Fast Multipattern Search Algorithms for Intrusion Detection

Josué Kuri* Gonzalo Navarro†

Abstract

We present new search algorithms to detect the occurrences of any pattern from a given pattern set in a text, allowing in the occurrences a limited number of spurious text characters among those of the pattern. This is a common requirement in intrusion detection applications. Our algorithms exploit the ability to represent the search state of one or more patterns in the bits of a single machine word and update all the search states in a single operation. We show analytically and experimentally that the algorithms are able of fast searching large sets of patterns allowing a wide number of spurious characters, yielding about a 75-fold improvement over the classical algorithm.

1 Introduction

A major challenge in intrusion detection is the effective detection of attacks as they are occurring, a problem known as *on-line* intrusion detection. Current research trends aim to a simplified representation of the problem in order to improve efficiency and performance. Pattern matching techniques are getting major attention as potential solutions because they have solved analog problems in domains as computational biology and information retrieval. In intrusion detection, pattern matching algorithms have been proposed as search engines in two different intrusion detection models. One is based in the concept of *state transition analysis* [9, 12] and the the other uses the *computer immunology* approach proposed in [8].

We give an example to illustrate how the pattern matching algorithms presented below can be used to solve an intrusion detection problem. Auditable events in the target system can be seen as letters of an alphabet Σ and the audit trail as a large *string* of letters in Σ^* (i.e. the text). The sequences of events representing attacks to be detected are then *substrings* (i.e., patterns) to be located in the main *string*. Potential attackers may introduce spurious events among those that represent an actual attack in order to disperse their evidence, so a limited number of spurious letters must be allowed when searching the pattern. We are interested in detecting a *set* of possible attacks at the same time. This intrusion detection problem can be regarded as a particular case of the multiple approximate pattern matching problem, where *insertion* in the pattern is the only allowed edit operation.

There is a wide variety of audit facilities, covering different sources of potential attacks. A common property of these facilities is that they generate huge amounts of audited data in a short time, in the order of several millions of events per hour for large computing infrastructures. On the other hand, attacks are typically short sequences of no more than 8 commands. Finally, the number of known attacks to system vulnerabilities is so large [11] that it is a common request for an intrusion detection system to search attack sets of more than 100 elements. Under the approach of mapping events to letters, the typical values for σ may vary from 60 to 80, depending on the number of different auditable events in a particular system.

*Ecole Supérieure d'Electricité, Avenue de la Boulaie – BP 28, 35511 Cesson Sévigné – France. jkuri@enst.fr. Work supported by CONACyT grant # 122688.

†Dept. of Computer Science, Univ. of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Work developed in part while the author was at postdoctoral stay at Institut Gaspard Monge, Univ. de Marne-la-Vallée, France. Supported in part by Fondecyt grant 1-990627 and Fundación Andes.

With respect to the typical k values, it is important to avoid false matches (i.e. triggering unnecessary alarms for sequences that do not really represent an attack because k is too large) and to avoid missing true attacks. Empirical values of k are typically between 6 and 10.

We formalize the above problem as follows. Our text, $T_{1..n}$, is a sequence of n characters from an alphabet Σ of size σ. Our pattern, $P_{1..m}$ is a sequence of m characters from the same alphabet. We want to report all the text positions that match the pattern, where at most k insertions between characters of P are allowed in its occurrence in T. We call $\alpha = k/m$ the "error level".

A lot of work has been carried out on an extended version of this problem (called *search allowing k differences*), where not only insertions, but also deletions and replacements are allowed. In a recent survey [17] four approaches are distinguished to search with k differences: dynamic programming, automata, filtering and bit-parallelism.

However, very little has been done to search with k insertions. Not all the algorithms for k differences can be successfully simplified for our restricted case. The most naive algorithm (which we show in Section 2) is a simplification of the classical dynamic programming solution for k differences, and the same $O(mn)$ search time is maintained. We consider this complexity as the reference point for further improvements. Automata approaches can be adapted with similar efficiency results: $O(n)$ search time but impractically high preprocessing and space requirements (exponential in m or k).

Filtering approaches are very successful to search with k differences and are generally based in the concept that some pattern substrings must match even in inexact occurrences. This is also our case: for example, if k insertions are allowed in the matches then at least one pattern piece of length $\lfloor m/(k+1) \rfloor$ must be found inside every occurrence. Hence we can search for those pieces and use a more expensive algorithm only in the text areas surrounding such occurrences of pattern pieces. However, in most applications of the k differences problem it is common that k is much smaller than m and therefore reasonably long pattern pieces have to be found. Instead, in intrusion detection k is normally large (in many cases $k > m$) and therefore filtering approaches are ineffective in general.

The most promising approach seems to be bit-parallelism (which we explain in Section 3), because the simplicity of the k insertions model allows devising faster algorithms. In particular, we present in Section 4 a search algorithm with time complexity $O(nm \log(k)/w)$ where w is the length in bits of the computer word. This is $O(n)$ for reasonably short patterns. Moreover, it is better than previous bit-parallel algorithms for the k differences, which were $O(nmk/w)$ time [20, 6], but it is worse than a later development [14] which achieves $O(mn/w)$. Interestingly, this last approach cannot be adapted to our problem, but that of [20] can be adapted at the same $O(nmk/w)$ time cost. A related but different problem, called "episode matching", is to find the pattern with the minimum number of insertions. Many algorithms are presented in [7], where the best one needing space polynomial in m takes $O(mn/\log m)$ time.

A special requirement of our application is the need for multipattern search. That is, we are given r patterns $P^1 ... P^r$ and we have to report all their occurrences. Very little work has been done on multipattern search for the k differences problem [13, 4, 15, 5, 16]. In Sections 5 and 6 we adapt two of those approaches to the k insertions problem. The first one obtains a speedup of $\sigma\alpha^\alpha/(1+\alpha)^{1+\alpha}$ (where $\alpha = k/m$) over the basic bit-parallel algorithm of Section 4. This speedup is larger than 1 for $\alpha < \sigma/e - 1$. The second one obtains a speedup of $w/\log_2(m+k)$, but it works well only for $m + k < \sigma$, i.e. short patterns.

All the algorithms mentioned form the first nontrivial solutions to the k insertions problem, both for single and multiple patterns. In Section 7 we show some experimental results about the practical performance of the algorithms. For typical cases our bit-parallel version outperforms the classical dynamic programming by a factor of 3, while the multipattern filters obtain a 25-fold speedups. The

net result is a 75-fold speedup over a classical approach.

2 The Insertion Distance and a Naive Algorithm

Our problem can be modeled using the concept of *insertion distance*. The insertion distance from a to b, denoted $id(a,b)$, is the number of insertions necessary to convert a into b. We say that $d(a,b) = \infty$ if this is not possible. Clearly, $id(a,b) = |b| - |a|$ if a is a subsequence of b, and ∞ otherwise.

A more interesting definition arises when we search for a pattern P in a text T allowing insertions. At each text position $j \in 1..n$ we are interested in the minimum number of insertions needed to convert P into some suffix of $T_{1..j}$. This is defined as

$$lid(P, T_{1..j}) \quad = \quad \min_{j' \in 1..j} id(P, T_{j'..j})$$

The search problem can therefore be formalized as follows: given P, T and k, report all text positions j such that $lid(P, T_{1..j}) \leq k$.

An immediate solution to the problem comes from adapting an algorithm for k differences [19]. A vector of values C_i ($i \in 0..m$) is updated for each new text character T_j. The invariant is that, after processing text position j, $C_i = lid(P_{1..i}, T_{1..j})$. Therefore, we report all text positions j satisfying $C_m \leq k$. Initially (for $j = 0$) we have $C_0 = 0$ and $C_i = \infty$ for $i > 0$. When reading the text character T_j the C_i values are updated to the new C'_i values using the formula

$$C'_i \quad = \quad \text{if } (P_i = T_j) \text{ then } \min(C_{i-1}, C_i + 1) \text{ else } C_i + 1 \tag{1}$$

which has the following rationale: if the new text character T_j does not match P_i, then we keep the previous match of P_i in a suffix of $T_{1..j-1}$ (the cost is C_i) and add an insertion to reflect that undesired last character T_j. If, on the other hand, the new text character matches P_i then we have also the choice of using it and matching $P_{1..i-1}$ with the best suffix of $T_{1..j-1}$ (the cost is C_{i-1}).

This algorithm is $O(mn)$ time and $O(m)$ space.

3 Bit-parallelism

Bit-parallelism is a technique of common use in string matching [2], firstly proposed in [1, 3]. The technique consists in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most w, where w is the number of bits in the computer word. Since in current architectures w is 32 or 64, the speedup is very significant in practice (and improves with technological progress). In order to relate the behavior of bit-parallel algorithms to other works, it is normally assumed that $w = \Theta(\log n)$, as dictated by the RAM model of computation. We prefer, however, to keep w as an independent value.

We introduce now some notation we use for bit-parallel algorithms.

- The length of the computer word (in bits) is w.
- We denote as $b_s...b_1$ the bits of a mask of length s. This mask is stored somewhere inside the computer word. Since the length w of the computer word is fixed, we are hiding the details on where we store the s bits inside it. We give such details when they are relevant.
- We use exponentiation to denote bit repetition (e.g. $0^3 1 = 0001$).

– We use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, " ^ " is the bitwise-xor and "$\sim$" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right, i.e. $b_s b_{s-1} ... b_2 b_1 << r = b_{s-r} ... b_2 b_1 0^r$. The shift-right, ">>" moves the bits in the other direction. Finally, we can perform arithmetic operations on the bits, such as addition and subtraction, which operates the bits as if they formed a number. For instance, $b_s ... b_x 10000 - 1 = b_s ... b_x 01111$.

Many text searching algorithms can be seen as implementations of clever automata (classically, in their deterministic form). Bit-parallelism has since its invention became a general way to simulate simple non-deterministic automata instead of converting them to deterministic. It has the advantage of being much simpler, in many cases faster (since it makes better usage of the registers of the computer word), and easier to extend to handle complex patterns than its classical counterparts. Its main disadvantage is the limitations it imposes with regard to the size of the computer word. In many cases its adaptations to cope with longer patterns are not so efficient. For our application, in particular, bit-parallelism seems to be a very promising approach.

4 A Bit-parallel Simulation

We show now how can we pack the C_i values of Section 2 in the bits of a computer word to speed up the search. Only the values from zero to $k+1$ are of interest, since if a C_i value is larger than $k+1$ then the outcome of the search is the same if we replace it by $k+1$. Therefore, we use $\ell = \lceil \log_2(k+1) \rceil$ bits to hold each C_i value, plus an extra overflow bit whose purpose is made clear shortly.

Taking minima in parallel is not impossible, but it is difficult. We show that the update formula (1) can be modified to avoid taking minima. First note that $C_{i-1} \leq C_i + 1$. That is, $lid(P_{1..i-1}, T_{1..j}) \leq lid(P_{1..i}, T_{1..j}) + 1$. This is clear, since any match of $P_{1..i}$ against a suffix of $T_{1..j}$ can be converted into a match of $P_{1..i-1}$ just by removing the alignment of P_i and considering it as an extra insertion (the +1). Hence the best alignment must be at most of that cost. Therefore, Eq. (1) is equivalent to

$$C'_i \quad = \quad \text{if } (P_i = T_j) \text{ then } C_{i-1} \text{ else } C_i + 1$$

which we now parallelize. We precompute a table $B : \Sigma \to \{0,1\}^{m(\ell+1)}$, defined as

$$B[c] \quad = \quad 0\ b(c, P_m)\ \ 0\ b(c, P_{m-1})\ \ \cdots\ \ 0\ b(c, P_2)\ \ 0\ b(c, P_1)$$

where $b(c,c) = 1^\ell$ and $b(c,c') = 0^\ell$ for $c \neq c'$. That is, $B[c]$ has m chunks of zeros or ones, indicating which pattern positions match character c. The idea is to use $B[c]$ to implement the test $(P_i = T_j)$, assigning C_{i-1} where it has ones and leaving $C_i + 1$ where it has zeros.

The state of the search is kept in a bit mask D, composed of m chunks of ℓ bits each (plus the overflow bit), so that the i-th chunk stores the current C_i value, i.e.

$$D \quad = \quad 0\ [C_m]_\ell\ \ 0\ [C_{m-1}]_\ell\ \ \cdots\ \ 0\ [C_2]_\ell\ \ 0\ [C_1]_\ell$$

where $[x]_\ell$ is the number x represented in ℓ bits in the usual way (right-aligned). Note that C_0 is not represented because it is always zero. In principle, the update formula could be as simple as

$$D' \quad = \quad (B[T_j]\ \&\ (D << (\ell+1))) \mid (\sim B[T_j]\ \&\ (D + (0^\ell 1)^m))$$

where $B[T_j]$ is being used to select between $(D << (\ell+1))$ (which puts the previous value C_{i-1} at the i-th chunk) and $(D + (0^\ell 1)^m)$ (which adds 1 to the current C_i values). In particular, the left shift brings zero bits to the first chunk C_1, which is adequate since $C_0 = 0$. The problem with this scheme is that the C_i values could surpass the barrier of $k+1$.

To overcome the problem we use the overflow bit. We let the C_i values grow over $k+1$ provided they fit in ℓ bits. As soon as they overflow, the overflow bit will be set. At this point, we subtract one to them. The easiest way to subtract one to all the C_i values whose overflow bit is set is to isolate the overflow bits, shift them ℓ positions to the right and subtract the mask from D.

The final problem is how to determine the text positions that match. In the dynamic programming version we simply check $C_m \leq k$. In the bit-parallel version the C_m value corresponds to the highest bits, and therefore we can numerically compare the whole bit mask D against $[k]_\ell 1^{(\ell+1)(m-1)}$, which avoids any additional bit shift or masking. We also want to report only text positions that end a genuine match, i.e. such that the last text character matches the last pattern character. Otherwise we would be reporting trivial extensions of previously found matches. This can be determined by looking at the m-th chunk of $B[T_j]$. The final algorithm is shown in Figure 1.

```
Search (T,n,P,m,k)

        /* Preprocessing */
  ℓ ← ⌈log2(k+1)⌉
  for c ∈ Σ do B[c] ← 0^{m(ℓ+1)}
  for i ∈ 1..m do B[P_i] ← B[P_i] | 0^{(m-i)(ℓ+1)} 01^ℓ 0^{(i-1)(ℓ+1)}
        /* Searching */
  for j ∈ 1..n
     Ds ← D << (ℓ+1)
     D ← D + (0^ℓ 1)^m
     D ← D − ((D >> ℓ) & (0^ℓ 1)^m)
     D ← (B[T_j] & Ds) | (∼ B[T_j] & D)
     if (D ≤ [k]_ℓ 1^{(ℓ+1)(m-1)}) and ((B[T_j] & 01^ℓ 0^{(m-1)(ℓ+1)}) ≠ 0^{m(ℓ+1)})
        then report a match ending at j
```

Figure 1: The bit parallel algorithm. All the constants and repeated expressions are of course precomputed.

If the bits of the simulation do not fit in the computer word we set up as many computer words as needed. Since each one is updated in $O(1)$ time per text character, the total complexity is $O(nm\log(k)/w)$. For short patterns (i.e. $m\log k = O(w)$) this is $O(n)$.

5 A Multipattern Filter

As already noted in [4], the ability of bit-parallel algorithms to allow classes of characters can be used to build multipattern filters. Imagine that the pattern is not a sequence of letters but a sequence of *classes* of letters. A letter a is said to match P at position i if $a \in P_i$, i.e. if it belongs to the corresponding class.

If we have a pattern which is a sequence of classes of characters, the algorithm of Section 4 can still be used, just by changing the preprocessing phase. The idea is that we can redefine the b function to

$$b(c, c') \quad = \quad 1^\ell \text{ if } c \in c' \text{ and } 0^\ell \text{ otherwise}$$

which is equivalent to changing the third line in the preprocessing of Figure 1 to

$$\texttt{for } i \in 1..m \texttt{ do for } c \in P_i \texttt{ do } B[c] \leftarrow B[c] \mid 0^{(m-i)(\ell+1)}01^\ell 0^{(i-1)(\ell+1)}$$

that is, we allow the value of C_{i-1} to pass to position i for any character c that matches pattern position i.

Consider now that we have r patterns $P^1...P^r$ of the same length m (otherwise we truncate them to the shortest one). From them we generate a much more relaxed pattern with classes of characters, which we call the *superimposition* of $P^1...P^r$. This is defined as

$$P \quad = \quad \{P_1^1, ..., P_1^r\}\ \{P_2^1, ..., P_2^r\}\ ...\ \{P_m^1, ..., P_m^r\}$$

which necessarily matches when one of the P^j matches, although the converse is not true. For instance, if we search `"abcd"` and `"adcc"` then the superimposed pattern is `"{a}{b,d}{c}{d,c}"`, and the text window `"adcd"` will match with *zero* insertions, even if it is not in the set of patterns.

To make this more clear, consider the NFA of Figure 2. The rows represent the number of insertions. The first one zero, the second one 1, and so on. Each column represents a pattern prefix. Horizontal arrows represent matching a pattern letter with a text letter, while vertical arrows represent skipping a text letter (since we advance in the text but not in the pattern, and increment the number of insertions). The initial state has a self-loop to allow any text position to start a match. State in row $s \in 0..k$ and column $i \in 0..m$ is active each time a suffix of the text read matches $P_{1..i}$ with s insertions, so each time the lower right state is active we have an occurrence of the pattern in the text with at most k insertions.

Indeed, it can be proved that if state (s, i) is active then any state (s', i) with $s' > s$ is active as well, and that the C_i value of Section 2 is the minimum row of an active state at NFA column i. Therefore, our bit-parallel simulation can be thought of as a mechanism to pack the information of this NFA in bits and to simulate the transitions that occur along the arrows of the automaton.

The NFA of Figure 2 has been built for the superimposition of `"abcd"` and `"adcc"`. For instance, the arrows in the second column can be traversed either by the letter `"b"` or `"d"`. Clearly this automaton will recognize any occurrence of the two patterns, and some others as well.

Therefore, the technique consists in superimposing the search patterns, search the superimposition with the same algorithm of Section 4, and then checking the areas where the superimposition is found for the presence of any of the individual patterns. That is, each time the algorithm finds the superimposed pattern at text position j, we check each of the patterns separately (with the same algorithm) in the text area $T_{j-m-k+1..j}$. A similar idea was proposed in [4, 5, 16] for the k-differences problem.

To avoid re-verification due to overlapping areas, we keep track of the last position verified and the state of the verification algorithm. If a new verification requirement starts before the last verified position, we start the verification from the last verified position, avoiding to re-verify the preceding area.

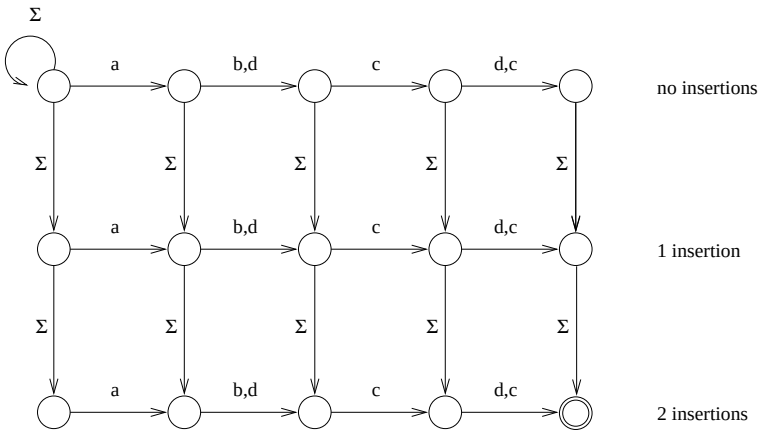

Figure 2: An NFA to search the superimposition `"abcd"` and `"adcc"` allowing 2 insertions.

5.1 Hierarchical Verification

Instead of checking one by one the patterns for each occurrence of the superimposed pattern, we can build up a hierarchy of superimpositions [18, 16]. Imagine that $r = 8$. Then we build, at preprocessing time, the superimposition of the 8 patterns, called $P^{1..8}$. We consider this the root of a binary tree, whose two children are $P^{1..4}$ and $P^{5..8}$, i.e. they superimpose only 4 patterns. The first one has two children $P^{1..2}$ and $P^{3..4}$, and so on. Finally, the leaves of the tree are the actual patterns. If r is not a power of two we build the tree as balanced as possible. Figure 3 illustrates.

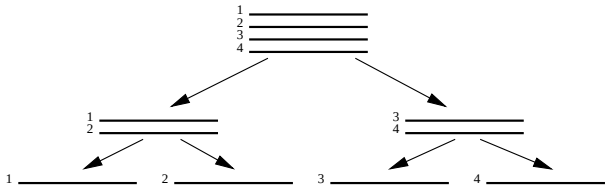

Figure 3: Hierarchical verification for 4 superimposed patterns.

We search $P^{1..8}$ in the text. When it is found, we do not check immediately all the leaves P^1 to P^8, but just its two children $P^{1..4}$ and $P^{5..8}$. It is possible that, despite that the root was found, none of the two children appears (and therefore no leaf can appear as well). So we can avoid performing 8 verifications at the cost of 2. Of course it is also possible that one and even both of the children appears in the text area and then their children have to be checked in turn until the leaves are found (and these are actually reported). In particular, if a leaf appears it will require all the path of verifications. However, as we show next, hierarchical verification pays off.

5.2 Analysis

Superimposing r patterns gives of course better search time because only one search is carried out instead of r. On the other hand, however, it makes necessary to check the occurrences of the superimposed pattern for the presence of the actual ones. Moreover, the probability of matching raises as we superimpose more patterns, because up to r characters of the alphabet match each pattern position.

We start by giving an upper bound on the matching probability of a random pattern of length m at a random text position, with up to k insertions. Consider a random text position j. The pattern P appears with k insertions at a text position ending at j if and only if the text window $T_{j-m-k+1..j}$ contains the m pattern letters in order. The window positions that match the pattern letters can be chosen in $\binom{m+k}{m}$ ways. Those letters are fixed but the other k can take any value. Therefore the probability that the text window matches the pattern with k insertions is at most

$$\binom{m+k}{m}\frac{\sigma^k}{\sigma^{m+k}} \quad = \quad \binom{m+k}{m}\frac{1}{\sigma^m}$$

where we are overestimating because not all the selections of window positions give different windows. For instance the pattern `"abcd"` matches in text window `"abccd"` with $k=1$ in two ways, but only one text window should be counted. In particular, our overestimation includes the case of $k' < k$ insertions, which is obtained by selecting the first $k-k'$ characters of the text window as insertions and distributing the k' remaining insertions in the remaining text window of length $m+k'$.

If we are given r patterns and superimpose them in groups of r', there are at most r' out of σ alphabet letters that will match each pattern position now. The net effect is that of dividing σ by r' in the formulas. If we consider that no hierarchical verification is used, then each match of the superimposed pattern triggers a verification of r' original patterns in a text area of width $m+k$. Therefore the total search cost is on average (assuming that the patterns fit in a computer word)

$$\frac{nr}{r'}\left(1+\binom{m+k}{m}\frac{(m+k)r'}{(\sigma/r')^m}\right) \quad = \quad nr\left(\frac{1}{r'}+\binom{m+k}{m}\frac{(m+k)r'^m}{\sigma^m}\right)$$

Assume now that we use hierarchical verification. In this case, 2 searches with $r'/2$ patterns are triggered for each occurrence of the superimposed pattern. For each occurrence of those superimpositions of $r'/2$ patterns we will have to check a text window with 2 patterns superimposing $r'/4$ original patterns, and so on. Abstracting from the mechanism we use to find the nodes of the tree of superimpositions, we have that in total, in the hierarchy there are 2^i groups of $r'/2^i$ patterns, for $i = 0..\log_2(r')-1$. Each such group matches with probability $\binom{m+k}{m}/(\sigma 2^i/r')^m$, and each match costs the verification of a window of length $m+k$ for other two patterns. The total verification cost is

$$\binom{m+k}{m}\frac{2(m+k)r'^m}{\sigma^m}\sum_{i=0}^{\log_2(r')-1}\frac{2^i}{(2^i)^m} \quad = \quad \binom{m+k}{m}\frac{2(m+k)r'^m}{\sigma^m}\,(1+O(1/2^m))$$

which is $r'/2$ times cheaper than without hierarchical verification. The search cost becomes now

$$nr\left(\frac{1}{r'}+\binom{m+k}{m}\frac{2(m+k)r'^{m-1}}{\sigma^m}\right)$$

which is minimized for

$$r' \quad = \quad \frac{\sigma}{\left(2\binom{m+k}{m}(m+k)(m-1)\right)^{1/m}}$$

and gives a search time of

$$\frac{nr}{\sigma}\frac{m}{m-1}\left(\binom{m+k}{m}2(m+k)(m-1)\right)^{1/m}$$

An asymptotic simplification (for large m and $\alpha = k/m$ considered constant) of the cost can be obtained using Stirling's approximation to the factorial $m! = (m/e)^m \sqrt{2\pi m}(1 + O(1/m))$:

$$\frac{nr}{\sigma} \frac{(1+\alpha)^{1+\alpha}}{\alpha^\alpha}$$

which monotonically worsens with α, as expected.

This shows that in the best case we may expect a speedup of $O(\sigma)$ by superimposing the subpatterns. The speedup is σ for $k = 0$ and it moves to 1 as α grows. A natural question up to which error level the speedup is larger than 1 (i.e. useful). This is, when it happens that $\sigma\alpha^\alpha > (1+\alpha)^{1+\alpha}$, i.e. $\sigma > (1+\alpha)(1+1/\alpha)^\alpha$. A sufficient condition can be obtained by noticing that $1 \leq (1+1/\alpha)^\alpha \leq e$, and therefore $\alpha < \sigma/e - 1$ suffices. In general it has to hold $\alpha < \sigma/(r'e) - 1$.

For longer patterns all search costs get multiplied by $m \log_2(k)/w$. On the other hand, if the patterns are very short, we may do multipattern search by packing the states of many patterns inside the same computer word, so that we update the states of all the searches in a single operation. The size of the representation of each pattern, however, is nearly $m \log_2(k)$, which makes the idea impractical except for very short patterns. In the next section we present a filter that needs much less information per pattern and therefore is suitable for this approach.

6 A Counting Filter

A different approach to filter the search for multiple patterns is to use a "counting" filter. The filter is based on the notion that if a pattern is found at text position j, then all its characters must appear in the text window $T_{j-m-k+1..j}$. The idea is to keep count at any text position j of how many pattern characters are present in the text window, updating this information in $O(1)$ operations per text character. Note that we cannot ensure that the pattern characters appear in the correct order, so we filter with a necessary condition which is not sufficient to guarantee a match. Moreover, we show that for a multipattern search many counters (one per pattern) can be stored in a single computer word and all can be updated in $O(1)$ operations per text character. Each time a counter reaches the critical value m, it means that all its characters are in the text window and therefore the window is checked using the algorithm of Section 4. A similar idea has been proposed in [10, 15, 16] for the k-differences problem. We now describe the algorithm and later show how to adapt it for multiple patterns (by combining it with bit-parallelism).

6.1 One Pattern

The filter passes over the text examining an $(m+k)$-letters long window. It keeps track of how many characters of P are present in the current text window (accounting for multiplicities too). If, at a given text position j, the m characters of P are in the window $T_{j-m-k+1..j}$, the window area is verified with a classical algorithm (in this paper, with the bit-parallel algorithm of Section 4).

We implement the filtering algorithm as follows: we build a table $A[\,]$ where, for each character $c \in \Sigma$, the number of times that c appears in P is initially stored. Throughout the algorithm, each entry of $A[\,]$ indicates how many occurrences of that character can (still) be taken as belonging to P. We also keep a counter *count* of matching characters. To advance the window, we must include the new character T_{j+1} and exclude the last character, $T_{j-m-k+1}$. To include the new character, we subtract one at the proper entry of $A[\,]$. If the entry was greater than zero *before* the operation, it is because the character is in P, so we increment the counter *count*. To exclude the old character, we

add one at the proper entry of $A[\,]$. If the entry is greater than zero *after* the operation, it is because the character was in P, so we decrement *count*. When the counter *count* reaches m we verify the preceding area.

When $A[c]$ is negative, it means that the character c must leave the window $-A[c]$ times before we accept it again as belonging to the pattern. For example, if we run the pattern `"abca"` over the text `"aaaaaaaa"`, with $k = 1$ it will hold $A['\mathtt{a}'] = -3$, and the value of *count* will be 2. Figure 4 shows another example.

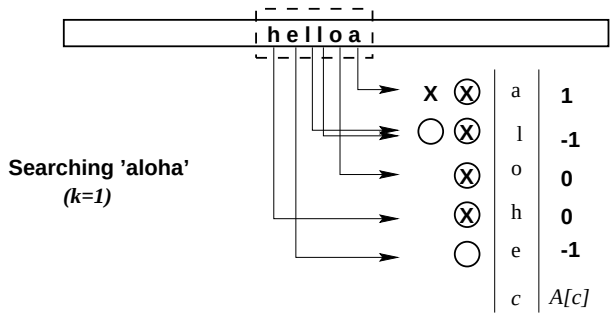

Figure 4: An example of the counting filter. The crosses represent elements which $A[\,]$ accepts, and the circles are the elements that appeared in the window. $A[c]$ stores crosses minus circles, and *count* counts circled crosses.

Figure 5 shows the pseudocode of the algorithm. As it can be seen, the algorithm is not only linear time (excluding verifications), but the number of operations per character is very small.

```
CountFilter (T,n,P,m,k)

        /* Preprocessing */
  for c ∈ Σ do A[c] ← 0
  for i ∈ 1..m do A[P_i] ← A[P_i] + 1
  count ← 0
        /* Searching */
  for j ∈ 1..m + k do    /* fill the initial window */
     if A[T_j] > 0 then count ← count + 1
     A[T_j] ← A[T_j] − 1
  for j ∈ m + k + 1..n do    /* move the window */
     if count = m then verify T_{j−m−k..j−1}
     if A[T_j] > 0 then count ← count + 1
     A[T_j] ← A[T_j] − 1
     A[T_{j−m−k}] ← A[T_{j−m−k}] + 1
     if A[T_{j−m−k}] > 0 then count ← count − 1
```

Figure 5: The filtering algorithm for one pattern.

6.2 Multiple Patterns

The previous algorithm can search for one pattern only. However, we can extend it to handle multiple patterns. To search r patterns in the same text, we use bit-parallelism to keep all the counters in a single machine word. We must do that for the $A[]$ table and for *count*.

The values of the entries of $A[]$ lie in the range $[-m-k..m]$, so we need exactly $1+\ell$ bits to store them, where $\ell = \lceil \log_2(m+k+1) \rceil$. This is also enough for *count*, since it is in the range $[0..m]$. Hence, we can pack

$$\left\lfloor \frac{w}{1+\lceil \log_2(m+k) \rceil} \right\rfloor$$

patterns in a single search (recall that w is the number of bits in the computer word). If we have more patterns, we must divide the set in subsets of at most this size and search each subset separately. We focus our attention on a single subset now.

The algorithm simulates the simple one as follows. We have a table $MA[]$ that packs all the $A[]$ tables. Each entry of $MA[]$ is divided in bit areas of length $1+\ell$. In the area of the machine word corresponding to each pattern, we store $2^\ell + A[] - 1$. When, in the algorithm, we have to add or subtract 1, we can easily do it in parallel without causing overflow from an area to the next. Moreover, the corresponding $A[]$ value is not positive if and only if the most significant bit of the area is zero. Figure 6 illustrates.

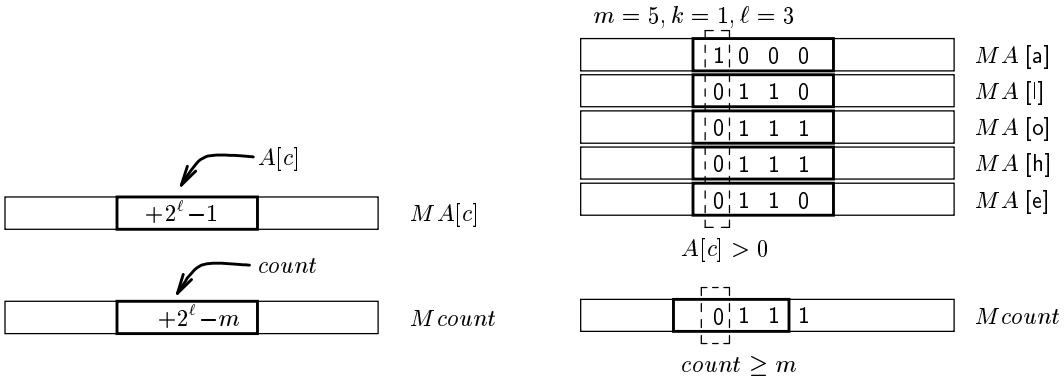

Figure 6: Scheme (left) and an example (right) of the bit-parallel counters. The example follows that of Figure 4.

We have a parallel counter $Mcount$, where the areas are aligned with $MA[]$. It is initialized with $2^\ell - m$ in each area. Later, we can add or subtract 1 in parallel without causing overflow. Moreover, the window must be verified for a pattern whenever the most significant bit of its area reaches 1. The condition can be checked in parallel, although if some counter reaches zero we sequentially verify which one did it.

Observe that the counters that we want to selectively increment or decrement correspond exactly to the $MA[]$ areas that have a 1 in their most significant bit (i.e. those whose $A[]$ value is positive). This yields a bit mask-shift-add mechanism to perform this operation in parallel on all the counters.

Figure 7 shows the pseudocode of the parallel algorithm. As it can be seen, the algorithm is more complex than the simple version but the number of operations per character is still very low.

```
CountFilter (T, n, P^{1..r}, m, k)

        /* Preprocessing */
    ℓ = ⌈log_2(m + k)⌉;
    for c ∈ Σ do MA[c] ← (01^ℓ)^r
    for s ∈ 1..r do
        for i ∈ 1..m do MA[P_i^s] ← MA[P_i^s] + 10^{(s−1)(ℓ+1)}
    Mcount ← (10^ℓ − m)) × (0^ℓ1)^r
        /* Searching */
    for j ∈ 1..m + k do     /* fill the initial window */
        Mcount ← Mcount + ((MA[T_j] >> ℓ) & (0^ℓ1)^r)
        MA[T_j] ← MA[T_j] − (0^ℓ1)^r
    for j ∈ m + k + 1..n do     /* move the window */
        if Mcount & (10^ℓ)^r ≠ 0^{r(ℓ+1)} then
            for s ∈ 1..r do
                if Mcount & 0^{(r−s)(ℓ+1)}10^ℓ0^{(s−1)(ℓ+1)} ≠ 0^{r(ℓ+1)} then
                    verify T_{j−m−k..j−1} for pattern P^s
        Mcount ← Mcount + ((MA[T_j] >> ℓ) & (0^ℓ1)^r)
        MA[T_j] ← MA[T_j] − (0^ℓ1)^r
        MA[T_{j−m−k}] ← MA[T_{j−m−k}] + (0^ℓ1)^r
        Mcount ← Mcount − ((MA[T_{j−m−k}] >> ℓ) & (0^ℓ1)^r)
```

Figure 7: The multiple-pattern algorithm. All the constants are of course precomputed.

6.3 Analysis

We want to determine the probability that the filter triggers a verification for a given pattern. Since the m characters of P can appear at any window position in any order, the probability can be upper bounded by (recall Section 5.2)

$$\binom{m+k}{m}\frac{m!}{\sigma^m} \quad = \quad \frac{(m+k)!}{k!\sigma^m}$$

which, compared to the real matching probability we have been using, has an extra $m!$ factor. Since we pack a pattern in $\lceil \log_2(m+k) \rceil$ bits, the total search cost is

$$nr\left(\frac{\log_2(m+k)}{w} \;+\; \frac{(m+k)!}{k!\sigma^m}\,(m+k)\right)$$

where, unlike the case of superimposed automata, we have to pack the maximum number of patterns together, since the number of verifications triggered does not depend on how the packing is done. We are interested, on the other hand, in the maximum error level α for which this filter is useful.

Applying Stirling's approximation to the matching probability formula we get an asymptotic simplification for large m:

$$\left(\frac{(1+\alpha)^{1+\alpha}m}{e\sigma\alpha^\alpha}\right)^m$$

which is exponentially decreasing with m as long as the base is smaller than 1. When this happens, all the verification costs become negligible. When, on the other hand, the cost is not exponentially decreasing with m, the verifications dominate the search cost and the filter is no longer useful.

So the simplified condition for the filter to be useful is

$$\frac{(1+\alpha)^{1+\alpha}}{\alpha^{\alpha}} < \frac{e\sigma}{m}$$

which worsens as m or α grow. A simplified condition can be obtained by noticing again that $(1+\alpha)^{1+\alpha}/\alpha^{\alpha} = (1+\alpha)(1+1/\alpha)^{\alpha} \leq e(1+\alpha)$, and therefore it suffices that

$$\alpha < \sigma/m - 1$$

to ensure that the filter is useful. Note that the condition is equivalent to $m + k < \sigma$.

7 Experimental Results

In this section we present some experimental results about our algorithms and their analyses.

7.1 Probability of Matching

We test experimentally the probability that a random pattern matches at a random text position. We generated a random text and 100 random patterns for each experimental value shown. Figure 8 (left) shows the number of matches found in a text of 3 Mb for a pattern with $m = 300$, where pattern and text were randomly generated over an alphabet of size $\sigma = 68$. As can be seen, there is a k value from where the matching probability starts to grow abruptly, moving from almost 0 to almost 1 in a short range of values. Despite that this phenomenon is not as abrupt as for the k differences problem [6, 16], it is sharp enough to make this k value the most important parameter governing the behavior of the algorithm. We call k^* this point, and $\alpha^* = k^*/m$ the corresponding error level.

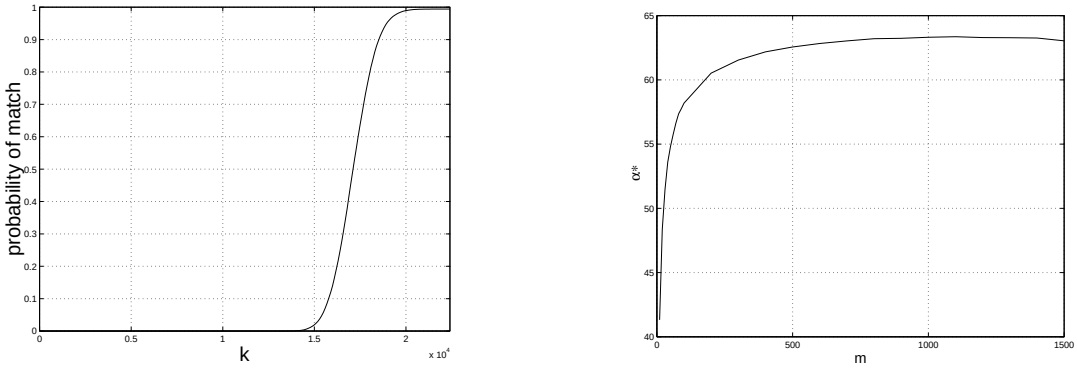

Figure 8: On the left, number of matches found for increasing k values and fixed $m = 300$. On the right, the α^* limit as m grows. **[eje y]**

On the right part of Figure 8 we have shown this limiting α^* value for different pattern lengths, showing that α^* tends to a constant for large m, despite that it is smaller for short patterns.

Finally, we show in Figure 9 how the alphabet size σ affects the α^* value. As can be seen, the curve looks as a straight line, where least squares estimation yields $\alpha^* = \sigma/1.0856 - 0.8878$.

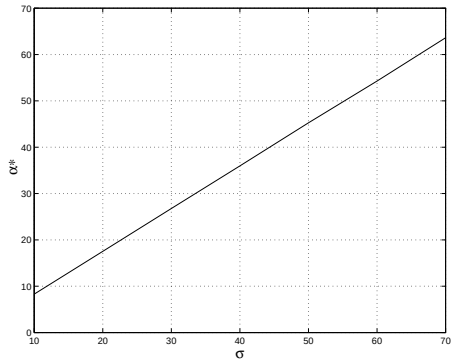

Figure 9: The α^* limit as σ grows.

All this matches our analytical results in the sense that (a) there is a clear error level α^* where the matching probability goes almost from 0 to 1; (b) this point does not depend on m asymptotically; and (c) it depends on σ linearly as predicted by the analysis ($\alpha^* = \sigma/e - 1$) except because the e has been changed to about 1.09. Interestingly, this is similar to the result obtained for the k differences problem in [6, 16] when relating their analytical predictions ($\alpha^* = 1 - e/\sqrt{\sigma}$) with the experiments ($\alpha^* = 1 - 1.09/\sqrt{\sigma}$) and shows a consistent behavior of the pessimistic analytical model used in both cases.

7.2 The Algorithms

We experimentally study our algorithms now. We tested with 35 Mb of random text ($\sigma = 68$) and a set of 100 random patterns of lengths $m \in \{4, 5, 6\}$. This is a typical setup for intrusion detection applications. We use a Sun Enterprise 450 server (4 x UltraSPARC-II 250MHz) running SunOS 5.6 with 512 Mb of RAM and $w = 32$. Each data point was obtained by averaging the Unix's real time over 10 trials.

A first concern is which is the scanning efficiency of the algorithms compared to plain dynamic programming for one pattern, independently of their filtering efficiency to deal with multiple patterns. Figure 10 shows the scanning efficiency of the dynamic programming, the bit-parallel simulation and the conting filter (using the bit-parallel simulation as the verification engine) for single random patterns with $m = 4$. We measure the megabytes per second (Mb/s) processed by the algorithms as k increases. As can be seen, the bit-parallel simulation is 2.5 to 3 times faster than the classical solution even for very large k values. The counting filter is in between.

We compare now the impact of the number of patterns r' in the multipattern filter based on superimposed automata. We take $m = 4$ (i.e., the length of the shortest pattern in the set) and $\sigma = 68$ for our analytical estimation of optimal superimposition, which yields $r'_{k=4} = 8.93$, $r'_{k=6} = 6.41$ and $r'_{k=8} = 4.94$. Figure 11 (left) shows the Mb/s processed when using different values of r' over a set of 100 patterns. As the analysis predicts, there is an optimal amount of superimposition that is reduced as k grows. The analytically estimated optima are below the practical ones, since our analysis uses a pessimistic bound on the matching probability. We use the experimental optima in the tests that follow.

We now show the degree of parallelism achieved by the superimposition and counting filters algorithms, in terms of the ratio between the parallel version and r applications of the corresponding single-pattern algorithm. We search the same set of randomly selected patterns ($m \in \{4, 5, 6\}$) with

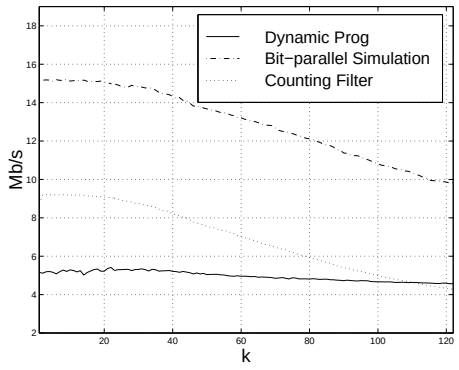

Figure 10: Scanning efficiency of the bit-parallel simulation and the counting filter compared to the classical dynamic programming algorithm.

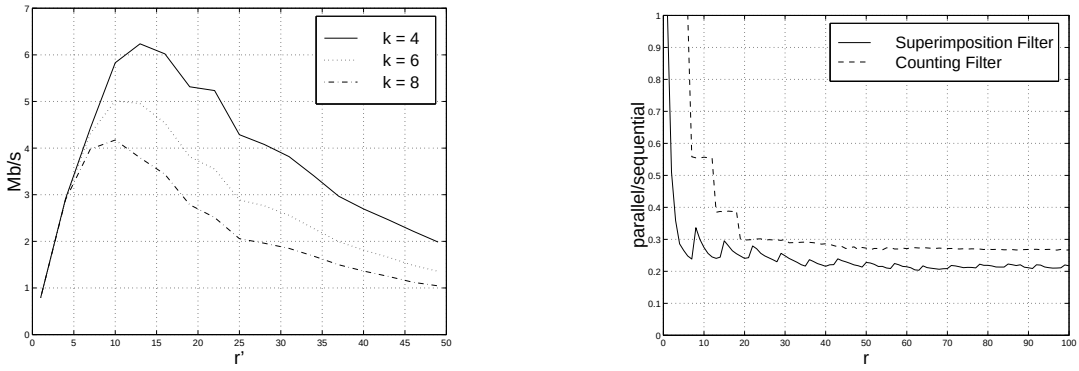

Figure 11: On the left, Mb/s vs partition size for $k = 4$, $k = 6$ and $k = 8$ over a set of 100 patterns with $m \in \{4, 5, 6\}$. On the right, ratio between parallel and sequential versions of the algorithms.

$k = 8$. Figure 11 (right) shows the behavior in terms of r. We observe that the multipattern filter quickly converges to a 5-fold improvement over its sequential version as r increases. The counting filter achieves a lower degree of parallelism, taking 0.27 of its sequential counterpart.

Figure 12 shows the impact of searching allowing different numbers of insertions for both algorithms, for pattern sets of $r = \{1..100\}$. We observe that performance remains stable up to a limit around $r = 25$ with low k. For higher k values, however, performance drops drastically from the beginning. The counting filter resists more this behavior, which shows its higher tolerance to insertions for short patterns. To see this, note that the case $m = 6$, $k = 25$ and $\sigma = 68$ is totally inside the scope of the counting filter according to the analysis, while the superimposition filter can only superimpose 3 patterns under this setup.

8 Conclusions

We have presented a string matching approach to the problem of intrusion detection, which is formalized as the problem of multipattern matching allowing insertions. Besides the classical solution for one pattern adapted from the field of approximate pattern matching, we have presented two new search algorithms which we also extended to handle multiple patterns. Each of the two algorithms

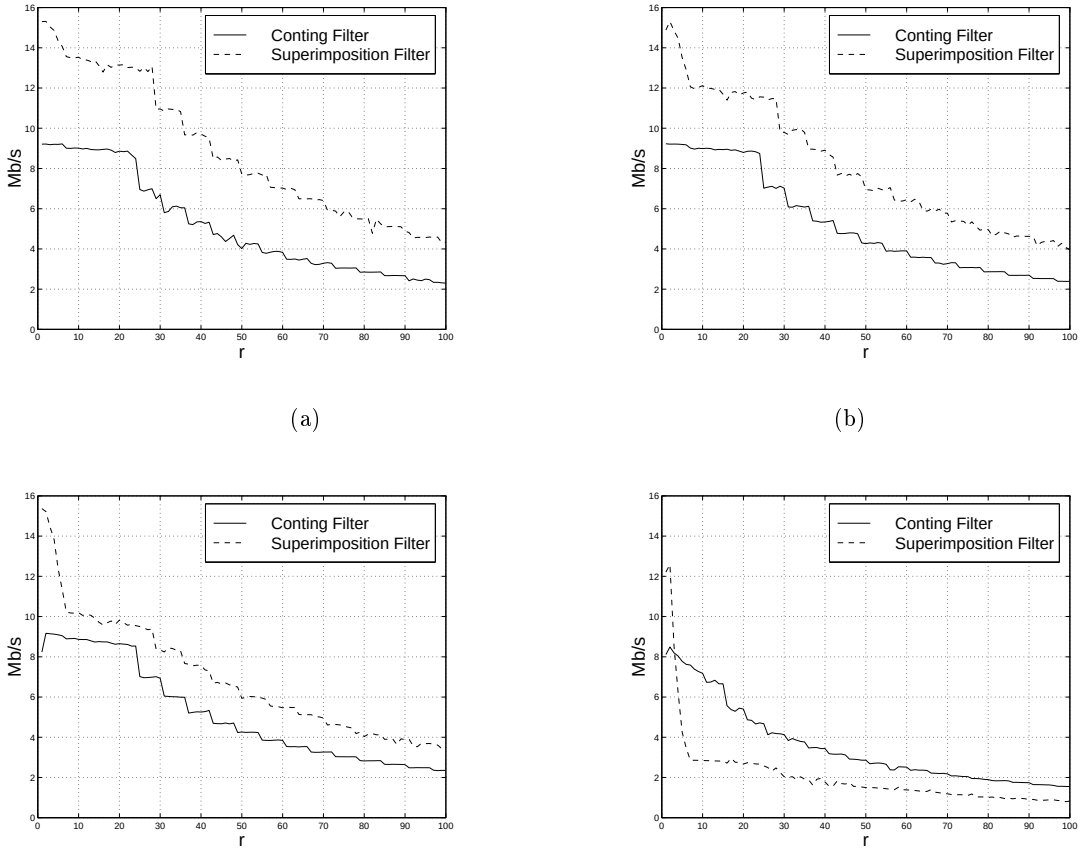

Figure 12: Mb/s processed by both algorithms for a set of patterns with $m \in \{4, 5, 6\}$ with (a) $k = 4$, (b) $k = 6$, (c) $k = 8$ and (d) $k = 25$.

can be better than the other depending on the number of insertions allowed.

We have presented analytical and experimental results concerning the performance of the new algorithms. As an example, we illustrate the case of 4-letters patterns searched allowing 4 insertions, which is a case of interest in intrusion detection applications. The single pattern versions are typically 3 times faster than the classical solution. The multipattern algorithms allow searching 100 patterns at the same cost of 4 single pattern searches (a 25-fold speedup). As a result, our new algorithms allow searching for 100 patterns at a rate of 4 Mb/s in our machine, while the classical algorithm can search for just one single pattern at 5 Mb/s.

In the field of approximate string matching, the fastest algorithms are filters able to discard most of the text by checking a necessary condition. In general, those filters cannot easily be applied here because the error levels typical in intrusion detection applications are too high for the standards of the approximate string matching problem. We have shown, however, that some filtration techniques can be adapted to this problem to obtain a large improvement in the performance of multipattern

searching.

Future work involves finding new algorithms, as well as a detailed study of optimization and extensions on the current ones:

– In the multipattern filter algorithm, if the patterns have different length, we truncate them to the shortest one when superimposing the automata. We can select cleverly the substrings to use, since having the same character at the same position in two patterns improves the filtering mechanism.

– We used simple heuristics to group subpatterns in the superimposed automata. These can be improved to maximize common letters too.

– The multipattern filter is limited to patterns of size $m(\lceil \log_2(k+1) \rceil + 1) \leq w$. Automaton and pattern partition techniques [6] can be incorporated to search longer patterns. Furthermore, the combination of techniques can be considered in order to increase the tolerance to insertions.

Related to this last point about the length of the patterns, we point out that we have concentrated in the parameters typical of intrusion detection, where the patterns are rather short, the error level is quite high, and the number of patterns is large. The new algorithms we have presented are very well suited to this setup, but other variants of the problem could be of interest in other applications and could demand (or permit) different approaches. In particular, more sophisticated models of attacks may yield more cpomplex pattern matching iproblems.

References

[1] R. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Dept. of Computer Science, Univ. of Waterloo, May 1989. Also as Research Report CS-89-17.

[2] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.

[3] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Comm. of the ACM*, 35(10):74–82, October 1992.

[4] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, LNCS 1272, pages 174–184, 1997.

[5] R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. Technical Report TR/DCC-98-10, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/multi.ps.gz`.

[6] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[7] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. CPM'97*, LNCS 1264, pages 12–27, 1997.

[8] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proc. IEEE Symp. on Research in Security and Privacy*, 1994.

[9] K. Ilgun. USTAT: A real-time intrusion detection system for UNIX. Master's thesis, Computer Science Dept., University of California, Santa Barbara, July 1992.

[10] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.

[11] K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, MIT, Dept. of Electrical Engineering and Computer Science, June 1999.

[12] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Dept. of Computer Science, Purdue University, August 1995.

[13] R. Muth and U. Manber. Approximate multiple string search. In *Proc. CPM'96*, LNCS 1075, pages 75–86, 1996.

[14] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS 1448, pages 1–13, 1998.

[15] G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP'97*, pages 125–139. Carleton University Press, 1997.

[16] G. Navarro. *Approximate Text Searching*. PhD thesis, Dept. of Computer Science, Univ. of Chile, December 1998. Technical Report TR/DCC-98-14. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/thesis98.ps.gz`.

[17] G. Navarro. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. Submitted. `ftp://ftp.dcc.uchile.cl/pub/-users/gnavarro/survasm.ps.gz`.

[18] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/dexp.ps.gz`.

[19] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[20] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.