed our system with these features in order to simplify the mathematical analysis of its behavior. Although a matching rule based on contiguous matching regions makes sense immunologically, there may well be more appropriate rules for a computational environment. An important area of future research is to investigate other matching rules and to revisit the decision to partition self into equal-size segments. To date, we have only studied how the method can be applied to computer virus detection. However, we suspect that it is also applicable to a wide variety of network and operating system problems, and this is an area which we are currently investigating.

Finally, our approach unifies a wide variety of computer and data security problems by treating them as the problem of distinguishing self from other. Negative selection is only one of many mechanisms that the immune system has evolved to distinguish self from other. We are interested in discovering other information-processing methods used by the immune system that can be translated into useful algorithms.

6 Acknowledgments

The authors gratefully acknowledge the Santa Fe Institute for encouraging and supporting the interdisciplinary research that produced the results reported here. Support was provided to Forrest by the National Science Foundation (grant IRI-9157644). David Mathews helped prepare the figures and John McHugh made many helpful suggestions about the manuscript.

References

[1] R. J. De Boer and A. S. Perelson. How diverse should the immune system be? In *Proc. Roy. Soc. London B*, volume 252, pages 171–175, London, 1993.

[2] J. D. Farmer, N. H. Packard, and A. S. Perelson. The immune system, adaptation, and machine learning. In D. Farmer, A. Lapedes, N. Packard, and B. Wendroff, editors, *Evolution, games and learning*, pages 187–204. North–Holland, Amsterdam, 1986. (Reprinted from *Physica*, 22D, 187–204).

[3] S. Forrest, B. Javornik, R. Smith, and A. S. Perelson. Using genetic algorithms to explore pattern recognition in the immune system. *Evolutionary Computation*, 1(3):191–211, 1993.

[4] S. Forrest and A. S. Perelson. Genetic algorithms and the immune system. In H. Schwefel and R. Maenner, editors, *Parallel Problem Solving from Nature*, Berlin, 1991. Springer-Verlag (Lecture Notes in Computer Science).

[5] R. Hightower, S. Forrest, and A. S. Perelson. The evolution of secondary organization in immune system gene libraries. In *Proceedings of the Second European Conference on Artificial Life*, (in press).

[6] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. Technical Report CSD-TR-93-071, Purdue University, Dept. of Cmputer Sciences, Purdue University, West Lafayette, IN 47907-1398, 1993.

[7] M. Ludwig. *The little black book of computer viruses.* American Eagle Publishers, 1991.

[8] J. K. Percus, O. Percus, and A. S. Perelson. Predicting the size of the antibody combining region from consideration of efficient self/non-self discrimination. *Proceedings of the National Academy of Science*, 90:1691–1695, 1993.

[9] J. K. Percus, O. E. Percus, and A.S. Perelson. Probability of self-nonself discrimination. In A. S. Perelson and G. Weisbuch, editors, *Theoretical and Experimental Insights into Immunology*, NY, in press. Springer-Verlag.

[10] R. Smith, S. Forrest, and A. S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2):127–149, 1993.

[11] I. Stadnyk. Schema recombination in pattern recognition problems. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 27–35, Hillsdale, NJ, 1992. Lawrence Erlbaum Associates.

[12] J. V. Uspensky. *Introduction to Mathematical Probability.* McGraw-Hill Book Co., NY, N.Y., 1937. pp. 77-79.