An Immunological Approach to Change Detection: Algorithms, Analysis and Implications

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

Abstract

We present new results on a distributable change-detection method inspired by the natural immune system. A weakness in the original algorithm was the exponential cost of generating detectors. Two detector-generating algorithms are introduced which run in linear time. The algorithms are analyzed, heuristics are given for setting parameters based on the analysis, and the presence of holes in detector space is examined. The analysis provides a basis for assessing the practicality of the algorithms in specific settings, and some of the implications are discussed.

1. Introduction

It is impractical to find and patch every security hole in a large computer system. Thus, the need for a more comprehensive approach to security is increasing. Any single protection mechanism is likely vulnerable to some class of intrusions. For example, relying on a protection mechanism that is designed for known types of intrusion implies vulnerability to novel intrusion methods. It is our belief that a multi-faceted approach is most appropriate, in which, similar to natural immune systems, both specific and non-specific protection mechanisms play a role.

This paper is concerned with one aspect of our overall strategy: the very general problem of change detection. The method discussed is non-specific, in the sense that it is not specifically aimed towards certain well-known attacks, as opposed to, for example, one using known signatures. It is also general in the sense that it could be used for a wide variety of change-detection problems, including those requiring some tolerance of noise, or involving dynamic streams of data (such as activity patterns in running processes [7]). On the other hand, it might not always be as efficient 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 changes that might otherwise go undetected.

The change-detection method we are studying was inspired by the generation of T-cells in the immune system. In the thymus, T-cells with essentially random receptors are generated, but before they are released to the rest of the body, those T-cells that match self proteins are deleted [9, 10]. Similarly, our method distinguishes self strings (the protected data or activities) from nonself strings (foreign or malicious data or activities) by generating detectors for anything that is not in the set of self strings. This principle of trying to match anything that has not previously been encountered we call "negative detection."

Many methods for change detection rely on a centralized detection protocol, i.e., each object has to be checked in its entirety, and the monitor has to contain all the information about the original objects. Our method on the other hand is inherently distributable:

- Small sections of an object can be checked for change independently.
- Different independently generated detector sets (running on different machines for example) can be used to achieve a higher detection rate for a single object. The failure rate decreases exponentially with the number of independent detector sets used.
- The individual detectors in the detector set can be run independently as well, for instance in a scheme with autonomous agents (such as the one presented in [1]), where each agent would contain one or a few detectors.

We think this distributability property is crucial because it allows each copy of the algorithm to use a unique set of detectors. Having identical protection algorithms can be a major vulnerability in large networks of computers because an intrusion at one site implies that all sites are vulnerable.

The negative detection method was introduced in [6]. Our current emphasis is on extending the theoretical basis of the method and addressing the important question of practicality, including (i) the feasibility of generating detectors, (ii) determining how to choose parameters for the algorithm, and (iii) discussing the

To appear at the 1996 IEEE Symposium on Security and Privacy.

implications for real-world problems. This paper presents algorithmic and analytical results that enable us to address these questions. The original algorithm simply generated random detectors and then censored the ones that matched self strings (according to a predefined matching rule). We present two new algorithms, both of which are more efficient at generating detectors. Section 2 gives an overview of these algorithms, their time and space complexity and some of the formulas and parameter bounds derived from them. Section 3 compares results obtained with the different algorithms and formulas, both on randomly generated files and on a real binary file. Section 4 touches on some of the practical issues, including guidelines on applying the method, how to choose the parameters, and the formulas involved. Section 5 presents conclusions and outlines areas for further study.

Currently we restrict ourselves to the case where both self strings and detectors are strings of length l over an alphabet of size m. In this paper, the alphabet is usually binary (m=2). Self consists of an unordered set of these strings (a multiset, because strings can occur more than once). Figure 1 shows the relevant sets of strings and how they relate to each other. The goal of our method is to find a detector set R that matches as many of the nonself strings in N as possible, without matching any of the self strings in S. We define the *failure probability* P_f as the probability that a random nonself string will not be matched by any of the detectors in R. We further

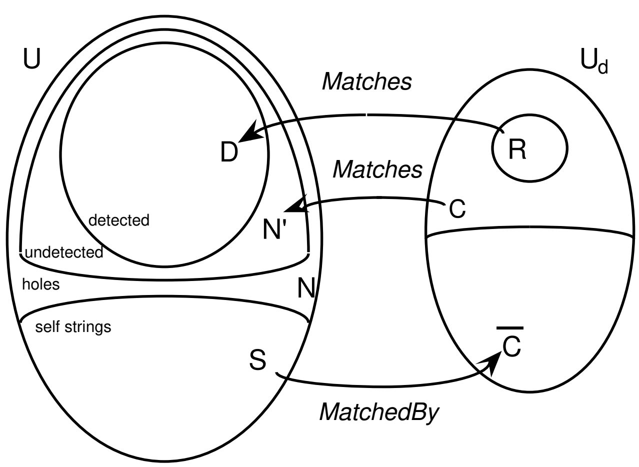

Figure 1: Sets of strings and their relations. String space U and detector space U_d are drawn separately for clarity, even though $U=U_d$ for this paper. P is *MatchedBy* Q iff Q contains all the detectors matching any string in P. Q *Matches* P iff P contains all the strings matched by any detector in Q. S: self strings; N: nonself strings; C: candidate detectors; R: detector repertoire chosen from C; N´: detectable nonself strings; D: detected nonself strings. Not indicated are the set of holes H=N-N´, and the set of undetected nonself strings F=N´-D.

define the *matching probability* P_m as the probability that a randomly chosen string and detector match according to the specified matching rule. To simplify the notation, we will write N_X for the size (i.e. cardinality) $|X|$ of a set of strings X. In particular, N_S is the size of the self set and N_R is the detector set size.

2. Detector generating algorithms

This section describes three different algorithms for generating detector sets: the original exhaustive generating algorithm and two new algorithms, based on dynamic programming, which run in linear time with respect to the size of the input. See [6] for an exposition of the exhaustive detector generating algorithm (2.1). For more details on the linear time algorithm (2.2), see [8] and [3]. This last report also covers the greedy algorithm (2.3) and the algorithm for counting the holes (2.4), including some examples and a derivation of the time and space complexities.

2.1. Exhaustive detector generating algorithm

This algorithm mirrors most closely the generation of T-cells in the immune system. Candidate detectors 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.

This algorithm requires generating a number of candidate detectors (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) [2]:

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

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

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

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

$$N_R \approx -\ln\left(P_f\right) / P_m \ . \qquad (2)$$

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 1) 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 (1) would indicate.

The time complexity of this algorithm is proportional to N_{R_0}, 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:

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

2.2. Linear time detector generating algorithm

The generate-and-test algorithm described above is inefficient because most of the candidate detector strings are rejected. However it does work for arbitrary matching rules. For specific matching rules we might be able to find a more efficient detector generating algorithm. 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 solve a counting recurrence for the number of strings unmatched by strings in S (candidate detectors, set C in Figure 1). In Phase II, we use the enumeration imposed by the counting recurrence to pick detectors randomly from this set of candidate detectors.

We will adopt the following notation:

s denotes a bit string.

$\hat{s}$ denotes s stripped of its first (leftmost) bit.

$s\cdot b$, where $b\in\{0,1\}$, denotes s appended with b. In particular, $\hat{s}\cdot b$ is s stripped of its first bit and appended with b.

A *template* of order r is a size l string consisting of l-r "blank" symbols (represented by asterices here) and r fully specified contiguous bits. In particular, a template $t_{i,s}$, is that template in which the r specified bits start at position i and are given by the r-bit string s. For example, with l=6, r=3, s=`010`: $t_{2,s}$=`*010**`.

A string (or template) *matches* a string if they are identical (no blanks) in at least r contiguous positions.

A right (left) *completion* of a template t is that template with all the blanks to the right (left) replaced by bits. For example: `*01011` is a valid right completion for `*010**`.

$(a,b]$ stands for the integer interval (a+1)...b.

Phase I: Solving the counting recurrence

For bit strings s of length r and for $1\le i\le(l-r+1)$, let $C_i[s]$ = the number of right completions of $t_{i,s}$ unmatched by any string in S. Each entry $C_i[s]$ in the array corresponds to an order r template $t_{i,s}$. i In essence, these templates enumerate all the possible ways two strings can match each other over r contiguous bits. In particular, for i=l-r+1, $t_{i,s}$ consists of l-r blanks, followed by r consecutive bits. There are no blanks to the right, so the only right completion of such a template is the template itself. Therefore C_{l-r+1}[s] will be zero if the template $t_{l-r+1,s}$ is matched in S, one otherwise:

$$C_{l-r+1}[s]=\begin{cases}0, \text{if } t_{l-r+1,s} \text{ is matched in } S\\ 1, \text{ otherwise}\end{cases}$$

For $1\le i<(l-r+1)$, we can calculate the number of unmatched right completions based on the number of unmatched right completions at i+1. If $t_{i,s}$ is directly matched in S, $C_i[s]$ is zero. Otherwise, we can subdivide the right completions of $t_{i,s}$ into those with a `0` bit directly following the r significant bits of s, and those with a `1` bit there. These are exactly the number of right completions for $\hat{s}\cdot 0$ and $\hat{s}\cdot 1$ respectively:

$$C_i[s]=\begin{cases}0, \text{ if } t_{i,s} \text{ is matched in } S\\ C_{i+1}[\hat{s}.0]+C_{i+1}[\hat{s}.1], \text{ otherwise}\end{cases}$$

For example, with l=6, r=3, suppose s_1 = `110100` is one of the strings in S, then the following templates are directly matched by this string: `110***`, `*101**`, `**010*` and `***100`. Therefore, C_1[`110`] = C_2[`101`] = C_3[`010`] = C_4[`100`] = 0. Suppose string s_2 = `100101` is also in S. The template `**110*` is not directly matched by s_1 nor s_2. However, because both `***100` and `***101` are matched in S (by s_1 and s_2 respectively), `**110*` will not have any unmatched right completions either: C_3[`110`] = C_4[`100`] + C_4[`101`] = 0. This can easily be verified: `**110*` has two right completions: `**1100` and `**1101`. The first one is matched by s_1, the second by s_2.

Phase II: Generating strings unmatched by S

Note that as the recurrence progresses from column C_{l-r+1}[.] of the C array to column C_2[.], the remaining blanks in the right completions are gradually filled up, and for C_1[.], the right completions are fully specified l-bit strings. Therefore, $C_1[s]$ denotes the number of unmatched l-bit binary strings starting with the r-bit binary string s. The total number of strings unmatched by S is

$$T=\sum_s C_1[s].$$

C_1[.] can be viewed as a partitioning of the space of unmatched strings into partitions of size $C_1[s]$ for each initial r-bit string s. Of all the unmatched strings starting with s, we know that $C_2[\hat{s}.0]$ have a 0 bit next, while $C_2[\hat{s}.1]$ have a 1 bit next, so C_2[.] can be viewed as a further partitioning of this space. Similarly for C_3[.] to C_{l-r+1}[.]. After partitioning according to C_{l-r+1}[.], each partition consists of one single l-bit string. We can therefore impose an explicit numbering from 1 to T on the unmatched strings, based on the natural order of bit strings. Given this explicit numbering, we can generate N_R random integers in $\{1..T\}$ and retrieve the corresponding strings. For a number $k\in\{1..T\}$, we find the k^{th} unmatched string u_k in the following way:

First, do a binary search on C_1[.] to find s_1 such that

$$P_1=\sum_{s<s_1}C_1[s] \;<\; k \;\le\; Q_1=\sum_{s\le s_1}C_1[s].$$

All unmatched strings in $(P_1,Q_1]$ have s_1 as their leading r bits. The string u_k we are looking for is in the partition of unmatched strings numbered $(P_1+1)\ldots Q_1$, therefore the first r bits of u_k are given by s_1.

Now we can determine for each $i = 2\ldots(l-r+1)$ the bit at position (r+i-1) of u_k, by checking in which partition k falls. For example, to determine the bit at position r+1, we can partition the interval further into $(P_1, P_1 + C_2[\hat{s}_1.0]]$ and $(P_1 + C_2[\hat{s}_1.0], Q_1]$, corresponding to the strings with either a 0 or 1 bit coming next. We add a bit b_1=0 if k is in the first interval, b_1=1 bit if k is in the second one. We then set P_2 and Q_2 using:

$$P_i = \begin{cases} P_{i-1}, \text{ if } b_{i-1} = 0 \\ P_{i-1} + C_i[\hat{s}_{i-1}.0], \text{ if } b_{i-1} = 1 \end{cases}$$

and

$$Q_i = \begin{cases} P_{i-1} + C_i[\hat{s}_{i-1}.0], \text{ if } b_{i-1} = 0 \\ Q_{i-1}, \text{ if } b_{i-1} = 1 \end{cases}$$

Let $s_2 = \hat{s}_1 \cdot b_1$. k is now in the interval $(P_2, Q_2]$, which we can split up into intervals $(P_2, P_2 + C_3[\hat{s}_2.0]]$ and $(P_2 + C_3[\hat{s}_2.0], Q_2]$. Bit b_2, will be determined by whether k falls in the first or second interval. Further bit positions of u_k are determined similarly.

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:

$$\text{time: } O\big((l-r)\cdot N_S\big) + O\big((l-r)\cdot 2^r\big) + O\big(l \cdot N_R\big),$$

$$\text{space: } O\big((l-r)^2 \cdot 2^r\big).$$

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 exhaustive detector generating algorithm, which ran in time exponential to the size of the self set, but required essentially only constant space. The linear time algorithm still requires time and space exponential in the length r of the matching region, which may present a problem if we need to choose long strings (l) and matching regions (r).

2.3. Greedy detector generating 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 $C_{l-r+1}[.]$ to $C_1[.]$). We can also go through the strings from left to right, constructing a second array C' starting at $C'_1[.]$ and calculating the following levels using a similar recurrence relation as for C. Because $C_i[s]$ represents the number of nonmatching right completions for template $t_{i,s}$ and $C'_i[s]$ the number of nonmatching left completions, $D_i[s] = C_i[s] \times C'_i[s]$ represents the number of unmatched fully specified bit strings corresponding to this template.

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.

Phase I: Generate *D* arrays: D_S and D_R

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. The D_S array tells us which templates we are allowed to choose from 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. For each new detector to be generated, we then try to select the templates matching the most unmatched strings. We have to update this array D_R each time a new detector is generated, so it will generally be cheaper to just keep the C_R and C'_R arrays around and update these incrementally. Because we begin with R being empty, we can initialize C_R and C'_R with their maximum values: $C_{R,i}[s] = 2^{(l-r+1-i)}$ and $C'_{R,i}[s] = 2^{(i-1)}$ (corresponding to $D_{R,i}[s] = 2^{(l-r)}$).

For each new detector, we search 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 have to update the C_R and C'_R arrays to reflect that a new detector has been added to R. We can do this incrementally, by setting those entries in C_R and C'_R to zero that directly match the detector, and recalculating the appropriate entries.

We repeat this process of picking a detector and updating C_R, C'_R and 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. This gives us an absolute lower bound on the number of detectors needed [12]:

$$N_R \geq \left(1-P_f\right)/P_m . \tag{3}$$

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:

$$N_R \propto 2^r \cdot \left(1-2^{-r}\right)^{N_S} . \tag{4}$$

Because we need to update the template array each time a new detector is generated, the time complexity is quite a bit higher than for the previous algorithm, although the space complexity is of the same order:

$$\text{time}: O\big((l-r)\cdot 2^r \cdot N_R\big),$$

$$\text{space}: O\big((l-r)^2 \cdot 2^r\big).$$

2.4. Counting the holes

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 nonself strings, called "*holes,*" for which it is impossible to generate valid detectors. For example, if S contains two strings s_1 and s_2 that match each other over (r-1) contiguous bits, they can 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:

$$\begin{array}{ll} s_1: & a_1 \ldots a_k \; b_{k+1} \ldots b_{k+r-1} \; c_{k+r} \ldots c_l \\ s_2: & a'_1 \ldots a'_k \; b_{k+1} \ldots b_{k+r-1} \; c'_{k+r} \ldots c'_l \\ & \qquad\qquad \Downarrow \\ h_1: & a_1 \ldots a_k \; b_{k+1} \ldots b_{k+r-1} \; c'_{k+r} \ldots c'_l \\ h_2: & a'_1 \ldots a'_k \; b_{k+1} \ldots b_{k+r-1} \; c_{k+r} \ldots c_l \end{array}$$

where a_i, a'_i, b_i, c_i, and c'_i are single bits.

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). In fact, almost all practical matching rules with a fixed matching probability can be expected to exhibit holes [4, 5]. However, we can eliminate holes altogether by choosing a matching rule with a variable matching radius, such that potential holes are filled by detectors with high specificity.

Because holes will never be detected by any detector, they imposes a lower bound on the failure probability P_f we can achieve, whether we use a single set of detectors or several independent detector sets generated for the same matching rule. It is therefore advisable in a distributed setting to choose a different matching rule (or simply different parameters) for each machine, so each will have a different set of holes which are likely covered by some other machines. On the other hand, we can take advantage of the position of these holes to provide a certain level of noise tolerance in our detection method: because holes are close to self strings, we might not really care if they go undetected. 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.

Running the greedy algorithm until all valid detectors have been used up would tell us exactly how many nonself strings cannot be detected. In [3] 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. Its space and time complexities are similar:

$$\text{time}: O\big((l-r)\cdot N_S\big) + O\big((l-r)\cdot 2^r\big),$$

$$\text{space}: O\big((l-r)^2 \cdot 2^r\big).$$

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 very minimum, we want the number of holes N_H to be small enough to allow the desired failure probability P_f: $N_H \leq P_f \cdot 2^l$. If we stick close to this upper bound on the allowed number of holes, almost all valid detectors will be needed to cover the very last nooks and crannies of the detectable nonself string space. If the number of holes is much smaller than this, 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 may want to use

$$N_S \leq 2^r . \tag{5}$$

L_S (a)	N_S (b)	l (c)	r (d)	P_m (e)	indep. P_f=0.1 (f)	greedy complete (g)	estimate complete (h)	greedy P_f=0.1 (i)	optimal P_f=0.1 (j)	entropy P_f=0.1 (k)	number of holes (l)
500B	250	16	10	0.00391	589	793	802	374	230	52	634
			9	0.00879	262	320	314	212	102		4438
			8	0.01953	118	88	96	(*)	46		21076
1KB	250	32	11	0.00562	410	1796	1813	313	160	26	2649
			10	0.01172	196	821	802	153	77		24911
			9	0.02441	94	378	314	76	37		2150714
			8	0.05078	45	89	96	(*)	18		5.1815e+08
	500	16	11	0.00171	1347	1610	1604	829	526	104	882
			10	0.00391	589	632	628	445	230		3854
			9	0.00879	262	155	193	(*)	102		24937
2KB	500	32	12	0.00269	856	3584	3625	650	335	52	4787
			11	0.00562	410	1668	1604	316	160		52318
			10	0.01172	196	772	628	159	77		2420706
			9	0.02441	94	201	193	(*)	37		4.6564e+08
	1000	16	12	0.00073	3154	3204	3207	1856	1233	208	1353
			11	0.00171	1347	1258	1257	1233	526		5428
			10	0.00391	589	340	385	(*)	230		23933
4KB	1000	32	13	0.00128	1799	7144	7251	1352	703	104	8475
			12	0.00269	856	3304	3207	659	335		85798
			11	0.00562	410	1516	1257	328	160		3991790
			10	0.01172	196	378	385	(*)	77		5.2296e+08

Table 1: Repertoire sizes and number of holes for different configurations. (a): file size in bytes, (b), (c), (d): parameters chosen for the matching rule. (e): corresponding matching probability P_m for the r-contiguous-bit matching rule. (f): N_R calculated according to formula (2), for independent detectors. (g): size of *complete* detector repertoire generated with the greedy algorithm. (h): estimate for (g) based on formula (4). (i): N_R generated by greedy algorithm until $P_f \leq 0.1$ ((*) means that $P_f \leq 0.1$ could not be achieved). (j): lower bound on (i) based on formula (3). (k): entropy-based lower bound on (h).[1] (l): number of holes present.

3. Results and analysis

In this section we use the algorithms and analysis from section 2 to explore which parameter settings are practical. Subsection 3.1 looks at results obtained for relatively small randomly generated self sets, and draws some conclusions from comparisons between different algorithms and formulas. Subsection 3.2 looks at a much larger real-world example and examines how the failure probability scales with detector set size and matching length r.

3.1. Repertoire sizes using different algorithms

Table 1 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 a 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 (f) and (i), we see that the greedy algorithm generates a detector set that is from 8% to 41% smaller than the size predicted for the 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 (i). For the linear time and the exhaustive algorithm, there is no guarantee that a detector set of the size indicated by (2) will achieve the specified P_f. This is not a problem for the greedy algorithm, because we can continue generating detectors until P_f exceeds the specification.

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

[1] References [4, 5] show that for a set of independent self strings, the following is a lower bound on the number of detectors needed: $N_R \geq N_S \cdot \log_2\left(1/P_f\right) / l \cdot \log_2(m)$ (6). This bound is based on the amount of information that needs to be stored in R about S.

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 and the maximum detection rate (complete repertoire).

Both (3) and (6) (columns (j) and (k) in Table 1) 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 (j) is based on optimal spacing of the detectors, which is precisely what the greedy algorithm tries to achieve, so we could view column (j) 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., (j)$\leq$(k)), it is unable to do so (entry in (i) is "(*)") because the number of holes in the string space (column (l)) is too large with respect to the desired P_f. This suggests that

$$\left(1 - P_f\right) \big/ P_m \geq \frac{N_S \cdot \log_2\left(1/P_f\right)}{l \cdot \log_2(m)} \tag{7}$$

is an interesting lower 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 space of candidate detectors becomes smaller and the number of holes due to interaction between self strings gets larger. For the smallest r for which we can construct an adequate detector set R, 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 detector 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 r smaller and still have an acceptable detector set. This is exactly opposite to what we would expect if we wanted to keep the matching probability P_m constant.

3.2. P_f versus N_R for a real data file

Figure 2 shows how P_f varies with N_R for a binary file (GNU emacs v19.25.2 SGI binary, 3.2MB). The data

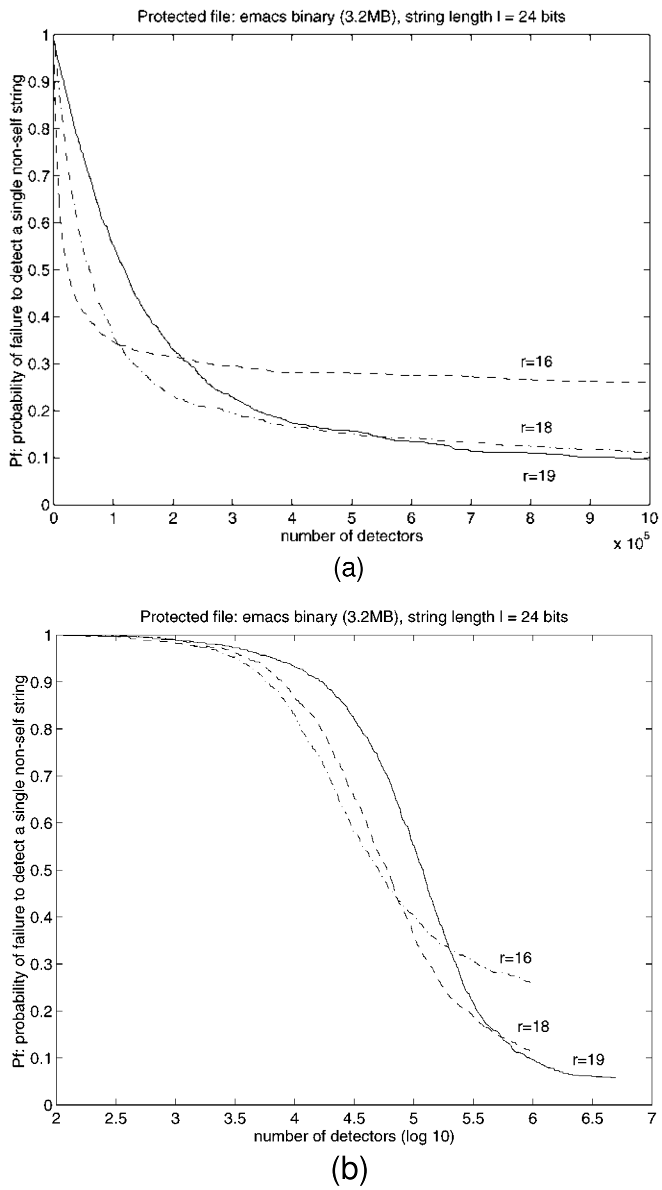

Figure 2: P_f versus N_R for a binary file. (a): linear scale for N_R; (b): log scale for N_R.

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 *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.

The figure shows a sharp drop in the failure probability for the first couple of hundred thousand detectors. This is due to the probabilistic nature of coverage of the string space by detectors. We would expect this decline to be even more pronounced if we were to generate the detectors using the greedy algorithm, because then the detectors chosen first are those which cover as many nonself strings as possible. The decline in P_f is sharper for smaller values of r, and therefore for larger matching probabilities, because each detector matches a larger fraction of string space, so most of the space is covered by a small number of detectors. However, as the detector set size increases, P_f levels off at a higher level for smaller r because there are more holes.

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 want to exploit the fact that the failure probability decreases exponentially with an increasing number of machines, each of which is running its own detector set, we may already be satisfied with $P_f \approx 0.5$. Figure 2 shows that this can be achieved with a repertoire as small as 30,000 to 120,000 detectors. This is quite an achievement given that the original self file contains about one million strings.

4. Summary of formulas, practical issues and rules of thumb

This section gives a summary of the appropriate formulas for the r-contiguous-bits matching rule, and gives step-by step instructions on how to choose the matching rule parameters for a real data stream.

4.1. 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 $P_m \propto m^{-r}$ [11, 13]. 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, [6] 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.

4.2. 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:

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

Table 2 illustrates some values for L_S and the corresponding lower bounds for l.

L_S	384	2K	10K	49K	229K	1M	4.7M	21M	92M	403M
l	6	8	10	12	14	16	18	20	22	24

Table 2: L_S versus lower bound for l

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, if the data to be protected is a database consisting of a series of 4-byte records, then a multiple of four 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. Choosing the matching length

To keep the number of holes low, we want to pick r such that $N_S \leq 2^r$ (5). 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 1, formula (7) 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:

$$\left(1-P_f\right)/P_m \geq \frac{N_S \cdot \log_2\left(1/P_f\right)}{l \cdot \log_2(m)} \ . \tag{7}$$

Finally, after a minimum value for r has been chosen, we may want to count the exact number of holes present in the string space. If this exceeds the acceptable number of holes of $N_H \leq P_f \cdot 2^l$, we choose a smaller matching probability (i.e. larger r) and try again. In general, because we don't 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.

4.4. Choosing the number of detectors

Above 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 (2):

$$N_R \approx -\ln\left(P_f\right)/P_m \ . \tag{2}$$

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 can be quite a bit larger than the value indicated by (2). 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):

$$N_R \geq \left(1-P_f\right)/P_m \ . \tag{3}$$

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 may be hard to pick detectors that are spread apart optimally. Note that if we have chosen r according to formula (7), using this lower bound for N_R will automatically imply that we also satisfy the entropy lower bound (6). 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. (4) provides a fairly close estimate for the size of the complete detector set for independent self strings:

$$N_R \propto 2^r \cdot \left(1-2^{-r}\right)^{N_S} \ . \tag{4}$$

4.5. 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 needs to be checked against the entire detector set. Other, more conventional, change-detection algorithms may be more appropriate in this case.
- Intermittent checking: every so often a small number of strings can be checked against a small number of detectors chosen at random from the detector set. This relies on the probabilistic nature of this change-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 [1]) 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 can be used to protect a whole network of computers.

5. Conclusion

The linear time algorithm has made it practical to construct efficient detector sets for large data sets. The space requirements for this construction algorithm are substantial. However, this space is only needed once, to calculate the detector set, and can be discarded afterwards. The greedy algorithm allows us to sacrifice some of the speed of detector generation in exchange for a more compact detector set and a failure probability guaranteed to be below the acceptable P_f. 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 change-detection method it can supplement 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:

- It might be possible to derive more exact formulas for non-random data by looking at some measures of the self strings (entropy, average number of nonzero entries in the template arrays, number of holes etc.)
- It is possible to construct 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.
- 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?
- The effect of using negative detection as opposed to positive detection is not very well understood. More research would be needed to clarify this issue.

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. This work is supported by grants from the National Science Foundation (grant IRI-9157644), Office of Naval Research (grant N00014-95-1-0364), Interval Research Corporation, General Electric Corporate Research and Development, and Digital Equipment Corporation.

References:

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

[2] 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.

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

[4] 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.

[5] P. D’haeseleer, “An Immunological Approach to Change Detection: Theoretical Results”, accepted to the 9th IEEE Computer Security Foundations Workshop, 1996.

[6] 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.

[7] S. Forrest, S. A. Hofmeyr, A. B. Somayaji and T. A. Longstaff, “A sense of self for UNIX processes”, submitted to the 1996 IEEE Symposium on Security and Privacy, 1995.

[8] 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.

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

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

[11] J. K. Percus, O. E. Percus and A. S. Perelson, “Probability of Self-Nonself discrimination” in Theoretical and Experimental Insights into Immunology, 1992.

[12] 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.

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