A Distributed Approach to Anomaly Detection

Patrik D'haeseleer
Dept. of Computer Science
University of New Mexico
Albuquerque, NM, 87131
patrik@cs.unm.edu

Stephanie Forrest
Dept. of Computer Science
University of New Mexico
Albuquerque, NM, 87131
forrest@cs.unm.edu

Paul Helman
Dept. of Computer Science
University of New Mexico
Albuquerque, NM, 87131
helman@cs.unm.edu

DRAFT - Saturday, August 30, 1997

Abstract:

The natural immune system has evolved many interesting mechanisms to solve the problem of self-nonself discrimination. An anomaly detection system based upon principles derived from the immune system was introduced in [Forr94]. Its main advantages are that it is distributable, local, and tunable. This paper provides an overview of the theoretical, algorithmic and practical developments extending the original proposal. In particular, we present information theoretic results on the detection method, show the possibility of strings that cannot be detected for a given combination of self set and matching rule, present efficient algorithms to generate the detector set, and provide rules of thumb for setting the parameters to apply this method to a real data set.

1. Introduction

1.1 Motivations from Immunology

Much of computer security can be viewed as the problem of distinguishing *self* (legitimate users, authorized actions, original source code, uncorrupted data, etc.) from *nonself* (intruders, computer viruses, spoofing, Trojan horses, etc.). Nature, more specifically the natural immune system, has been solving a similar problem for hundreds of millions of years, using methods quite different from those typically used to protect computers.

The body's immune system is comprised of a wide range of mechanisms, the intricacies of which are still being unraveled (see [Jane97] for an overview of immunology). These mechanisms are often categorized as either "specific" or "nonspecific" depending on whether they provide specialized protection against a known type of intrusion (such as antibodies reacting against one specific kind of virus), or a more general protection against anything coming in from the outside (such as the skin, or inflammatory responses against cell damage). Likewise, computer security measures can be divided into specific (virus checking with signatures, security analysis tools [Farm93], etc.) and nonspecific (good code hygiene, firewalls, encryption, etc.). However, most of the nonspecific measures are passive, in the sense that they only *prevent* nonself from intruding on the system but do not detect intrusions in progress, whereas the specific measures usually have a difficult time keeping up with the new attacks continually being developed or discovered by malicious agents. Any single protection mechanism is likely to have its own vulnerabilities, making it hard to find and patch every security hole in a large computer system. It is our belief that a comprehensive multi-faceted approach is necessary, in which, similar to natural immune systems, both specific and nonspecific protection mechanisms play an integrated role.

In earlier work by Forrest et al. [Forr94], an anomaly-detection algorithm was introduced based on one of the mechanisms used by the immune system: The generation of T lymphocytes in the thymus (see [Dipa97] for a survey of other immunity-based systems for engineering applications). T lymphocytes, or T cells, are one of the many kinds of specialized cells in the immune system. The surface of a lymphocyte is covered with receptors that can bind to antigens (foreign proteins). Each lymphocyte has one specific kind of receptor, and each receptor binds to a small group of structurally related antigens. The receptors on T cells are constructed by a pseudo-random process. As the T cells mature in the thymus, they undergo a censoring process called *negative selection*, in which those T cells that bind self proteins are destroyed [Kapp87, Paul89]. After censoring, T cells that do not bind self are released to the rest of the body and provide the basis for our immune protection against foreign antigens. This mechanism in the immune system is very robust, because of its distributed nature, and remarkably efficient. The repertoire of T cells produced in this way seems to cover most of the antigen space (estimated at around 10^{16} different possible foreign molecules), while tolerating an estimated 10^{6} different body proteins as “self” [Inma78]. In immunology, T cells are counted under the specific immune response, because each individual T cell responds to a specific antigen. However, the generation of a repertoire of randomly generated T cells covering almost all of the antigen space can be viewed as a nonspecific mechanism.

The algorithm presented in [Forr94] works in a similar way. Detectors are generated to match anything that does not belong to self (for this reason, it is also sometimes called the “negative selection method”). In operation, these detectors are used in much the same way as signatures are used to scan for specific computer viruses. However, a virus scan program only checks for a small number of known signatures and needs to be updated continuously with signatures for newly emerging viruses. Our method relies on having a large enough set of random detectors that virtually all nonself objects will be detected. Thus, the negative selection method can be classified as an active, yet nonspecific, protection mechanism. As a nonspecific method, it might not always be as efficient or effective as some of the knowledge-intensive special-purpose mechanisms for detecting specific kinds of changes or known attacks. Its strength, however, is its generality; it potentially could be applied in many settings as a safety net to catch anomalies that might otherwise go undetected.

1.2. A Distributable, Local and Tunable Anomaly Detection Method

In the immune system, the self and nonself objects, as well as the detectors, are peptides, short strings of amino acids. It is not entirely clear at what level of granularity we should place the corresponding elements in a computer system. For instance, the self objects could be the users logged on a system, with each detector looking at some random part of their audit trail. Our emphasis, however, has been on a lower level of granularity, where both the detectors and the data to be protected consist of fixed length strings over a given alphabet of symbols (usually binary). However, the method does not depend on this representation and can be applied to any collection of objects. In immunology, binding between antigen and T-cell receptors depends on interactions between the complementary shapes of antigen and receptor molecules (the so-called “lock and key” model). The match between antigen and receptor need not be exact, so each receptor can bind a small range of similar antigens and vice versa. We simulate this by using a “matching rule” which, given a detector and an unknown string, decides whether the detector matches the string. This decision can be based on any of a number of string matching algorithms, such as edit distance, Hamming distance etc.

The choice of matching rule is crucial to the performance of this method. To date, our research has mainly used the “r-contiguous-bits” matching rule, taken from theoretical immunology [Perc92]. With this rule, two strings are said to match if they are identical in at least *r* contiguous positions (where *r* is a parameter of the matching rule). For instance, for *r*=5, the two strings on the left match under this rule, whereas the two strings on the right do not match:

```
1000111101011010        1010010001000110
0101011101101000        0101001011100010
```

Although this is the matching rule used throughout most of the paper, it should be stressed that this choice of rule was fairly arbitrary. It does lend itself quite well to analysis and serves mainly as an illustration of the feasibility of the method. In [Forr94] for example, it was used successfully to do some simple virus detection. For use in a real security system, other matching rules based on checksums or even cryptographic one-way hash functions might be more appropriate.

Similar to the generation of T cells in the thymus, detector strings can be generated at random. The ones that match any of the self strings to be protected (according to the specified matching rule) are eliminated. The ones that fail to match any of the self strings form the detector set (or “repertoire”) *R*. This process is repeated until we have enough detectors to ensure the desired protection level (defined in terms of the probability of success in detecting any nonself strings with at least one detector). This generate-and-test algorithm, illustrated in Figure 1, works for any matching rule, however it requires sampling a number of candidate detectors that is exponential in the size of the self set [DeBo93]. Similarly, in the thymus, 95-99% of all immature T cells get eliminated, an apparent waste of resources. In a highly parallel system like the body, however, this does not pose a problem. In a computer system we would prefer to be able to generate valid detectors directly,

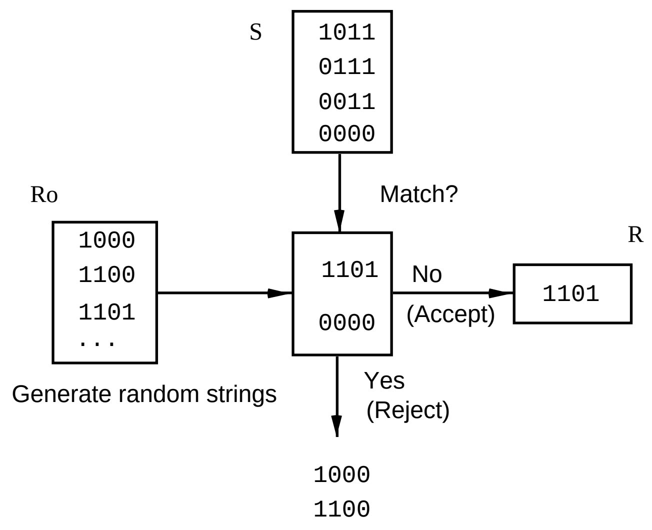

Figure 1: Constructing a set of detectors *R* for a set of self strings *S*, using generate-and-test. The matching rule used here is “r-contiguous-bits” with *r*=2. The strings in R_0 are generated at random. The ones that match any of the strings in *S* are rejected.

without having to guess randomly. We will present some much more efficient algorithms to generate detectors in Section 3.

Figure 2 shows the relevant sets of strings and how they relate to each other. String space U and detector space U_d are drawn separately for clarity, even though in this paper $U = U_d$. We introduce the following notation:

Matches(P) = Q (read: "P matches Q") iff Q contains all the strings matched by any detector in P.

MatchedBy(Q) = P (read: "Q is matched by P") iff P contains all the detectors matching any string in Q.

Matching probability P_m : probability that a randomly chosen string and detector match according to the matching rule.

Failure probability P_f : probability that a single random nonself string will not be matched by any of the detectors in R.[1]

We write N_X for the size (cardinality) of a set X. In particular, N_S is the size of the self set and N_R is the detector set size.

Given a set of self strings $S \subset U$, the goal is to find a detector set $R \subset U_d$ that matches as many of the nonself strings in N as possible ($N = U$-S), without matching any of the self strings in S. The detectors in R are chosen from the set of candidate detectors C, consisting of all strings that do not match any string in S (i.e., $C = U_d$ - *MatchedBy*(S)). If we were to include all candidate detector strings in R, we would be able to detect all nonself strings in set N', which may or may not be equal to N (see the discussion of *holes* in Subsection 2.3). Generally, only a small subset of candidate detectors will be included in R, allowing us to detect all nonself strings in D (with $D \subset N' \subset N$). Failure probability $P_f = N_D/N_N$, or $P_f \approx N_D/N_U$ if $N_S << N_U$.

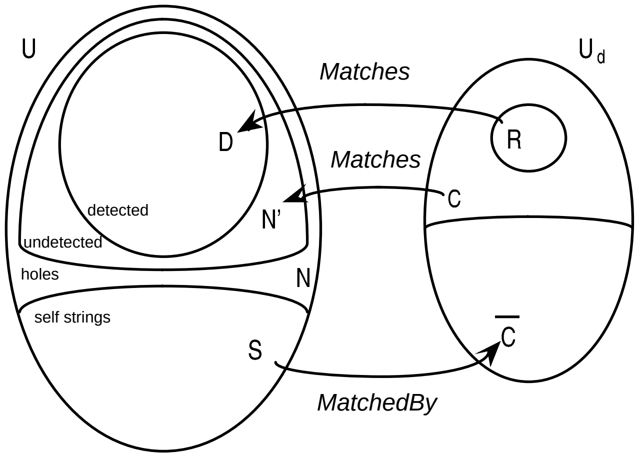

Figure 2: Relationships between the subsets of string space U and detector space U_d. String space is partitioned into self strings S and nonself strings N. Candidate detectors (C) are those that do not match any string in S. The detector set R is a subset of the candidate detectors. Nonself strings can be further subdivided in detectable nonself strings (N') and holes ($H = N - N'$, not labeled). Detectable nonself strings can be detected (D) or not ($F = N' - D$, not labeled), depending on the choice of R.

This method (as is true of most other anomaly detection methods) relies on having an accurate and stable description of self. In this sense it could be considered as static (although some of the more dynamic mechanisms in the immune system can be used to provide a form of online learning). Earlier work emphasized the change detection properties of the algorithm [Dhae95b, Dhae96a, Dhae96b], but its application is not limited to purely static data sets such as data files. Thus it is properly classified as an *anomaly* detection rather than a change detection algorithm. Our current research is applying it to monitoring patterns of activity. For instance, the pattern of "normal" sequences of system calls for certain running processes has been shown to be quite stable [Forr95] and can be used to detect anomalous behavior of these processes. Some work has also been done on applying this method to anomaly detection in manufacturing processes [Dipa95].

The negative selection method has a number of features which distinguish it from most other methods:

- No prior knowledge of intrusions is necessary. This is in contrast with signature-based virus scanners.
- Detection is probabilistic but tunable. That is, in general we will not be generating a *complete* repertoire of detectors (i.e. a set of detectors that covers all possible nonself strings). Instead we will content ourselves with matching all but a small fraction P_f of nonself strings, in exchange for a smaller set of detectors. The desired probability of success in detecting random nonself strings with a single set of detectors can be weighed against the expected number of nonself strings per anomaly and the cost of generating, storing and checking detectors.
- The detection scheme is inherently distributable. Small sections of the protected object can be checked separately, different sites can have independently created detector sets for the same object, and the detectors in the detector set can themselves be run independently. No communication between detectors or detector sets is needed until a change is detected. This makes it a promising tool for computer security in networked or distributed computer environments, or in a scheme with autonomous agents such as the one presented in [Cros95].
- Detection is local. Classical change detection methods (such as checksum methods) typically require the entire data set to be checked at once. Our method allows small sections of the data to be checked, and when a detector does find an anomaly it can be localized to the string that the detector is checking.
- The set of detectors at each site can be unique. This means that if one site should be compromised, others would still be protected. For independent detector sets, the *system-wide* failure probability decreases exponentially with the number of sites protected. This may allow us to choose a fairly high P_f (and thus small detector sets) for the individual systems.
- The set of self strings and the detector set are mutually protective, meaning that the detector set protects the self set against change and vice versa. Also, detection is local: we can pinpoint the individual string (or detector) which has been changed.

An often heard comment when describing this anomaly detection method is “Why don’t you just use checksums”? Firstly, it should be noted that checksum methods are really a special case of the method described here, where the self set consists of a single object (the file to be protected), the detector set contains a single detector (the checksum), and the matching rule consists of calculating the checksum over the file and comparing it with the stored checksum. Therefore, given appropriate parameter choice, out method could have all the same advantages of normal checksum methods. In that sense, it could be considered as a generalized checksum method for protection of N_S files or file sections using N_R checksums. This allows the detection to be distributed and localized. An added benefit is its tunability, which allows it to be used as a anomaly detection method in a dynamic environment, where sensitivity to single-bit changes is not desirable.

[1] Of course, the probability that an anomaly goes undetected will tend to be much smaller if a single anomaly results in several nonself strings.

One of the difficulties in implementing this method is that there are many degrees of freedom in choosing the parameters. Section 2 gives an overview of the theoretical analysis, aimed towards deriving some theoretical bounds for these parameters and a greater understanding of the properties of this anomaly detection method. Information theoretic approaches have been very useful in this regard. In Section 3, we give an overview of the algorithms that have been developed to generate detector sets efficiently. We give the time and space complexity of these algorithms, as well as some formulas and parameter bounds derived from them. We also present a similar algorithm that can be used to count the exact number of holes (undetectable nonself strings). Section 4 compares results obtained with the different algorithms and formulas, both on randomly generated files and on a real binary file. We also touch on some of the practical issues, including guidelines for applying the method, how to choose the parameters, and the formulas involved. Section 5 presents conclusions and outlines areas for further study.

2. Theoretical Analysis

In subsection 2.1, we derive an estimate of the amount of information that is lost by splitting a data stream into unordered strings, and show how this estimate can be used to guide the choice of string length. Subsection 2.2 derives two separate lower bounds on the size of the detector set, one based on the geometry of the string space, the other based on information-theoretic arguments. We show how these bounds are related, and that they can be used to derive an upper bound for the matching probability P_m. The principle of holes (undetectable nonself strings) is illustrated in subsection 2.3, along with a proof of their existence for a large class of matching rules. The influence of holes on the achievable failure rate is discussed, along with guidelines on how to avoid holes. A preliminary version of the material in this section appeared in [Dhae95b].

2.1. Splitting data into strings

The anomaly detection method described above works on an unordered set of strings. In order to apply the method to a real data set (such as a file on disk, an audit trail, the behavior of a process, a sequence of system calls of a running program, etc.), we will typically have to preprocess the data by splitting it into strings of length *l* over an alphabet of size *m*. This destroys some of the internal structure of the data that could be used to make the detection proceed more efficiently. Because we want to use as much information as possible about Self, we would like to minimize this information loss. In the following analysis we assume a binary alphabet, but extension to larger alphabets is straightforward.

The information lost by splitting up a stream $\hat{S}$ into a set of unordered strings *S* is equal to the amount of information (in bits) we would need to be given in order to reconstruct the original stream from the unordered set, or alternatively the amount of information we would gain by being given the original stream $\hat{S}$ if we already knew *S*. This can be denoted by $H(\hat{S} \mid S)$ (i.e. the conditional entropy of $\hat{S}$ given *S*). In information theory, this quantity is often called the Doubt, since it is the amount of information about $\hat{S}$ that is missing in the encoding *S*. For a given string length *l*, a stream $\hat{S}$ of L_S bits will be split up into a set *S* of N_S *l*-bit self strings.[2] Usually, not all of these strings will be unique, so *S* is really a multiset. Say we get *k* unique *l*-bit strings, where each string s_i appears N_i times:

$$\sum_{i=1..k} N_i = N_S \ .$$

Given such a set of N_S strings, the original stream could have been one of

$$\binom{N_S}{N_1, N_2 \cdots N_k} = \frac{N_S\,!}{N_1\,! \cdot N_2\,! \cdots N_k\,!}$$

possible rearrangements of these strings. If we assume that each possible rearrangement is equally likely (i.e. we have no prior information about the nature of the data stream), then the amount of information lost is:

$$\Delta I_1 = H\left(\hat{S} \middle| S\right) = \log_2 \binom{N_S}{N_1, N_2 \cdots N_k} \text{ bits.}$$

From Stirling's approximation for the factorial function, or by a reasoning similar to the one in [Cove91], it can be shown that:

$$\binom{N_S}{N_1, N_2 \cdots N_k} \leq 2^{N_S \cdot H\left(N_i/N_S\right)} \qquad (1)$$

where $H(N_i/N_S)$ stands for the entropy associated with the relative frequencies of occurrence of the strings in *S*. Taking the logarithm base 2, this becomes:

$$\log_2 \binom{N_S}{N_1, N_2 \cdots N_k} \ \leq \ N_S \cdot H\left(\frac{N_i}{N_S}\right)$$

or, in "big-$\mathcal{O}$" notation:

$$\log_2 \binom{N_S}{N_1, N_2 \cdots N_k} = \mathcal{O}\left(N_S \cdot H\left(N_i/N_S\right)\right) .$$

Therefore, as a bound on the amount of information lost, we get:

$$\Delta I_1 = \mathcal{O}\left(N_S \cdot H\left(N_i/N_S\right)\right) = \mathcal{O}\left(L_S \cdot H\left(N_i/N_S\right)/l\right) . \qquad (2)$$

This analysis suggests that if we measure the average entropy per bit of the strings in *S* for different string lengths *l*, we could minimize the information loss by choosing the value of *l* for which the entropy per bit is minimal. In general, this will only be one of many factors that need to be considered in the choice of *l*. Although a small *l* tends to reduce the entropy of the strings because it reduces the size of the total string space, the entropy *per bit* will tend to go up with decreasing string length. On the other hand, increasing *l* beyond the point where most of the strings in *S* are unique will reduce entropy

[2] The alternative would be to generate the strings using a sliding window of length *l*. This can result in significantly less information loss but is harder to analyze.

because it reduces the number of strings in *S*, but choosing a large *l* will make generation of detectors difficult in terms of space and time requirements.

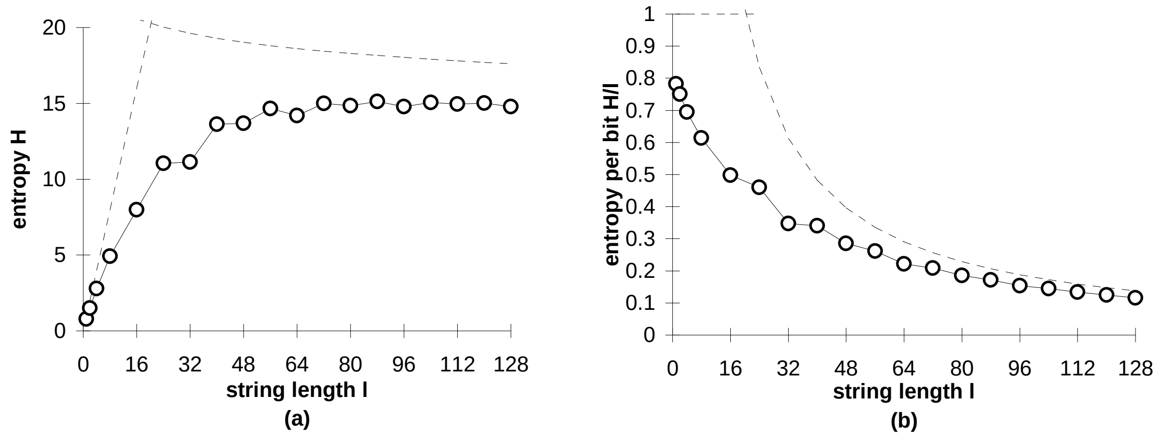

Figure 3: Entropy of the set of strings obtained by splitting an emacs binary into *l*-bit strings. The dotted lines indicate the envelope within which *H* must fall, delimited by H/l=1 and $H=log_2(N_S)$. (a) shows the entropy $H(N_i/N_S)$ of the resulting set of strings. (b) shows the entropy per bit of these strings, which is proportional to ΔI_1.

Figure 3 illustrates how $H(N_i/N_S)$ changes with respect to *l* for a real data file. The data chosen was an emacs binary (GNU emacs v19.25.2 SGI binary, 3.2MB). The file was split up into strings of length l=1 (single bits), 2, 4, and multiples of 8 (i.e. at byte boundaries) up to 16 bytes long. For each value of *l*, the entropy of the resulting set of strings was calculated. Figure 3(a) shows clearly that there are minima in the entropy at multiples of 4 bytes, corresponding to the 32-bit RISC instruction used to compile the binary. The values plotted here at 8-bit intervals are themselves local minima as well: for intermediate values of *l* the entropy would be much higher, because this corresponds to cutting up the data at non-byte boundaries, ignoring the natural byte structure present in the data. The entropy per bit plotted in Figure 3(b) is proportional to ΔI_1 (apart from a constant factor L_S). This graph is uniformly decreasing, so in order to minimize information loss for this data we would want to choose *l* as high as our other constraints allow, with a preference for multiples of 4 bytes. Note that most of the gain occurs at lower string lengths, so increasing string length beyond 64 bits (8 bytes) gives little extra improvement.

This calculation of the information lost by splitting up the data at *l*-bit boundaries applies not only to the anomaly detection method described here. This is a general result, applicable to any method that transforms a stream of data into an unordered set of strings. In particular, it would also hold for methods that take into account the frequency of occurrence of the self strings. However, because our method ignores the frequency of occurrence of each unique *l*-bit string s_i, this constitutes a second source of information loss. If we assume N_S is known, then if we are given the set of *k* unique strings in S $(s_1, s_2, \dots, s_k)$, there are

$$\binom{N_S - 1}{k - 1}$$

possible ways of assigning the values for N_i (such that they sum up to N_S), and thus of reconstructing the unordered multiset S. If each of those assignments is equally likely (again, assuming we have no prior knowledge of the contents of the data stream), the amount of information (in bits) lost by ignoring the frequencies of the strings is given by:

$$\begin{aligned} \Delta I_2 &= H(S|s_i) \\ &= \log_2 \binom{N_S - 1}{k - 1} \\ &\leq (N_S - 1) \cdot H\left(\frac{k-1}{N_S - 1}\right) \qquad \text{(see [3])} \\ &< N_S \ . \end{aligned}$$

In general, ΔI_2 will be small compared to ΔI_1 and can be ignored. If we do want to minimize ΔI_2, minimizing N_S by choosing a large string length l would be the obvious way to do it. This would tend to reduce the number of duplicate strings and thus the amount of information lost by ignoring duplicates, although this may in some cases increase ΔI_1 because of an increase in $H(N_i/N_S)$.

2.2. Detector set size

The number of detectors needed to achieve a specified failure rate P_f is an important factor in comparing different parameter settings, because this affects the time needed to generate the detector set, the storage space needed for the detector, the amount of nonself space covered by each detector, etc. It is possible to derive theoretical lower bounds on the number of detectors needed.

A first lower bound for N_R can be derived from the average matching probability P_m (probability that a randomly chosen string and detector match according to the specified matching rule) [Pere79]. At best, we can spread the detectors apart such that no two detectors match the same nonself string. The amount of string space covered by N_R detectors is then $P_m N_R N_U$. This needs to be at least as big as $(1-P_f)N_U$, in order to cover enough nonself strings for a failure probability of P_f (ignoring N_S with respect to N_U), so:

$$N_R \geq (1 - P_f) / P_m \ . \qquad (3)$$

A second, more general, lower bound can be derived based on the amount of information about S that can be stored in R. A set of detectors R constructed for a given set of self strings S can be viewed as a message encoding information about that self set. Given a "perfect" detector set R', that is, a set of detector strings that exactly recognizes all nonself strings, it would be possible to reconstruct the original self set S (by checking for each string in the string space whether it is detected by any detector in R'). Therefore, this set R' would have to contain at least the same information as the original set S. By calculating the information content of the original file of self strings it is possible to get an absolute lower bound on the information content, and thus on the size, of the detector set. For a random self set, the information content will be of the same order as its size in bits. This means that the perfect detector set *R´* would have to be of approximately the same size (in bits) as the self set *S*. For large self sets this will generally not be acceptable.

We can alleviate this problem by allowing a certain amount of error in the self-nonself detection performed by the detectors. For the purpose of this analysis, we will assume that a fraction P_f of the nonself strings are not matched by the detectors. In other words, we allow at most $P_f(N_U\text{-}N_S)$ (where *U* is the total string space, and $N_U=|U|$) false negatives (nonself strings not detected) but no false positives (no self strings classified as nonself)[3]. Not requiring an exact encoding of the self set *S* means that we can get away with a much smaller information content, and thus, a much smaller size for the detector set.

If we choose the self strings from the set of all possible strings *U*, then there are

$$M = \binom{N_U}{N_S}$$

possible self sets of size N_S. If we assume (i) that the set *S* of self strings is independent, i.e. that each string is chosen at random out of the set of all possible strings, then each of those sets is equally likely, so the average information content (i.e. entropy) of a self set of size N_S is:

$$H(S) = \log_2 \binom{N_U}{N_S}.$$

Given a detector set *R* constructed for this self set *S* we could try to reconstruct *S*. Since *R* has been constructed to allow a certain amount P_f of false negatives, we won't be able to reconstruct *S* exactly from *R*. Instead we would get a set S' consisting of the N_S original self strings plus $P_f(N_U\text{-}N_S)$ unmatched nonself strings.

Assumption (i) says that the self strings are independent of each other. We will make a second assumption (ii) that given *S´* we have no knowledge of which subset of N_S strings constitutes the original set *S*. The amount of information about *S* that is missing in the encoding *R* is then $H(S|R)$ (i.e. the amount of information we would gain by being given *S* if we already knew *R*). Assumptions (i) and (ii) tell us that any subset of size N_S, taken from the set S' of size $N_S + P_f(N_U\text{-}N_S)$, is equally likely to be the original self set *S*, so:

$$H(S|R) = \log_2 \binom{N_S + P_f\left(N_U - N_S\right)}{N_S}.$$

The difference between *H(S)* and $H(S|R)$ (i.e. the information in S, minus the information about *S* that is missing in *R*) represents the amount of information about *S* that is preserved in the encoding *R* and is called the Mutual Information of *S* and *R* (denoted by $I(S;R)$):

[3] A similar analysis can be made for the general case, with both false positives and false negatives. See [Dhae97].

$$
\begin{aligned}
I(S;R) &= H(S) - H(S|R) \\
&= \log_2 \binom{N_U}{N_S} - \log_2 \binom{N_S + P_f(N_U - N_S)}{N_S} \\
&= \log_2 \left[\binom{N_U}{N_S} \Big/ \binom{N_S + P_f(N_U - N_S)}{N_S} \right] \\
&= \log_2 \left[\frac{N_U!}{N_S! \cdot (N_U - N_S)!} \Big/ \frac{(N_S + P_f(N_U - N_S))!}{N_S! \cdot (P_f(N_U - N_S))!} \right] \\
&= \log_2 \left[\frac{N_U}{N_S + P_f(N_U - N_S)} \cdot \frac{N_U - 1}{N_S + P_f(N_U - N_S) - 1} \cdot \ldots \cdot \frac{N_U - N_S + 1}{N_S + P_f(N_U - N_S) - N_S + 1} \right]
\end{aligned}
$$

Using a third and last assumption, (iii): $P_f N_U >> N_S$ and thus also $N_U >> N_S$, we can simplify this further (the validity and consequences of these three assumptions will be discussed later):

$$
\begin{aligned}
I(S;R) &\approx \log_2 \left[\frac{N_U}{P_f N_U} \cdot \frac{N_U - 1}{P_f N_U - 1} \cdots \frac{N_U - N_S + 1}{P_f N_U - N_S + 1} \right] \\
&\approx \log_2 \left[\left(1/P_f \right)^{N_S} \right] \\
&\approx N_S \cdot \log_2 \left(1/P_f \right).
\end{aligned}
$$

(Similar results can be obtained using (1)). Since R needs to contain at least this much information, this is a lower bound on the size of the detector set, expressed in bits. If the detectors are strings of length l in an alphabet of size m, the required number of detectors N_R is given by:

$$
N_R \geq \frac{N_S \cdot \log_2 \left(1/P_f \right)}{l \cdot \log_2 (m)} . \qquad (4)
$$

Table 1 illustrates this theoretical lower bound for N_R as a function of string length l for a data stream of one million bits. Note that for a tenfold decrease in failure rate we only need a twofold increase in the number of detectors.

l	N_S	N_R ($P_f = 0.1$)	N_R ($P_f = 0.01$)
16	62500	12977	25953
20	50000	8305	16610
24	41667	5768	11535
28	35715	4238	8474
32	31250	3245	6489
36	27778	2564	5127
40	25000	2077	4152

Table 1: lower bounds on N_R for a 1Mbit long data set, for different string lengths.

If the self strings are not independent of each other (as is often the case in reality), this will affect both assumptions (i) and (ii). Assumption (i) (self strings are independent) led us directly to the formula for $H(S)$ because we were allowed to assume that all self sets were equally likely. In reality this will not be the case. We can expect real data to show some structure, which may be reflected in a greater degree of similarity between the self strings than in sets of totally random strings. Our experiments so far seem to confirm that we can use significantly smaller detector sets for real data. Also, if we have high-level knowledge about the nature of the ensemble of "normal" self sets, we may be able to use that information to achieve a smaller encoding.

It is unclear what effect non-independent self strings would have on assumption (ii). Clearly, if the N_S self strings in S' share some characteristics that make them stand out from the $P_f(N_U\text{-}N_S)$ undetected nonself strings, we have some information about which subset of N_S strings is more likely to be the original self file S and therefore $H(S|R)$ can be smaller. However, the character of the undetected nonself strings will largely depend on the matching rule and repertoire construction algorithm chosen. With our current matching rules and algorithms, we would expect the undetected nonself strings to lie approximately in the same region of space as the self strings. This may mean that assumption (ii) would still hold. Of course, in practice it is recommended to check the real failure probability achieved with the generated detector set, by sampling nonself strings or expected intrusion strings, if the distribution of these is known.

Assumption (iii) states that $P_f N_U >> N_S$. In general it will be reasonable to assume at least $N_U >> N_S$ (otherwise almost all possible detector strings would match at least one of the self strings, and most of the nonself strings would in fact be undetectable holes). For failure rates that are not excessively small, assumption (iii) will usually still hold. For instance, with P_f=0.1, every single nonself string has 10% chance of escaping detection by the detectors. If an intrusion in the self file consists of one single changed string, 10% may be quite a large error rate. However, intrusions will generally consist of several different nonself strings and it is sufficient to catch only one of these to sound the alarm. If we have to use an excessively small value for P_f, our approximation may no longer hold.

Although the above analysis was performed with the anomaly detection algorithm in mind (and bit strings for detectors and self strings), it is clear that this approach is more general. In fact, the result above holds for any possible classification scheme for a set S with zero false positives and P_f false negatives. Examples include: Matching rules with a variable radius for each detector, matching rules based on Hamming distance, symbolic representations for self objects and detectors, other self-nonself detection methods (such as k-nearest-neighbors, Neural Networks, etc.), and even the natural immune system. In particular, for checksum methods, (4) reduces trivially to $P_f = 2^{-l}$, where l is the length of the checksum.

The lower bounds given by (3) and (4) use different lines of reasoning and appear to have little in common. Whereas (3) depends mainly on the matching rule (via the matching probability P_m), (4) is a property of the self string space and desired success rate. Depending on the parameters chosen, (3) can be smaller or larger than (4). If (3) > (4), we might be able to construct a detector set which in principle contains enough bits of information to distinguish between S and N up to an error rate P_f, but which can not cover enough of the string space because each detector matches too few strings. On the other hand, if (4) > (3) this implies that there might exist a set of detectors that are spread apart in such a way that they cover all but a fraction P_f of N, but which does not really contain enough bits of information to distinguish between S and N with this accuracy. Because such a detector set clearly cannot exist, it must be impossible to create such a small detector set that still covers enough of the string space. In other words, given this matching probability it must be impossible to avoid overlapping detector regions, so the optimal coverage assumed in (3) cannot be obtained.

If we are given a specific string space, set of self strings and desired success rate, lower bound (4) is fixed. We then need to choose the matching rule to achieve this success rate in the most efficient way. Increasing the matching probability P_m allows each detector to cover a larger part of the nonself space and results in a smaller detector set, as indicated by the bound given by (3). However, the size of the detector set cannot be lowered beyond the lower bound (4), so increasing the matching probability even further only causes more overlap between detectors, making it harder to detect all nonself strings without also detecting some self strings (see Sections 2.3 and 4.1). Therefore, (3) ≥ (4) becomes an upper bound on the useful matching probability P_m:

$$\left(1-P_f\right)\Big/P_m \geq \frac{N_S \cdot \log_2\left(1/P_f\right)}{l \cdot \log_2(m)}$$

or

$$P_m \leq \frac{l \cdot \log_2(m) \cdot \left(1-P_f\right)}{N_S \cdot \log_2\left(1/P_f\right)} \qquad (5)$$

2.3. The existence of holes

The bounds on detector set size seem to indicate that we merely have to choose a large enough detector set in order to achieve any desired failure probability P_f. However, generally speaking, there will be a lower bound on the achievable failure probability as well. This depends largely on the specific matching rule used. For the “r-contiguous-bits” matching rule (two l-bit strings match each other if they are identical in at least r contiguous positions), we have developed efficient detector generating algorithms [Helm94, Dhae95a], making it possible to generate a *complete* detector repertoire, in the sense that it covers all nonself strings that *can* be covered. As shown in [Dhae96a], it turns out that for the r-contiguous bits matching rule there may be some nonself strings, called “*holes*,” for which it is impossible to generate valid detectors. If S contains two strings s_1 and s_2 that match each other over at least (r-1) contiguous bits, they may induce two other strings h_1 and h_2 that cannot be detected because any candidate detector would also match either s_1 or s_2, as shown below for two 16-bit self strings, with r = 7.

$$\begin{array}{l} s_1\text{: } 0000\,\mathbf{010101}\,000000 \\ s_2\text{: } 1111\,\mathbf{010101}\,111111 \end{array} \Rightarrow \begin{array}{l} h_1\text{: } 0000\,\mathbf{010101}\,111111 \\ h_2\text{: } 1111\,\mathbf{010101}\,000000 \end{array}$$

A similar argument shows that we also can have holes using a Hamming distance matching rule (where two strings match if their Hamming distance is less than or equal to a fixed radius r). Figure 4 shows that a similar situation occurs when self strings are represented as points in a Cartesian plane and a match between a string and a detector is said to occur if the Euclidean distance between the corresponding points is smaller than a certain radius r.

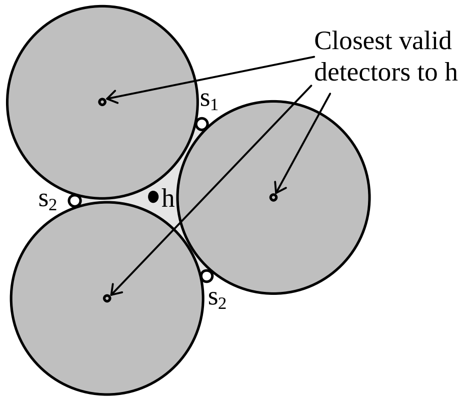

Figure 4: if self strings are presented as points in a plane and the matching rule is Euclidean distance, three self strings s_1, s_2, s_3 may induce a hole h for which no valid detector can be generated.

We show that all practical matching rules with a constant matching probability (i.e. each detector matches equally many strings and vice versa) can exhibit holes even with a non-trivial set of self strings. Let U be the set of all strings, S the set of self strings to be protected, *Matches*(d) the strings detector d matches, and *MatchedBy*(s) the detectors string s is matched by, then $h \in U$ is a *hole* iff $\forall d \in$ *MatchedBy*(s), $\exists s \in S$: $s \in$ *Matches*(d). Given a string h and a matching rule M with constant matching probability P_m, it suffices to show that we can construct a non-trivial self set S such that h is a hole. Starting with the empty set, for each detector d matching h, we pick one string s_d matched by d and add this string to S. Then, by construction, for each possible detector for h there exists a self string matching that detector. The set S constructed in this way will be non-trivial if it does not need to contain a significant portion of the total string space U in order for h to be a hole. An example of a trivial set S inducing holes would be one where the number of nonself strings is smaller than the number of strings matched by any detector. If we are using a matching rule with a constant matching probability P_m, a self set of size N_S = |*MatchedBy*(h)| = $P_m N_U$ always suffices to induce a hole. Of course, we may already get holes for smaller self sets than that. As we have shown earlier, just two self strings matching over r-1 contiguous bits are sufficient for the r-contiguous-bits matching rule.

The existence of holes imposes a lower bound on the failure probability P_f we can achieve with a given detection method, because it will always fail to detect holes (however, we will present some techniques below that enable us to get around this limitation). If we calculate the required number of detectors to achieve a certain acceptable P_f without taking holes into account, the real P_f achieved with this detector set may be substantially higher than expected. Further, the failure probability associated with the holes themselves does not improve by distributing the algorithm if we use the same matching rule at all the sites. Although the detector set on each machine may be generated independently from the other machines, the same holes occur on all machines. Therefore, it might be useful to try to spread the self strings apart as much as possible, reducing the number of holes between them. This can be achieved by randomizing each self string with a hash function. We may also choose to use detectors with a greater specificity (smaller radius in Figure 4), at the expense of a larger detector set.

Another option is to try to eliminate the holes altogether, either locally or in a distributed fashion. Locally, this can be achieved by using a matching rule that does not exhibit holes. For instance, with the r-contiguous-bits rule, we can choose a different value of r for each detector, between r=1 (detector matches almost the entire string space) and $r=l$ (detector only matches a single string: itself), such that potential holes can be filled by detectors with high specificity. The value of r would have to be stored with each detector. This method also has the advantage that very large sections of string space could be covered with only a few detectors. Note that although this seems to be a much more efficient way to cover the nonself string space, the size of the detector set still has to conform to equation (4). In a distributed setting, with several sites running their own detector sets, we can choose a different matching rule (same rule with different parameters or different rule altogether) for each machine. Each will have a different set of holes which hopefully will be covered by some other machines. For example, each site could perform a different permutation on the bits in the bit strings before applying the r-contiguous-bits matching rule, or all strings could be encrypted with a site-dependent key. The combination of the permutation (or encryption) plus the r-contiguous-bits rule generates an entire family of matching rule, one for each different permutation (or encryption key).

Note that the existence of holes can be a blessing instead of a curse. Since holes are generated by interaction between a number of self strings, holes will tend to be “close to” these self strings in strings space (according to the matching rule used). In some applications, we might not wish to detect strings that are that close to self. For instance, if this method is used to monitor the behavior of a process, small deviations from the known normal behavior would likely be acceptable. Many other change detection algorithms (such as checksums and message digests) are sensitive to any change in the data and therefore not applicable when a certain amount of noise tolerance is required. On the other hand, if the method is being used to detect viral code in a binary program file, both the self strings and the viral strings will consist of short machine code segments. Pieces of viral code would likely be more similar to the valid self strings than to a totally random string of the same length, and thus be more likely to be obscured by holes. In either case, it may be wise to calculate P_f based on the a priori probabilities of expected changes (if known) instead of on random nonself strings.

3. Algorithms

This section describes three different algorithms for generating detector sets. Subsection 3.1 covers the original generate-and-test algorithm presented in [Forr94]. Subsections 3.2 and 3.3 present two algorithms based on dynamic programming, which run in linear time with respect to the size of the input: the linear time algorithm and the greedy algorithm. For some examples and a more detailed derivation of the time and space complexities of these algorithms see [Helm94] and [Dhae95a]. This last report also contains the algorithm for counting the holes, included here in subsection 3.4.

3.1. Generate-and-Test Algorithm

This algorithm mimics the generation of T cells in the immune system. Strings are drawn at random from U_d and checked against all strings in S. If they fail to match any of the self strings, they are kept as valid detectors. This process of random generation and checking against S is repeated until the required number of detectors is generated.

Generate-and-Test requires generating a number of random detector strings (N_{R_0}: initial detector repertoire size, *before* negative selection) that is exponential in the size of self (for a fixed matching probability P_m) [DeBo93]:

$$N_{R_0} = \frac{-\ln(P_f)}{P_m \cdot (1-P_m)^{N_S}} \ .$$

For independent detectors, we can approximate the failure probability P_f achieved by N_R detectors:

$$P_f \approx (1-P_m)^{N_R} . \qquad (6)$$

For P_m sufficiently small and N_R sufficiently large, this gives:

$$N_R \approx -\ln(P_f)/P_m \ . \qquad (7)$$

The assumption that the detectors are independent is not entirely valid. As N_S or P_m increases, the candidate detector set (C in Figure 2) will shrink, so the detectors chosen become less independent. Overlap among the detectors decreases the amount of string space covered, resulting in a higher failure probability P_f than (6) would indicate.

The time complexity of this algorithm is proportional to N_R, the number of candidate detectors that need to be examined, and N_S (because each string may have to be compared against all self strings). Space complexity is determined by N_S:

$$\text{time: } \mathcal{O}\left(\frac{-\ln(P_f)}{P_m \cdot (1-P_m)^{N_S}} \cdot N_S\right),$$

$$\text{space: } \mathcal{O}(l \cdot N_S).$$

3.2. Linear Time Algorithm

The generate-and-test algorithm is inefficient because most of the candidate detector strings are rejected, but works for arbitrary matching rules. Some matching rules allow for more efficient detector generating algorithms. Here, we describe a two-phase algorithm for the "r-contiguous-bits" matching rule (two l-bit strings match each other if they are identical in at least r contiguous positions) that runs in linear time with respect to the size of the input (for fixed matching parameters l and r). In Phase I, we use dynamic programming to solve a counting recurrence for the number of strings unmatched by strings in S (candidate detectors, set C in Figure 2). In Phase II, we use the enumeration imposed by the counting recurrence to pick detectors randomly from this set of candidate detectors.

Phase I: Solving the counting recurrence. Essentially, we try to answer the following question: For any r-bit string s, how many l-bit strings starting with s are not matched by any string in S? We answer this question recursively, by counting for any r-bit string s how many $r+k$ bit strings starting with s are not matched by the last $r+k$ bits of any string in S. This can be calculated based on the number of $r+k$-1 bit strings starting with $\hat{s}$ (denoting s stripped of its first bit) followed by either a 0 or 1, that are not matched by the last $r+k$-1 bits of any string in S. These numbers are stored in an array $C_k[s]$, built up recursively with k increasing from 0 to l-r. See the Appendix for a simple example of the construction of a C array.

Phase II: Generating strings unmatched by S. Note that $C_{l-r}[.]$ sums up to the total number of strings not matched by any string in S, i.e. |C|, the size of the set of candidate detectors C in Figure 2. We can impose a natural ordering on the candidate detectors by sorting them by their digital representation. The first candidate detectors will then start with r zeroes, and we know from above that there are exactly C_{l-r}[00…00] such strings not matched by S. Likewise, the next C_{l-r}[00…01] candidate detectors will start with 00…01, etc. If we now pick a random number between 0 and |C|, we can determine which *r*-bit string *s* the corresponding candidate detector starts with, based on the partitioning imposed by $C_{l-r}[s]$. Furthermore, by construction each $C_{l-r}[s]$ is the sum of two entries $C_{l-r}[\hat{s}.0]$ and $C_{l-r}[\hat{s}.1]$. This further partitions the $C_{l-r}[s]$ candidate detectors starting with *s* into those whose next bit is either a zero or a one. So for each random number between 0 and |C|, $C_{l-r}[.]$ determines the first *r* bits of the unique detector that corresponds to this number, whereas the next layers of the $C_k[.]$ array determine the succeeding bits, with $C_0[.]$ determining the last bit of the detector. We can use this to pick N_R valid detectors uniformly over the set of all candidate detectors C. See the Appendix for a simple example of how to retrieve a random detector based on a *C* array.

The principle data structure used in this algorithm consists of the large $(l-r)\times 2^r$ *C* array representing all the possible ways two strings can match over *r* contiguous bits. This has an impact on the time and space complexity of the algorithm. Phase I takes $\mathcal{O}\left((l-r)\cdot N_S\right)$ time to initialize those entries in *C* that match a self string, plus $\mathcal{O}\left((l-r)\cdot 2^r\right)$ time to recursively fill in the rest of the array. Phase II takes $\mathcal{O}(r)$ time to find the first *r* bits of each detector, and $\mathcal{O}(l-r)$ time to complete each detector:

time: $\mathcal{O}\left((l-r)\cdot N_S\right)+\mathcal{O}\left((l-r)\cdot 2^r\right)+\mathcal{O}\left(l\cdot N_R\right)$,

space: $\mathcal{O}\left((l-r)^2\cdot 2^r\right)$.

The above algorithm runs in time linear in the size of the self set and detector set (for given parameters *l* and *r*). This is in contrast to the generate-and-test algorithm, which ran in time exponential to the size of the self set, but required essentially constant space. The linear time algorithm still requires time and space exponential in the length *r* of the matching region, which could present a problem if we need to choose long strings (*l*) and matching regions (*r*).

Phase I of the linear time algorithm allows us to count exactly how many detectors do not match any of the self strings in *S*. Similarly, given a detector set *R*, we can use Phase I to count exactly how many self plus nonself strings are not matched by any of the detectors. This count can be used to check whether enough detectors have been generated to achieve a given allowable failure probability.

3.3. Greedy Algorithm

We can achieve a better coverage of the string space with the same number of detectors (or a smaller detector set for the same amount of coverage) by not selecting the detectors at random, but placing them as far apart as possible. The greedy algorithm we developed tries to do exactly that. At each step it picks one of the detectors that will match as many as possible of the as yet unmatched nonself strings.

To construct the *C* array in Phase I of the previous algorithm we chose to examine the strings from right to left (from $C_0[.]$ to $C_{l-r}[.]$). We can also go through the strings from left to right, constructing a second array C' starting at $C'_{l-r}[.]$ and calculating the following levels using a similar recurrence relation as for *C*. $D_k[s] = C_k[s]\times C'_k[s]$ then represents the number of *l*-bit strings that have the *r*-bit string *s* in position *k*, unmatched by any string in *S*. *s* and *k* define a *template*, i.e. a partial specification of some of the bits of a bitstring, and $D_k[s]$ represents the number of strings with this template that are unmatched by S.

If a given template has a zero entry in D, we know that all strings containing that template will match some string in S. Conversely, if we restrict ourselves to picking templates with nonzero entries in D when constructing bit strings, we know those strings will not be matched by any string in S. The algorithm uses two different D arrays, called D_S and D_R, the first one based on the self set S and the second one based on the current state of the detector set R.

Phase I: Generate the D_S array: this array will tell us from which templates we are allowed to choose when constructing detectors. We will call the templates that have nonzero entries in this D_S array *valid detector templates.*

Phase II: Generating strings unmatched by S. The second array D_R, based on the current detector set R, will indicate how many strings for each template are not yet matched by the previously generated detectors. The detector set starts out empty and all entries in D_R are initialized to 2^{l-r}, the total number of l-bit strings containing an r-bit template. For each new detector to be generated, we try to select the templates matching the most unmatched strings by searching through D_R for the valid detector template with the largest entry. If there is a tie between templates, we choose one at random. Starting from this template, we then traverse the D_R array to the left and to the right, each time choosing to add a 0 or 1 bit to the starting template depending on which represents the template with the highest number of strings not yet matched by R (while still restricting ourselves to valid detector templates). Next, we add the completed detector to R and update the D_R array to reflect accordingly.

We repeat this process of picking a detector and updating D_R until all valid detector templates have zero entries in D_R. At that point, for any template that is not in S there are no more strings that have not yet been matched by a detector, i.e., we have covered all strings that *can* possibly be covered by detectors. We call this a *complete* detector repertoire.

This algorithm also has the attractive property that we can keep a running count of the number of nonself strings that are still unmatched by any detector. For a given acceptable failure probability P_f we can simply keep generating detectors until we reach the corresponding number of unmatched nonself strings (or until we run out of candidate detectors, which would mean that there are too many holes to be able to reach the desired P_f).

At best, we can spread the detectors apart such that no two detectors match the same nonself string, which would get us the lower bound on the number of detectors mentioned earlier [Pere79]:

$$N_R \geq \left(1 - P_f\right) / P_m . \qquad (3)$$

Looking at the structure of the template array we can get another estimate for N_R. Each detector generated matches one of the 2^r templates in each of the (l-r) columns of the template array, and sets the corresponding count there to zero. We can zero out all the entries in a column with at most 2^r detectors:

$$N_R \leq 2^r .$$

This formula does not take into account the entries in the template array already zeroed out by matching self strings. If we assume the self strings are independent, each template has a chance $(1 - 2^{-r})^{N_S}$ of not matching any of the N_S self strings. The estimate for N_R then becomes:

$$N_R \propto 2^r \cdot \left(1-2^{-r}\right)^{N_S} \quad . \qquad (8)$$

The time complexity for this algorithm is dominated by the time needed to update the template array for each new detector generated, making it quite a bit slower than the previous algorithm, although the space complexity is of the same order:

time: $\mathcal{O}\left((l-r)\cdot 2^r \cdot N_R\right)$,

space: $\mathcal{O}\left((l-r)^2 \cdot 2^r\right)$.

3.4. Counting the holes

As we mentioned earlier, even though the above algorithm is capable of constructing a complete detector repertoire, this does not necessarily mean it can construct a detector set capable of recognizing all non-self strings, i.e., all strings not in S. Depending on the matching rule used and the strings in *S*, there may be some holes for which it is impossible to generate valid detectors.

Running the greedy algorithm until all valid detectors have been used up would tell us exactly how many nonself strings cannot be detected. In [Dhae95a] we developed a more efficient algorithm for counting the exact number of holes, similar to the way the number of detector strings not matched by S are counted in the linear time algorithm. In the previous section we showed how we can generate detectors by only picking valid detector templates in array D_S. Similarly, holes will have to contain only *invalid* detector templates, because if a string were to contain a valid detector template, we could construct a detector to match that string using this template. Once we have calculated D_S in order to determine which are the invalid detector templates, we can count the exact number of undetectable strings (self strings plus holes) similar to the way the number of candidate detector strings are counted in the linear time algorithm. The space and time complexities are similar:

time: $\mathcal{O}\left((l-r)\cdot N_S\right)+\mathcal{O}\left((l-r)\cdot 2^r\right)$,

space: $\mathcal{O}\left((l-r)^2 \cdot 2^r\right)$.

The reasonably short running time makes this a useful tool in discovering appropriate settings for the *l* and *r* parameters of the matching rule. At the least, we want the number of holes, N_H, to be small enough to allow the desired failure probability P_f : $N_H \le P_f \cdot 2^l$. If we stick close to this upper bound on the allowed number of holes, we can expect to need almost all valid detectors to cover the very last nooks and crannies of the detectable nonself string space. If the number of holes is much smaller, we may need substantially fewer detectors for the same P_f . The smaller the fraction of non-zero entries there are in the template array, the more holes there will be (because holes consist solely of templates that have zero entries in the template array). The template array becomes sparse if $N_S >> 2^r$, because each of the N_S self strings can match one of the 2^r templates. In order to get only a small number of holes, we should therefore use

$$N_S \le 2^r \quad . \qquad (9)$$

4. Practical considerations and results

In this section we use the algorithms and analysis from the previous sections to explore which parameter settings are practical. Subsection 4.1 looks at results obtained for relatively small randomly generated self sets, and draws some conclusions from comparisons between different algorithms and formulas. Subsection 4.2 looks at a much larger real-world example and examines how the failure probability scales with detector set size and matching length *r*. In Subsection 4.3 we present a summary of the appropriate formulas for the r-contiguous-bits matching rule, and give step-by step instructions on how to choose the parameters for a real data stream.

4.1. Repertoire sizes using different algorithms

Table 2 shows detector repertoire sizes obtained with the different algorithms using randomly generated self files and a number of different parameter sets (N_S , *l* and *r*), as well as some upper and lower bounds predicted by the formulas presented in the previous sections. We have arbitrarily chosen P_f =0.1 as an acceptable failure rate. Because the algorithm might be able to cover more of the total nonself string space with a smaller set of detectors if there is some structure to the self strings, independent self strings will tend to be a worst case situation for estimating N_R (ignoring the effect of the holes on the achievable P_f).

When we compare the results in columns (e) and (h), we see that the greedy algorithm generates a detector set that is from 8% to 41% smaller than the size predicted for independently chosen detectors. Also note that formula (2) indicates that the desired P_f should be reachable with a detector set of a certain size, although there might be so many holes in the string space that this P_f is unreachable, as indicated by the entries marked “(*)” in (h). For the linear time and the generate-and-test algorithm, generating a detector set of the size indicated by (7) does not give us any guarantee that we will achieve the specified P_f . For these algorithms, it will be necessary to count the number of undetected strings a posteriori (as described in Section 3.2), and generate some more detectors if necessary. This is not a problem for the greedy algorithm, because we have a running count of the number of unmatched strings (at the expense of a more time consuming algorithm), so we can continue generating detectors until P_f falls below the specification.

L_S (a)	$N_S \times l$ (b)	r (c)	P_m (d)	indep. $P_f = 0.1$ (e)	greedy complete (f)	estimate complete (g)	greedy $P_f = 0.1$ (h)	optimal $P_f = 0.1$ (i)	entropy $P_f = 0.1$ (j)	number of holes (k)
500	250×16	10	0.0039	589	793	802	374	230	52	634
		9	0.0088	262	320	314	212	102		4438
		8	0.0195	118	88	96	(*)	46		21076
1K	250×32	11	0.0056	410	1796	1813	313	160	26	2649
		10	0.0117	196	821	802	153	77		24911
		9	0.0244	94	378	314	76	37		2150714
		8	0.0508	45	89	96	(*)	18		5.182e+08
	500×16	11	0.0017	1347	1610	1604	829	526	104	882
		10	0.0039	589	632	628	445	230		3854
		9	0.0088	262	155	193	(*)	102		24937
2K	500×32	12	0.0027	856	3584	3625	650	335	52	4787
		11	0.0056	410	1668	1604	316	160		52318
		10	0.0117	196	772	628	159	77		2420706
		9	0.0244	94	201	193	(*)	37		4.656e+08
	1000×16	12	0.0007	3154	3204	3207	1856	1233	208	1353
		11	0.0017	1347	1258	1257	1233	526		5428
		10	0.0039	589	340	385	(*)	230		23933
4K	1000×32	13	0.0013	1799	7144	7251	1352	703	104	8475
		12	0.0027	856	3304	3207	659	335		85798
		11	0.0056	410	1516	1257	328	160		3991790
		10	0.0117	196	378	385	(*)	77		5.230e+08

Table 2: Repertoire sizes and number of holes for different configurations. (a): file size in bytes. (b), (c): parameters chosen for the matching rule. (d): corresponding matching probability P_m for the r-contiguous-bit matching rule. (e): N_R calculated according to formula (7), for independent detectors. (f): size of *complete* detector repertoire generated with the greedy algorithm. (g): estimate for (f) based on formula (8). (h): N_R generated by greedy algorithm until $P_f \leq 0.1$ ((*) means that $P_f \leq 0.1$ could not be achieved). (i): lower bound on (h) based on formula (3). (j): entropy-based lower bound (4) on (h). (k): number of holes present.

With a greedy selection of detectors, the last detectors generated will only match a small number of as yet unmatched nonself strings, and will therefore not have a significant effect on P_f. This means that when going from $P_f = 0.1$ to $P_f = \min_{N_R}(P_f)$, quite a large number of extra detectors may have to be added to match all of the tiny unmatched regions of nonself string space. This explains the large difference in detector set size between, for example, 90% detection rate (h) and the maximum detection rate (complete repertoire, (f)).

Both (3) and (4) (columns (i) and (j) in Table 2) are indeed effective lower bounds on the size of the detector set needed for a failure probability of P_f =0.1. The entropy-based lower bound is less strict, partially because it does not take the properties of the matching rule into account. The lower bound in column (i) is based on optimal spacing of the detectors, which is precisely what the greedy algorithm tries to achieve, so we could view column (i) as the optimal detector set size for the greedy algorithm. Interestingly, when the greedy algorithm aims for an optimal detector set size that is smaller or almost equal to the size indicated by the entropy-based lower bound (i.e., (i)≤(j)), it is unable to do so (entry in (h) is "(*)") because the number of holes in the string space (column (k)) is too large with respect to the desired P_f . This suggests that

$$P_m \le \frac{l \cdot \log_2(m) \cdot \left(1 - P_f\right)}{N_S \cdot \log_2\left(1/P_f\right)} \qquad (5)$$

indeed provides an interesting upper bound on the value for P_m (and therefore *r*) given N_S, *l* and P_f .

Within one set of rows with the same N_S and *l*, we see that the repertoire size decreases with matching length *r*. However, the number of holes in string space seems to increase in an exponential fashion with decreasing *r* up to the point where we can no longer find an adequate detector set for the acceptable failure probability P_f . Since P_m increases exponentially with (*l*-*r*), a smaller matching length means that each detector matches more strings, so fewer detectors are needed. However, each self string will also match more detectors, so the set of candidate detectors becomes smaller and the number of holes gets larger due to interaction between self strings. For the smallest *r* for which we can still achieve P_f=0.1, a large number of the nonself strings not matched by *R* are holes. As mentioned before, using different matching lengths at the same time would allow us to combine matching all nonself strings and covering most of the nonself string space by a small number of detectors.

By looking at rows with the same L_S, we can get an idea of the effect of choosing a shorter or longer string length to split the data up in self strings. For instance, the rows with N_S =250 and *l*=32 are generated for the same data as the rows with N_S =500 and *l*=16. Similarly for N_S =500, *l*=32 and N_S =1000, *l*=16. We see that with a longer string length *l*, a smaller number of detectors is needed to achieve P_f = 0.1. If we look at how much space is taken up by the detector set ($N_R \times l$), the larger string length still comes out ahead. Note that with the larger string length the number of holes in the string space is substantially larger (for the same values of *r*). This is due to the fact that the string space itself is much larger as well. However, the fraction of string space occupied by holes is smaller for larger *l* because the self strings are spaced farther apart and therefore interact with each other less. This means that for a larger string length *l*, we can choose a smaller *r* (larger matching probability) and still have an acceptable detector set.

4.2. P_f versus N_R for a real data file

Figure 5 shows how P_f varies with N_R for a binary file (GNU emacs v19.25.2 SGI binary, 3.2MB). The data for each value of *r* was derived from a single large detector set generated with the linear time algorithm (one million detectors for *r*=16 and *r*=18; 5 million detectors for *r*=19). 1000 nonself strings were checked against each of these detector sets, and for each nonself string we recorded the index of the *first* detector to match the string. We can then derive the probability of success (1- P_f) as the cumulative histogram over these values. Note that this means that the points in each curve are not completely independent. However, this approach gives us a reasonable approximation for a much smaller computational effort.

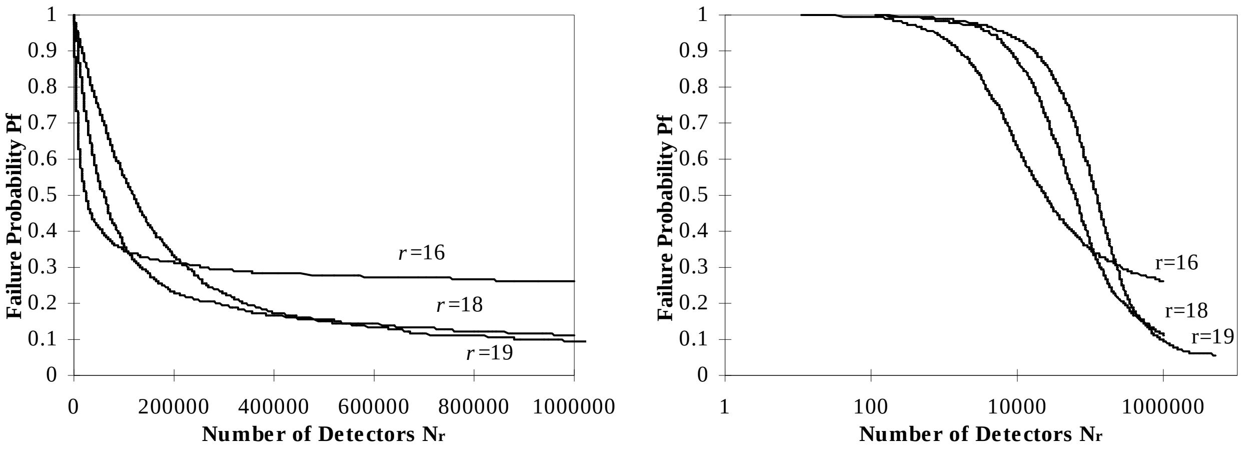

Figure 5: P_f versus N_R for a binary file. (a): linear scale for N_R; (b): log scale for N_R.

The figure shows at first a sharp, almost linear, drop in the failure probability, due to each detector covering a fraction of the nonself space. As the randomly chosen detectors start to overlap, this linear decline becomes an exponential. If we were to use the greedy algorithm to generate the detector set, we would expect this linear decline to continue much farther, as each new detector is designed to overlap with the previous detectors as little as possible. The decline in P_f is sharper for smaller values of *r*, i.e. larger matching probabilities, because each detector matches a larger fraction of string space, so most of the space can be covered by a small number of detectors. As the detector set size increases even further, P_f starts to level off to a non-zero value, because of the presence of holes. Even though P_f decreases sharper initially for smaller *r*, it levels off at a higher level because there are more holes. This leveling off at a non-zero value can be avoided by using variable-radius detectors, or by using several detector sets with different matching rules.

Note also that for each detector set size there is an optimal value for *r*. In general, if we want to have a smaller detector set, we will have to use a smaller value for *r* (such that each detector matches more nonself strings). Similarly, for each value of P_f there is an optimal value for *r*. As the acceptable failure rate decreases, we will have to go to larger values of *r* (more specific detectors) and larger detector sets.

Finally, if we are protecting this file in a distributed fashion, with a number of different machines each running its own detector set, we can exploit the fact that the failure probability decreases exponentially with the number of detector sets. In that case we may already be satisfied with a failure rate for the individual detector sets of $P_f \approx 0.5$. Theoretically, four independent detector sets with P_f = 0.5 would give a total failure rate of only 0.0625. Figure 5 shows that this can be achieved with a repertoire as small as 30,000 to 120,000 detectors. This is a much more reasonable size, given that the original self file contains about one million strings. Similarly, if we assume each anomaly results in a number of independent nonself strings, fairly high per-string failure rates like this can still give a very low per-anomaly failure rate.

4.3. Summary of formulas and rules of thumb

4.3.1. Choosing the alphabet size

The larger the alphabet size used, the harder it becomes to make an optimal choice for the matching length *r*. This is due to the fact that for the matching rules considered here, the matching probability $P_m \propto m^{-r}$ [Perc92, Uspe37]. Assuming that P_m has to stay within certain bounds for the detection algorithm to perform efficiently, the range of acceptable values for *r* becomes very narrow with increasing alphabet size.

For some applications, however, a non-binary alphabet may be more appropriate. For example, [Forr94] describes an experiment in which a C program compiled for a RISC architecture was checked for changes. Each opcode was mapped into one of 104 symbols (*m*=104). An important area of future investigation is to study the performance of different size alphabets, especially in those cases where a non-binary alphabet is most natural.

4.3.2. Choosing the string length

First, we determine a lower bound for *l* by requiring that the self strings should occupy only a fraction of the total string space:

$$N_S \approx L_S / l \leq 2^l \ .$$

This corresponds to 1-byte strings for a stream of up to 256 bytes (2Kbits), 2-byte strings for streams of up to 128KB (1Mbits), 3-byte strings for streams of up to 50MB (400Mbits), and 4-byte strings for streams up to 17GB (137Gbits). Note that this is not an exact lower bound for *l*, because splitting up the data into *l*-bit strings may cause many duplicate strings, so $N_S \leq L_S / l$. If we want to know the exact lower bound for *l* (or if the data to be checked for changes is not a fixed size file but rather a continuous stream of data) we can explicitly count the number of unique *l*-bit strings for different values of *l*.

The second step in selecting string length is to determine whether there exists a natural string length imposed by the data. For instance, to protect a database consisting of a series of 3-byte records, a multiple of three bytes would be an obvious string length to try because it preserves the structure of the data. If the data does not exhibit any natural string length, we might still be able to find by inspection certain recurring features that we want to be able to capture. The string length will have to be longer than these features for them to be preserved in the self strings. Finally, there will be an upper bound to the string length, imposed by the generating algorithms. Increasing *l* usually also means *r* has to be increased (certainly if we want the matching probability to remain constant). However, for large *l* and *r*, the algorithms described in this report become computationally very expensive.

4.3.3. Choosing the matching length

To keep the number of holes low, we want to pick *r* such that $N_S \leq 2^r$ (9). Again, we might want to replace N_S in this formula by the number of unique *l*-bit strings that appear in the data.

As illustrated by Table 2, formula (5) gives an approximate upper bound on P_m (and thus a lower bound on *r*, for a given string length *l*) in order to be able to construct a detector repertoire for the acceptable failure probability P_f :

$$P_m \le \frac{l \cdot \log_2(m) \cdot \left(1 - P_f\right)}{N_S \cdot \log_2\left(1/P_f\right)} \qquad (5)$$

Finally, after a minimum value for *r* has been chosen, we should count the exact number of holes present in the string space. If this exceeds the acceptable number of holes of $N_H \le P_f \cdot 2^l$, we choose a smaller matching probability (i.e. larger *r*) and try again. In general, because we do not have an exact procedure to determine the optimal values for *l* and *r*, a number of valid combinations may have to be tried and weighed against each other in terms of ease of construction of the detector set, size of the resulting detector set, attainable failure probability, etc.

4.3.4. Choosing the number of detectors

We have seen a number of lower bounds and estimates for N_R under different assumptions. We will go through these one by one, from the least tight lower bounds to what we expect to be the best estimates.

If the parameters of the self set and matching rule are such that generating the detectors using the greedy algorithm is too expensive, we can use the linear time algorithm to select a set of detectors independently chosen from the candidate detectors. If we assume the detectors chosen are also independent of each other with respect to the total string space, we get the lower bound from (7):

$$N_R \approx -\ln\left(P_f\right)/P_m \ . \qquad (7)$$

This last independence assumption usually doesn't hold: because of the limited number of candidate detectors, there will be more overlap between detectors than we would expect from strings chosen independently from the entire string space. If this is the case, the number of detectors needed to achieve P_f could be quite a bit larger than the value indicated by (7). We may have to estimate the actual P_f a posteriori over a sample of nonself strings.

If we use an algorithm that attempts to spread out the detectors, such as the greedy algorithm, the minimum number of detectors needed to achieve an acceptable P_f is given by (3):

$$N_R \ge \left(1 - P_f\right)/P_m \ . \qquad (3)$$

This is an absolute lower bound, in the sense that it is impossible to cover that much of string space with fewer detectors. However, depending on the structure present in the self data, it can be difficult to pick detectors that are spread apart optimally. Note that if we have chosen *r* according to formula (5), using this lower bound for N_R will automatically imply that we also satisfy the entropy lower bound (4). Using (3) to calculate the number of detectors needed for the greedy algorithm is only interesting in terms of getting a ballpark figure for N_R to evaluate whether the parameters *l* and *r* have been chosen efficiently. If we are satisfied with the estimate we simply run the greedy algorithm until it reaches exactly the desired P_f . If we are interested in achieving the minimum possible failure probability P_f , we can construct a complete detector repertoire by running the greedy algorithm till exhaustion. (8) provides a fairly close estimate for the size of the complete detector set for independent self strings:

$$N_R \propto 2^r \cdot \left(1 - 2^{-r}\right)^{N_S} \ . \qquad (8)$$

4.3.5. Detection scheme

One area that we have not yet examined closely is how best to implement the actual detection scheme once an appropriate detector set has been generated. Here are some possible examples:

- Maximum security: every string is checked against the entire detector set. Other, more conventional, anomaly detection algorithms might be more appropriate in this case.
- Intermittent checking: at certain intervals, a small number of strings is checked against a small number of detectors chosen at random from the detector set. This relies on the probabilistic nature of our anomaly detection method. It assumes that if a change is undetected, it will be detected during some other check, or on another machine. This might not be acceptable if a single occurrence of the change can be fatal.
- Weighted detection: detectors can be chosen more frequently depending on previous performance, based on known expected changes (known failure modes, virus signatures, etc.), or in order to get a homogeneous coverage of nonself string space (some areas of nonself string space may be covered by more detectors than others).
- Distributed detection: the detector set is split up over a number of autonomous agents (see [Cros95]) each doing checks in parallel. This is also the scheme used in the real immune system, where each T cell corresponds to a single detector.
- Distributed independent detection: each agent has a detector set generated independently from all other agents. This is similar to a population of individuals with immune systems. The advantage is that holes in one individual’s immune system can be covered by another’s, so an infection cannot spread through the entire population. A similar setup could be used to protect a whole network of computers.

5. Conclusions

Previous work on this new immunological approach to anomaly detection had shown its feasibility. In particular, the linear time algorithm has made it practical to construct efficient detector sets for large data sets. We have also made significant improvements towards quantifying the range of acceptable values for the different parameters associated with the detection method, to the point that we can give guidelines for setting up a detection system like this for any given data set. The distributed nature of this algorithm is promising for networked and distributed computing environments. As a very general-purpose anomaly detection method it could be a useful supplement to more specific, and therefore more brittle, protection mechanisms. We imagine a layered computer immune system, with specific protection mechanisms against well known or previously encountered intrusions, and non-specific protection mechanisms like the one presented here to intercept those intrusions that evade the specific mechanisms.

There are a number of remaining issues to be examined. Here are the most important ones:

- Developing more general formulas for systems in which a certain rate P_f' of false positives is allowed.
- Deriving approximate formulas for non-random data by looking at some measures of the self strings (entropy, number of unique self strings, etc.).

- It may prove useful to analyze the detectors with respect to their Vapnik-Chervonenkis dimension (VC-dimension). In particular, going from detectors with a fixed matching radius *r* to a variable radius, as was suggested in section 2.3 to eliminate holes, corresponds to using detectors with a higher VC-dimension.
- Constructing a linear time algorithm for the Hamming distance matching rule. This matching rule might give improved performance because it does not limit the length of the matching templates and should therefore be able to capture larger structures in the self strings.
- In order to eliminate the negative effect of holes on the system-wide failure probability, one would want to have a family of matching rules with the same basic parameters. This would allow the parameters to be chosen optimally for the task, while still allowing for an effectively different matching rule at each site. Families of matching rules based on permutation or encryption with a unique key (as touched upon in Section 2.3) need to be analyzed with respect to their ability to generate non-overlapping sets of holes.
- From a security point of view, it might be useful to have a matching rule for which it is provably hard to construct (and thus to forge) a detector set. Can we come up with matching rules based on some NP-complete problems for instance?

6. Acknowledgments:

The authors thank Dipankar Dasgupta, Derek Smith, Ron Hightower and Andrew Kosoresow for their useful suggestions and critical comments. The idea of a computer immune system grew out of a collaboration with Dr. Alan Perelson through the Santa Fe Institute. Portions of this paper were written while S. Forrest was on sabbatical at the MIT AI Lab. This work is supported by grants from the National Science Foundation (grants IRI-9157644 and IRI-9711199), Office of Naval Research (grant N00014-95-1-0364), Interval Research Corporation, General Electric Corporate Research and Development, and Digital Equipment Corporation.

APPENDIX: The Linear Time Algorithm in Action

Let l=6, r=3, S = {`110100,100101`}. Table 3 shows the C array constructed for this self set. First, those entries corresponding with templates occurring directly in any of the self strings are zeroed out. This is the case for `110***`, `100***`, `*101**`, `*001**`, `**010*`, `***100` and `***101` (marked in bold). All the remaining entries in C_0[`s`] are set to 1. The entries in the rest of the array are then filled up according to C_{n+1}[`000`] = C_n[`000`] + C_n[`001`]; C_{n+1}[`001`] = C_n[`010`] + C_n[`011`], … C_{n+1}[`111`] = C_n[`110`] + C_n[`111`]. Note that C_1[`110`] = 0, even though template `**110*` does not directly match any string in S. However, any string conforming to the template `**110*` will have to contain either `**1100` , and match `110100`, or `**1101`, and match `100101`.

s	C_3[s]	C_2[s]	C_1[s]	C_0[s]
000	4	4	2	1
001	6	**0**	2	1
010	4	4	**0**	1
011	6	2	2	1
100	**0**	4	2	**0**
101	6	**0**	2	**0**
110	**0**	4	0	1
111	6	2	2	1

Table 3: *C* Array constructed for *S* = {110100,100101}, *r*=3.

The sum of entries in C_3[s] is 32, indicating that there are 32 candidate detectors, the first 4 of which start with 000***, the next six with 001***, etc. We can now generate random, valid, detectors by choosing a number between 0 and 31, and reconstructing the corresponding detector. Suppose we pick 17. Table 4 shows how we proceed to retrieve the detector corresponding to this number. The second column in this table (labeled "range") shows that detectors 14 till 19 start with 011**. Detector 17 is the fourth detector in this range. We know that C_3[011] = C_2[110] + C_2[111] = 4 + 2. In other words, the first four detectors in this range, including detector 17, will have a 0 next (corresponding to template *110**), the last two detectors in this range will have a 1 next (corresponding to template *111**). Thus, we know that detector 17 starts with 0110**. Again, C_2[110] = C_1[100] + C_1[101] = 2 + 2. Detector 17 is the last detector within this range, therefore it belongs to template **101*. Similarly, we find template ***011 for the final bit. The total detector is thus 011011, which clearly does not match either string in S.

s	range	C_3[s]	C_2[s]	C_1[s]	C_0[s]
000	0-3	4	4	2	1
001	4-9	6	0	2	1
010	10-13	4	4	0	1
011	**14-19**	**6**	2	2	**1**
100	-	0	4	2	0
101	20-25	6	0	**2**	0
110	-	0	**4**	0	1
111	26-31	6	2	2	1

Table 4: Retrieving detector 17 from *C* Array.

References:

[Cove91] T. M. Cover, J. A. Thomas. *Elements of Information Theory*, a Wiley-Interscience Publication, 1991

[Cros95] M. Crosbie and G. Spafford. Defending a Computer System using Autonomous Agents. *In Proceedings of the 18th National Information Systems Security Conference*, 1995.

[DeBo93] R. J. De Boer and A. S. Perelson. How diverse should the immune system be? In *Proceedings of the Royal Society* London B, v.252, London, 1993.

[Dhae95a] P. D'haeseleer. Further efficient algorithms for generating antibody strings. Technical Report CS95-3, The University of New Mexico, Albuquerque, NM, 1995.

[Dhae95b] P. D'haeseleer. A change-detection algorithm inspired by the immune system: Theory, algorithms and techniques. Technical Report CS95-6, The University of New Mexico, Albuquerque, NM, 1995.

[Dhae96a] P. D'haeseleer, S. Forrest and P. Helman. An Immunological Approach to Change Detection: Algorithms, Analysis and Implications. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Dhae96b] P. D'haeseleer. An Immunological Approach to Change Detection: Theoretical Results. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Dhae97] P. D'haeseleer. An Entropic Lower Bound on Classifier Size, in preparation.

[Dipa97] D. Dasgupta and N. Attoh-Okine. Immunity-Based Systems: A Survey. To appear in *Proceedings of the 1997 IEEE International Conference on System, Man and Cybernetics*, 1997.

[Dipa95] D. Dasgupta and S. Forrest. Tool Breakage Detection in Milling Operations using a Negative-Selection Algorithm. Technical Report CS95-5, The University of New Mexico, Albuquerque, NM, 1995.

[Farm93] D. Farmer, W. Venema. Improving the Security of Your Site by Breaking Into it. ftp.win.tue.nl pub/security/admin-guide-to-cracking.101.Z, 1993.

[Forr94] S. Forrest, A. S. Perelson, L. Allen and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Forr95] S. Forrest, S. A. Hofmeyr, A. B. Somayaji and T. A. Longstaff. A sense of self for UNIX processes. *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Helm94] P. Helman and S. Forrest. An efficient algorithm for generating random antibody strings. Technical Report CS-94-07, The University of New Mexico, Albuquerque, NM, 1994.

[Inma78] J.K. Inman. The antibody combining region: Speculations on the hypothesis of general multispecificity. In *Theoretical Immunology*, M. Dekker, New York, NY, 1978.

[Jane96] C.A. Janeway and P. Travers. *Immunobiology : The Immune System in Health and Disease*, Garland Publishing, 3rd Edition, 1997.

[Kapp87] J.W. Kappler, N. Roehm, P. Marrack. T cell tolerance by clonal elimination in the thymus. In *Cell*, 49:273-280, 1987.

[Koso97] A. Kosoresow, S. A. Hofmeyr. A Shape of Self for Unix Processes. Submitted to the 1997 IEEE Symposium on Security and Privacy, 1997.

[Paul89] W. E. Paul, Ed. *Fundamental Immunology*, Raven Press Ltd. New York, 88-90, 1989.

[Perc92] J. K. Percus, O. E. Percus and A. S. Perelson. Probability of Self-Nonself discrimination. *In Theoretical and Experimental Insights into Immunology*, 1992.

[Pere79] A. S. Perelson and G. F. Oster. Theoretical Studies of Clonal Selection: Minimal Antibody Repertoire Size and Reliability of self-nonself Discrimination. In *Journal of Theoretical Biology*, 1979.

[Uspe37] J. V. Uspensky. *Introduction to Mathematical Probability*, McGraw-Hill, NY, 1937.