Fundamenta Informaticae XXI (2001) 1001–1027 1001
IOS Press

Fast Multipattern Search Algorithms for Intrusion Detection

Josué Kuri[*]
Dept. of Computer Science and Networks, ENST
46, rue Barrault, 75634 Paris, France
kuri@enst.fr

Gonzalo Navarro[† C]
Dept. of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Ludovic Mé
Supélec. Campus de Rennes, France
Ludovic.Me@supelec.fr

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 for large sets of patterns allowing a wide number of spurious characters, yielding in our machine about a 75-fold improvement over the classical dynamic programming algorithm.

Keywords: Approximate string matching, bit-parallelism, security, audit trails.

[*]Work supported by CONACyT grant # 122688.
[†]Supported in part by Fondecyt grant 1-020831.
[C]Corresponding author

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.

We give an example to illustrate how an intrusion detection problem can be translated into a pattern matching problem. Auditable events in the target system (such as TCP/IP packages in a network or commands typed by users of a multi-user computer) can be seen as characters of an alphabet Σ and the audit trail as a large *string* of characters 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 large 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 characters must be allowed when searching for 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.

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 common property of audit 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. The number of known attacks to system vulnerabilities is large, so a common request for an intrusion detection system to search for attack sets of more than 100 elements. Under the approach of mapping events to characters, the typical alphabet size may vary from 60 to 80, depending on the number of different auditable events in a particular system.

With respect to the typical k values, it is important to avoid false positives (*i.e.*, triggering unnecessary alarms for sequences that do not really represent an attack because k is too large) and to avoid false negatives (*i.e.*, missing true attacks). Empirical values of k are typically between 6 and 10. See [17, 14, 13] for justifications of all these values.

An extended version of this problem (namely searching allowing k differences, or allowing edit distance at most k) has received a lot of attention in the last decades [23], and some of the algorithms can be particularized to solve this problem for one pattern. However, no specially designed solutions exist that take full advantage of the nature of this problem. It would be misleading to think that permitting just insertions makes the problem easier. For example, several important invariants that make life easier hold in the k differences problem and do not hold in the k insertions problem (*e.g.*, that contiguous cells in the matrices used to compute the distances differ at most by one). Moreover, no solutions exist for the multipattern search problem, which is essential in this application.

In this paper we present two different solutions for multipattern searching allowing insertions, which are especially tailored to the setup typical of intrusion detection applications: short patterns, large error levels, large alphabets, large number of patterns. Both solutions are based on bit parallelism, a technique to pack many values in the bits of a single computer word and manage to update all them in parallel. A first one uses bit parallelism to simulate the behavior of a nondeterministic finite automaton that finds all

the occurrences of one pattern allowing k insertions, and searches for many patterns by "superimposing" their automata. A second one is a filter that discards most of the text by counting the number of pattern characters that appear in a window, and searches for many patterns by packing many counters in a single computer word. Both multipattern filters need, in order to work efficiently, that the lengths of the patterns involved are not very different. Although we reuse known techniques, their application to pattern matching allowing insertions is not trivial.

We analyze both algorithms and find the optimal way to set up their parameters, as well as their expected case complexity and the maximum error level k/m up to where they are useful. We also present experimental results that confirm our analysis and measure the practical performance of the algorithms. For typical cases our bit-parallel versions for one pattern outperform the classical dynamic programming algorithm by a factor of 3, while the multipattern filters obtain a 25-fold speedup. The net result is a 75-fold speedup over a classical approach. We include domain specific experiments as well.

This paper is organized as follows. Section 2 puts our work and results in context, giving more details about the complexities we obtain and how they relate to previous work and their applications. Section 3 introduces the concept of "insertion distance" and gives a naive algorithm obtained by adapting the classical solution for the k differences problem. Section 4 presents our first algorithm based on bit-parallel simulation of a nondeterministic automaton, for one pattern. Section 5 builds a filtering algorithm for multipattern matching using that simulation. Section 6 presents the counting filter, for one and for multiple patterns. Section 7 gives all our experimental results validating the analysis and testing the algorithms. In Section 8 we apply our algorithms in a real-life case study. Finally, Section 9 gives our conclusions.

Earlier versions of this paper have appeared in a string matching oriented conference [15] and in an intrusion detection oriented one [16].

2. Our Work in Context

2.1. Pattern Matching

A lot of work has been carried out on an extended version of our problem. This extension is called *search allowing k differences*, where not only insertions, but also deletions and replacements are allowed. In a recent survey [23] 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 3) 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 4), 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 [28, 4], but it is worse than a later development [20] which achieves $O(mn/w)$. Interestingly, this last approach cannot be adapted to our problem[1], but that of [28] 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 [8], where the best one needing space polynomial in m takes $O(mn/\log m)$ time. Finally, an independently developed work obtains also $O(nm\log(k)/w)$ time for the k insertions problem [7], yet it does not generalize to multipattern search, as explained next.

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. Little work has been done on multipattern search for the k differences problem [19, 21, 5, 22]. 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 $\alpha = k/m$ is the error level) 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. When the patterns have different lengths, these results still apply taking m as the minimum pattern length.

2.2. Intrusion Detection

Research in intrusion detection has emerged in recent years as a major subject in the computer security field because of the difficulty of ensuring that information systems are free from security flaws. Computer systems suffer from security vulnerabilities regardless of their purpose, manufacturer or origin. It is both technically hard and economically costly to ensure that systems are not susceptible to attacks. Two approaches have been proposed to address the problem [17, 9, 14].

A first approach, anomaly detection, suggests that user's activity in the system can be characterized so that a profile of "normal utilization" of the system is established and excursions from this profile are tagged as potential intrusions, or attacks in a more general sense. This approach leads to some difficulties: a flow of alarms is generated in the case of a noticeable systems environment modification and a user can slowly change his behavior in order to cheat the system.

We are more interested in misuse detection [10], which assumes that attacks are well-known sequences of actions, called scenarios or attack signatures, and that the activity of the system (in the form of logs, network traffic, etc.) may be audited in order to determine the presence of such scenarios in the system.

Misuse detection becomes an increasingly demanding task in terms of semantics and processing, as more sophisticated attacks are discovered every day [13] (which implies an increasing number of sophisticated scenarios to search for in audit trails). These challenges have lead to a research trend aimed to a simplified representation of the problem in order to improve performance and efficiency of

[1]The reason is that it strongly relies on the fact that consecutive cells in the dynamic programming matrix differ at most by 1, which permits representing a column using $2m$ bits, which is not possible when only insertions are permitted.

detection. In the short term, effective intrusion detection systems will incorporate a number of techniques rather than a "one-strategy-fits-all" approach. The greater the variety of available tools is, the better the intrusion detection system is.

In general terms, the misuse detection problem is to detect the existence of a priori known series of events within the traces of activity of a system to protect. Traces widely differ in their origin, form and content, depending on the type of potential attacks that they attempt to cover. For example, traces in the form of network traffic collected by a firewall or a sniffer may be used to detect well-known attacks to implementations of a TCP/IP protocol stack. Another example are the logs of commands typed by users of a multi-user computer. In both cases, traces may be collected at a single place (*e.g.*, an ethernet segment, a host computer) or at multiple locations simultaneously. We consider the detection of attacks using logs (audit trails) of commands typed by users of a distributed computer system.

A recent approach [18, 16] to the problem of handling a search of increasing complexity and magnitude is to develop systems for fast detection of potential attacks rather that accurate detection of actual attacks. The results of such a detection (*i.e.*, filtered audit trails, in which attacks may be present) would be used in turn as input for a more accurate (and slower) detection algorithm.

Under this approach, the misuse detection problem is modeled as a pattern matching problem in the following way: auditable commands in the system can be seen as characters of an alphabet Σ and the audit trail as a large string of characters 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. Since attackers may introduce spurious commands among those that represent an actual attack in order to disperse their evidence, a limited number of spurious characters must be allowed when searching for the pattern. Since the number of known attacks to system vulnerabilities is large, we are interested in simultaneously searching for a set of patterns. Thus, the misuse 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. Figure 1 illustrates our model to map the misuse detection problem as a multiple approximate pattern matching problem.

3. 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 $id(a, b) = \infty$ if conversion is not feasible. Clearly, $id(a, b) = |b| - |a|$ if a is a subsequence of b, and ∞ otherwise.

A related 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

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

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 a dynamic programming algorithm for k differences [27]. 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

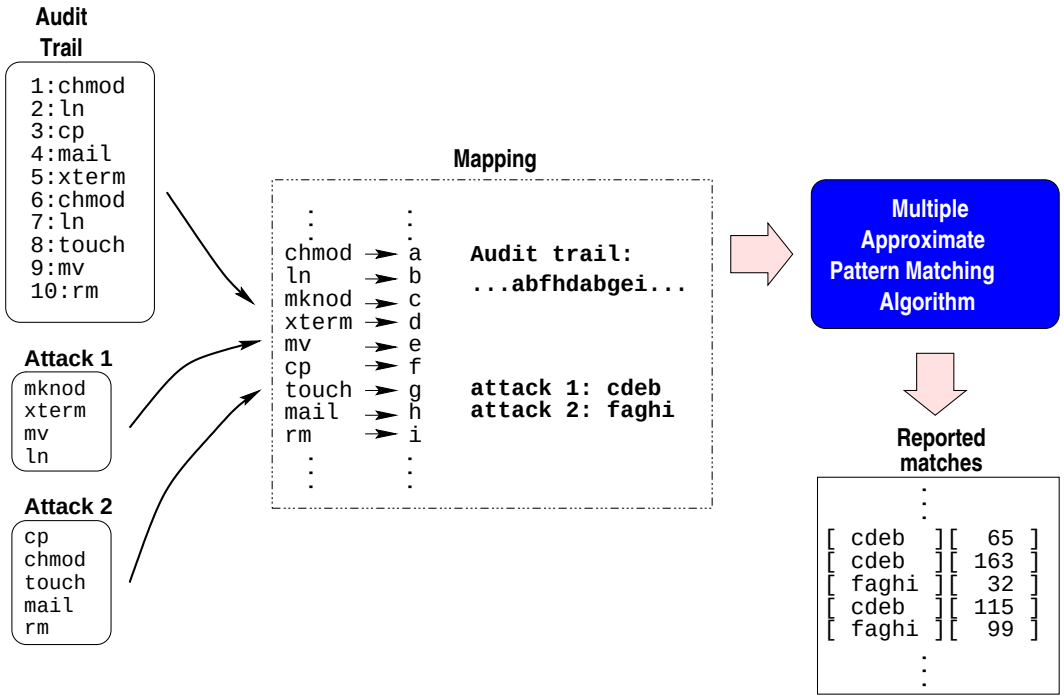

Figure 1. A model for intrusion detection based on pattern matching.

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

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

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.

4. A Bit-parallel Simulation

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).

We introduce now some notation we use for bit-parallel algorithms. We denote as $b_s...b_1$ the bits of a mask of length s. 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,

"$\hat{}$" 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 operation, "$>>$", moves the bits to the right and enters zeros from the left, *i.e.*, $b_s b_{s-1}...b_2 b_1 >> r = 0^r b_s b_{s-1}...b_{s-r+1}$. 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 become 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 computer registers), 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.

We show now how we can pack the C_i values of Section 3 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

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

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

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

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.*,

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

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

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

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 complete algorithm is shown in Figure 2.

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

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

Figure 2. 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

We show now how to search for several patterns simultaneously. We will assume that all them have the same length m. If this is not the case, a solution is to truncate all to the shortest length, and if a truncated pattern is found we must verify for its full occurrence. This solution works well as long as the differences in length are not too large.

As already noted in [5, 22], 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 characters but a sequence of *classes* of characters. A character 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

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

which is equivalent to changing line 4 in Figure 2 to

$$\texttt{4. for } c \in P_i \texttt{ do } B[c] \leftarrow B[c] \mid 0^{(m-i)(\ell+1)}01^\ell 0^{(i-1)(\ell+1)}$$

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. 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

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

which necessarily matches when one of the P^j matches, although the converse is not true. For instance, if we search for `"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.

Therefore, the technique consists in superimposing the search patterns, search for the superimposition with the same algorithm of Section 4 (as extended in Section 5 to handle classes of characters), 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 algorithm of Section 4) in the text area $T_{j-m-k+1..j}$. A similar idea was proposed in [5, 22] 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.

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 [25, 22]. 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.

We search for $P^{1..8}$ 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 appear (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

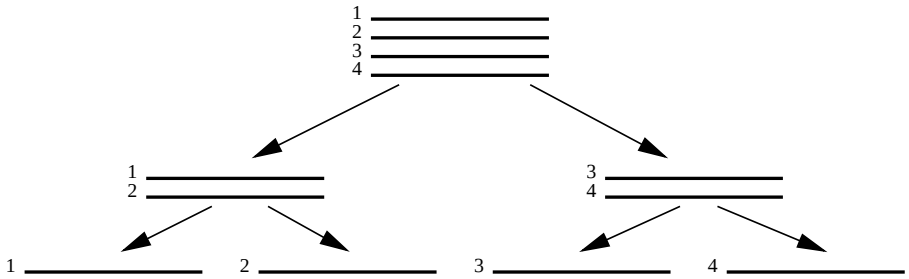

Figure 3. Hierarchical verification for 4 superimposed patterns.

are actually reported). In particular, if a leaf appears it will require all the path of verifications instead of a single verification for the root node. However, as we show next, hierarchical verification pays off because we rarely reach the root node.

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 given text position, with up to k insertions. Consider a 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 characters in order. The window positions that match the pattern characters can be chosen in $\binom{m+k}{m}$ ways. Those characters 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

$$\binom{m+k}{m}\frac{\sigma^k}{\sigma^{m+k}} \quad = \quad \binom{m+k}{m}\frac{1}{\sigma^m} \tag{2}$$

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 random patterns and superimpose them in groups of r', there are at most r' out of σ alphabet characters 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 at most (assuming that each pattern fits 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

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

(assuming $m \geq 2$), which is $r'/2$ times cheaper than without hierarchical verification. The search cost becomes now bounded by

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

which is minimized for

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

and gives a search time bound 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. This means that the amount of grouping permitted depends only on the alphabet size and the error level $\alpha = k/m$. The larger the alphabet or the lower the error level, the more grouping is possible. The speedup is σ for $k = 0$ and it moves to 1 as α grows.

A natural question is: 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 $\alpha < \sigma/(r'e) - 1$. That means that no multipattern search is effective under this method for sufficiently high error levels.

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 [12, 21, 22] for the k-differences problem and earlier [11] for the k-mismatches problem. We now describe the algorithm and later show how to adapt it for multiple patterns (by combining it with bit-parallelism).

Again we will assume that the patterns have the same length, with the possibility of truncating to the shortest pattern if this is not the case. Once more, this solution is effective only if the pattern lengths are not that different.

6.1. One Pattern

The filter passes over the text examining an $(m+k)$-characters 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, $A[c]$ indicates the difference between the number of times c appears in P and the number of times it has appeared in the current window. Only when $A[c]$ is positive we count a character c that enters the window. 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 decrement $A[T_{j+1}]$. 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 increment $A[T_{j-m-k+1}]$. 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 $A['\mathtt{a}'] = -3$, $A['\mathtt{b}'] = 1$, $A['\mathtt{c}'] = 1$, and the value of $count$ will be 2. Figure 4 shows another example.

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.

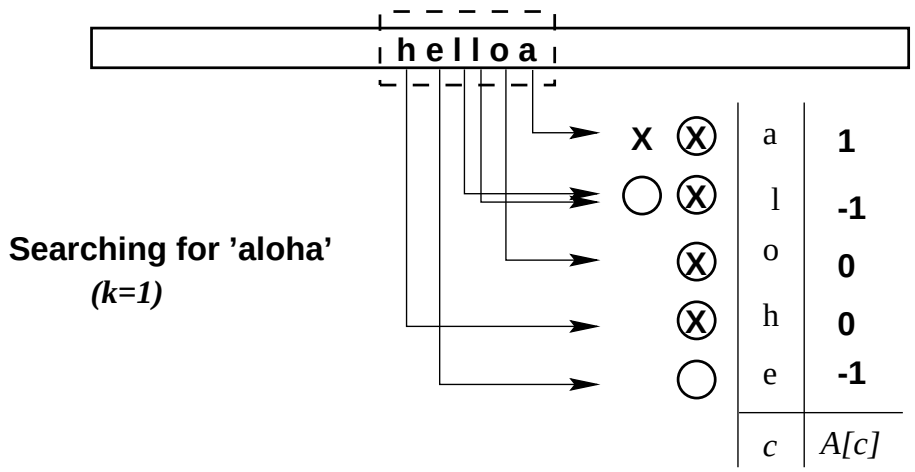

Figure 4. An example of the counting filter. The crosses represent elements which $A[\,]$ accepts, and the circles are the elements that appeared in the window. $A[c]$ stores the number of crosses minus circles, and $count$ counts circled crosses.

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

        /* Preprocessing */
 1.   for c ∈ Σ do A[c] ← 0
 2.   for i ∈ 1..m do A[P_i] ← A[P_i] + 1
 3.   count ← 0
        /* Searching */
 4.   for j ∈ 1..m+k do /* fill initial window */
 5.     if A[T_j] > 0 then count ← count + 1
 6.     A[T_j] ← A[T_j] − 1
 7.   for j ∈ m+k+1..n do /* move window */
 8.     if count = m then verify T_{j−m−k..j−1}
 9.     if A[T_j] > 0 then count ← count + 1
10.     A[T_j] ← A[T_j] − 1
11.     A[T_{j−m−k}] ← A[T_{j−m−k}] + 1
12.     if A[T_{j−m−k}] > 0 then count ← count − 1
```

Figure 5. The counting 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 for r patterns in the same text, we maintain one $A[\,]$ table and $count$ value for each pattern. We use bit-parallelism to keep all these elements in a single machine word, both for $A[\,]$ 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 $\ell = \lceil \log_2(m+k) \rceil$. This is also enough for $count$, since it is in the range $[0..m]$. Hence, we can pack $\lfloor w/(1+\lceil \log_2(m+k) \rceil) \rfloor$ 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 for 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.

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 ones did it. Figure 6 illustrates.

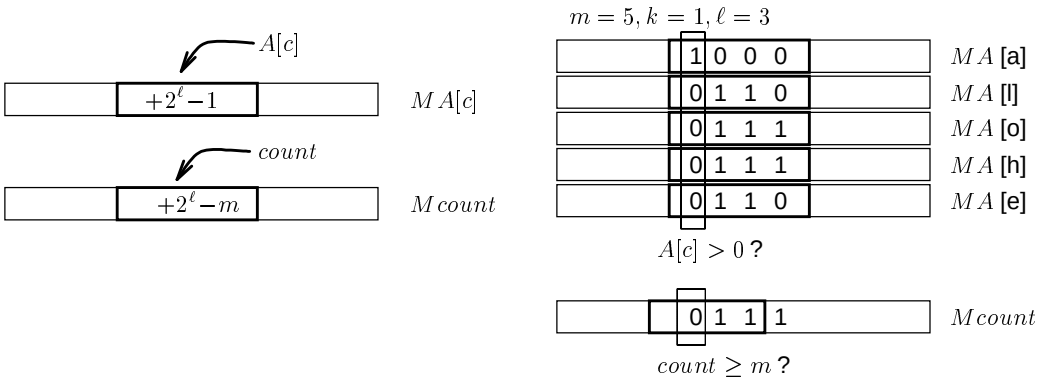

Figure 6. Scheme and an example of the bit-parallel counters. The example follows that of Figure 4.

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.

6.3. Analysis

We want to determine the probability that the filter triggers a verification for a random pattern. Since the m characters of P can appear at any window position in any order, the probability can be bounded from

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

            /* Preprocessing */
1.    ℓ = ⌈log_2(m + k)⌉
2.    for c ∈ Σ do MA[c] ← (01^ℓ)^r
3.    for s ∈ 1..r do
4.       for i ∈ 1..m do
5.          MA[P_i^s] ← MA[P_i^s] + 10^{(s−1)(ℓ+1)}
6.    Mcount ← (10^ℓ − m) × (0^ℓ 1)^r
            /* Searching */
7.    for j ∈ 1..m + k do /* fill initial window */
8.       Mcount ← Mcount + ((MA[T_j] >> ℓ) & (0^ℓ 1)^r)
9.       MA[T_j] ← MA[T_j] − (0^ℓ 1)^r
10.   for j ∈ m + k + 1..n do /* move window */
11.      if Mcount & (10^ℓ)^r ≠ 0^{r(ℓ+1)} then
12.         for s ∈ 1..r do
13.            if Mcount & 0^{(r−s)(ℓ+1)} 10^ℓ 0^{(s−1)(ℓ+1)} ≠ 0^{r(ℓ+1)} then
14.               verify T_{j−m−k..j−1} for pattern P^s
15.      Mcount ← Mcount + ((MA[T_j] >> ℓ) & (0^ℓ 1)^r)
16.      MA[T_j] ← MA[T_j] − (0^ℓ 1)^r
17.      MA[T_{j−m−k}] ← MA[T_{j−m−k}] + (0^ℓ 1)^r
18.      Mcount ← Mcount − ((MA[T_{j−m−k}] >> ℓ) & (0^ℓ 1)^r)
```

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

above by (recall Section 5.2)

$$\binom{m+k}{m}\frac{m!}{\sigma^m} = \frac{(m+k)!}{k!\sigma^m} \tag{3}$$

which, compared to the actual matching probability of Eq. (2), 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 of Eq. (3) 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 shows the probability of matching in a text of 3 Mb for a pattern with $m = 300$, where pattern and text were randomly generated over an alphabet of size $\sigma = 68$ (this value was chosen based on the number of different events present in our real audit trails; this is typically between 60 and 90). We chose a large m value because, as we see next, the behavior stabilizes for large m.

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 [4, 22], 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.

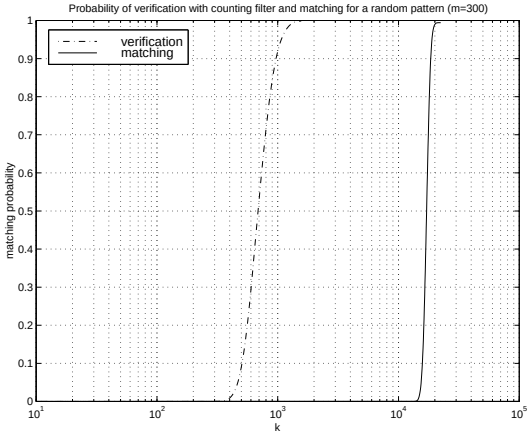

Figure 8. Actual matching probability and probability that the counting filter triggers a verification, for increasing k values and fixed $m = 300$.

Figure 8 also shows the probability that the counting filter triggers a verification. For large m, this probability goes to 1 much before the real matching probability does.

On Figure 9 (left) we have shown this limiting α^* value for increasing pattern lengths, showing that the actual α^* tends to a constant for large m ($\alpha^* = \sigma/e - 1$) in the analysis, despite that it is smaller for short patterns. On the other hand, the maximum error level α^*_{count} up to where the counting filter does not require to verify every position quickly reduces as m grows. The (pessimistic) analysis predicts a limit of the form $\alpha^*_{count} = \sigma/m - 1$. Least squares show an excellent fitting with the curve $\alpha^*_{count} = 13.5182 \times \sigma/m + 1.693$, with a percentual error of 12.32%.

Finally, we show in Figure 9 (right) how the alphabet size σ affects the asymptotic α^* and α^*_{count} values (really for $m = 300$). As can be seen, the curves look as straight lines, where least squares estimation yields $\alpha^* = \sigma/1.0856 - 0.8878$ and $\alpha^*_{count} = 21.5771 \times \sigma/m + 0.6931$ with a percentual error of 1.87%.

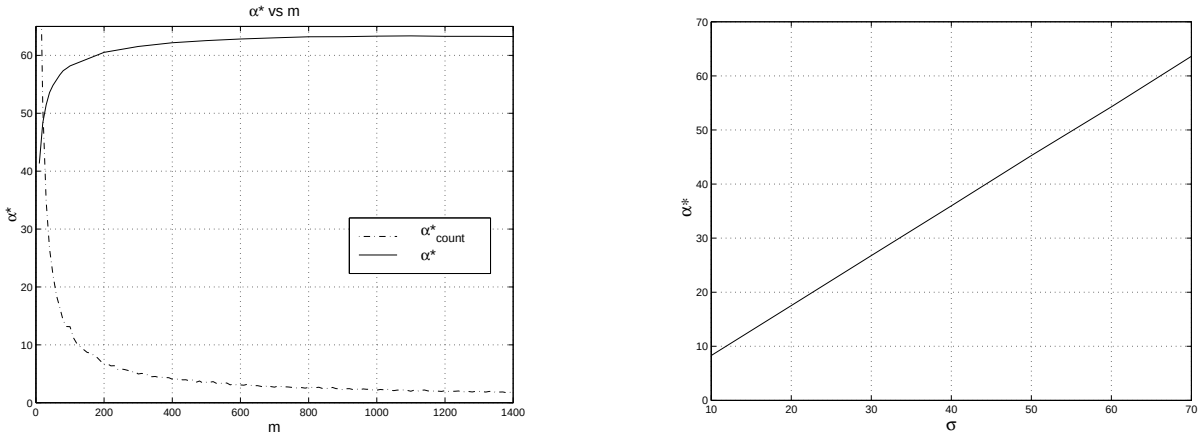

Figure 9. On the left, the α^* limit as m grows for the number of real matches and for the counting filter. On the right, both limits as σ grows, for $m = 300$.

All this matches our pessimistic analytical results. There is a clear error level α^* where the matching probability goes from almost 0 to almost 1, both for the actual matching probability and for the counting filter. This limit depends linearly on the alphabet size σ in both cases. Regarding the dependency with respect to the pattern length m, the real probability tends to a constant while the counting filter decreases with a curve of the form $O(\sigma/m)$.

Since the analysis is pessimistic, the analytical and empirical constants differ. However, there is a remarkable point regarding the real probability of matching. The analysis ($\alpha = \sigma/e - 1$) and the empirical curve ($\alpha = \sigma/1.09 - 0.9$) match provided we use 1.09 instead of e. This means that, even when our analysis is pessimistic, it does predict the real growth rate of the curves up to a constant factor. This is similar to the result obtained for the k differences problem [4, 22] 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. Filtering Efficiency

A second concern is about the ability of our algorithms to filter out text in more typical cases (*i.e.*, short patterns), rather than asymptotically. We are also interested in comparing the plain and hierarchical verification methods.

We have selected three groups of 64 patterns each, of lengths 4, 6 and 8. Each group is searched for with the superimposition method in seven possible ways: one search with all the 64 patterns superimposed, two searches of 32 patterns each, four searches with 16 patterns each, and so on until 64 searches for a single pattern (no superimposition). The same group is also searched with the counting method (the amount of verifications triggered does not depend on the parallelization in this case). The superimposition method is attempted both with the plain and hierarchical verification methods.

Figure 10 shows the number of verifications triggered per pattern and per text character (in the case $r = 1$ we count just the number of matches, so this represents the actual matching probability). This plot is similar to Figure 8 except that we use much smaller m and show also the result of superimposition.

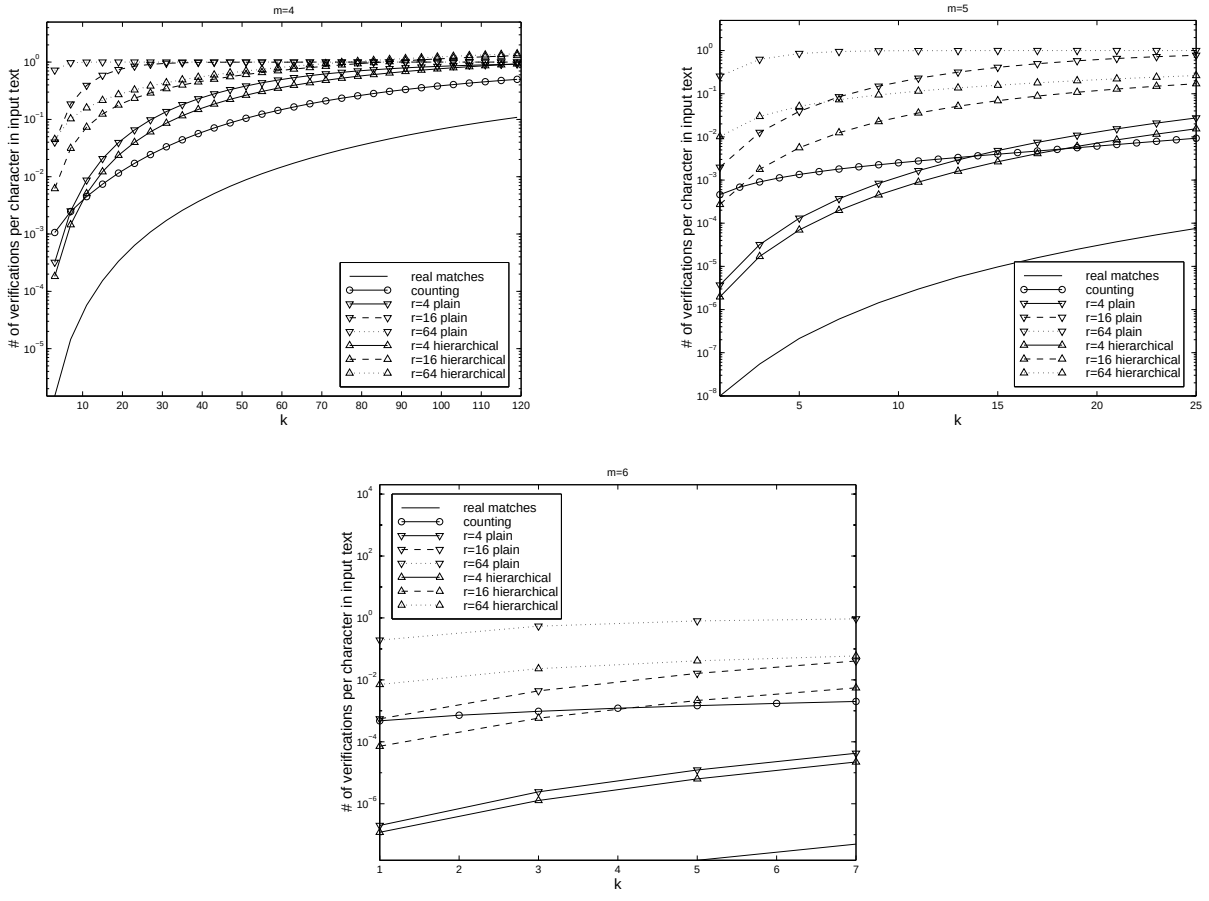

Figure 10. Per character number of real matches, verifications of the counting methods, and verifications for different superimposition schemes. We show the cases $m = 4$, 5 and 6.

As can be seen, hierarchical verification works better than plain verification, despite that in the worst

case it could work up to $\log_2(r)$ more times. The plots also show that (as it should be clear) much more verifications are triggered as r increases. The counting filter shows to be better for larger k values and for smaller m values.

7.3. Scanning Efficiency

We now study the scanning efficiency of our algorithms. 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, as explained in the Introduction. 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.

Our concern now 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 11 shows the scanning efficiency of the dynamic programming, the bit-parallel simulation and the counting 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.

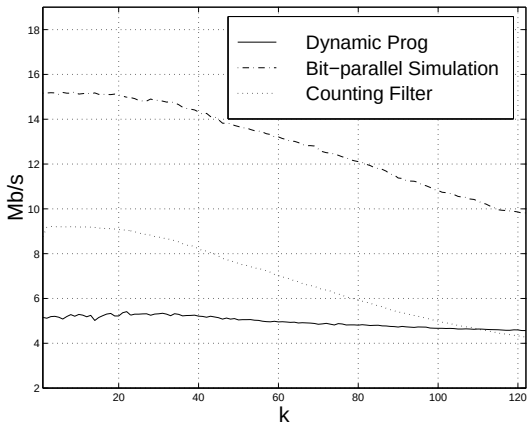

Figure 11. Scanning efficiency of the bit-parallel simulation and the counting filter compared to the classical dynamic programming algorithm.

7.4. Overall Performance

Finally, we consider how the filtering and scanning efficiency combine to form the overall performance.

We compare first 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 12 (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.

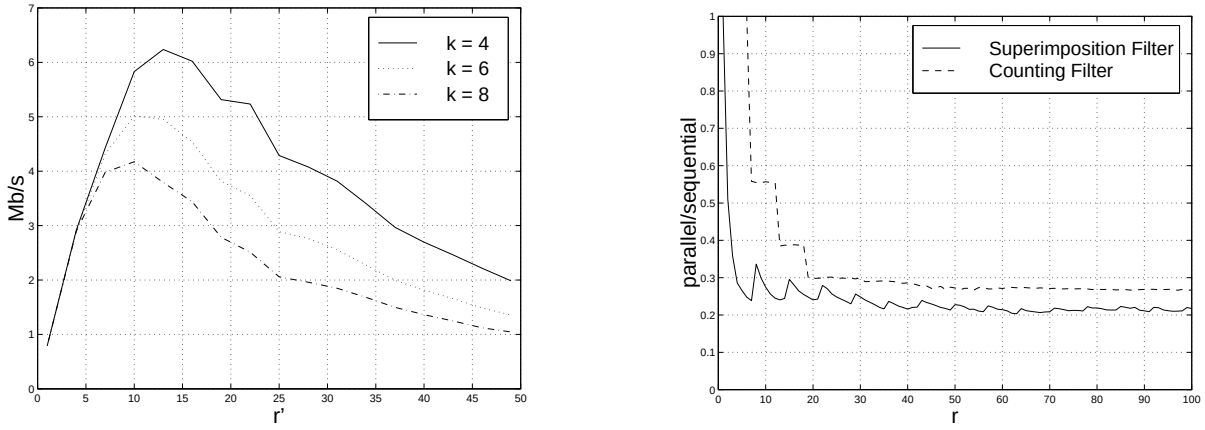

Figure 12. On the left, Mb/s vs partition size for $k = 4$, $k = 6$ and $k = 8$ over a set of 100 patterns with $m \in \{4, 5, 6\}$. On the right, ratio between parallel and sequential versions of the algorithms.

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 for the same set of randomly selected patterns ($m \in \{4, 5, 6\}$) with $k = 8$. Figure 12 (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. The "waves" in the superimposition filter is due to a discretization effect when the patterns are divided into groups.

Figure 13 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. Application to a Real-Life Case

The experiments of the previous section use an idealized model where text and patterns are randomly generated. This is useful to generate massive data and check the experimental performance against the predictions. We complete the above experiments with a real-life case study.

We experimentally study how the probabilistic model of string matching allowing insertions relates to the problem of false negatives and positives. Our goal is to determine how α^* relates to the ratio between false negatives and positives and the total number of reported attacks and, consequently, to the filtering efficiency of the model.

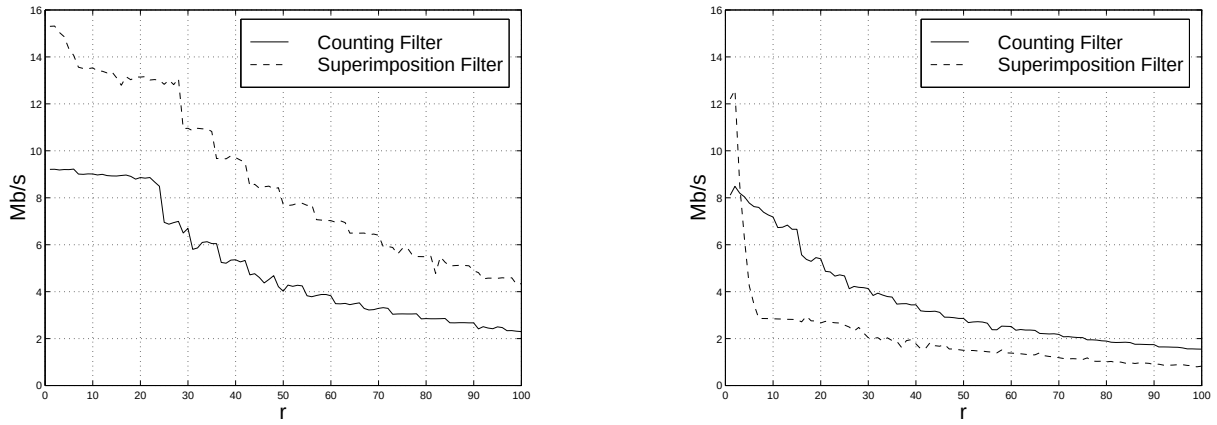

Figure 13. Mb/s processed by both algorithms for a set of patterns with $m \in \{4, 5, 6\}$ with $k = 4$ (left) and $k = 25$ (right).

The experimental input data consists of an audit trail and an attack database. The audit trail was collected using the Gassata intrusion detection system [18] in a real environment. The log format is an extension of the log format proposed in [6]:

```
#S#version=suntrad5.6#system=SOLARIS#deamon=system#ahost=amstel#no=28#
event=AUE_EXECVE#date=2000.3.14@14.29.41#program=/var/audit/ls#
file=/var/audit/ls#euid=root#egid=other#ruid=root#rgid=other#pid=13949#
error=-1#return=KO#E#I#

#S#version=suntrad5.6#system=SOLARIS#deamon=system#ahost=lancelot#no=29#
event=AUE_EXECVE#date=2000.3.14@14.29.41#program=/usr/bin/ls#
file=/usr/bin/ls#arg=ls,-als#euid=root#egid=other#ruid=root#rgid=other#
pid=13949#error=0#return=OK#E#I#
```

The attack database consists of attacks signatures with the following format:

```
>>> Attack_login
rule1
rule1
rule1
>>> Attack_file_creation
rule2
>>> Attack_ps_cmd
rule3
rule7
```

The rules are defined in the following way:

```
rule1 ::= ( (event=AUE_login)||(event=AUE_rlogin) ) && (return=KO) ;
rule2 ::= (event=AUE_CREAT) && ( (file co ls)||(file co cd) ) ;
rule3 ::= (event=AUE_EXECVE) && (program=/usr/bin/ps) ;
```

```
rule4 ::= (event=AUE_EXECVE) && (program co crack) ;
rule5 ::= (event=AUE_su);
```

where the `co` operator stands for "**co**ntains".

The audit trail and attack signatures are translated into a pattern matching representation in three steps. First, a different character is assigned to each rule (*e.g.*, rule1 = `'a'`). Attack signatures are then translated into patterns by mapping their rules to the corresponding characters. Finally, the audit trail is scanned and its events are matched against the rules. Events which match more than one rule are assigned the corresponding characters. Events which do not match a rule are assigned arbitrary characters. The final string is constructed by concatenating the sequence of characters corresponding to matches of rules and the arbitrary characters.

We used an audit file of 24,847 events and studied three different series of actions:

Chained who: represented as a pattern of four events of a `"who"` command. The probability of the corresponding character in our audit file is 0.004382 and there are four real attacks of this kind in the audit file.

Sensitive commands: represented as a pattern of ten events of any command in the set { `"last"`, `"ps"`, `"who"`, `"whois"` }. The probability of the corresponding character in our audit file is 0.007187 and there are two real attacks of this kind in the audit file.

Chained whois: represented as a pattern of four events of a `"whois"` command. The probability of the corresponding character in our audit file is 0.001402 and there is one real attack of this kind in the audit file.

The audit trail and attacks described above may seem not representative enough because of their small size. However, it must be noted that it is extremely difficult to obtain audit trails with traces of attacks for several reasons. First of all, the owners of such trails are reluctant to give their logs away because of confidentiality and security concerns. Secondly, the log generating systems are not perfect in the sense that they do not log all the events that could appear as part of an attack. As a consequence, detection of some attacks is impossible. Though the audit trail and attacks used in this experiment are small, we decided to use them because they are genuine, that is, they correspond to a real case.

8.1. Effectiveness of the Filter and False Negatives

We have searched for the three patterns in our audit file allowing an increasing number of insertions k. Our goal is to determine the effectiveness of the proposed filtering algorithm. That is: how much text is able to filter out in order to retrieve what fraction of the real attacks that occur in the audit file? The text that our filter is not able to discard has to be processed by a more sophisticated algorithm in order to determine the presence of a real attack. As those algorithms are much slower than our pattern matching based approach, the effectiveness of the filter is crucial.

By applying the analytical predictions of Section 5.2 to our real data, we computed the maximum k value for which the matching probability does not reach 1 (recall that the model is pessimistic). To compute that maximum value, we have used the most precise formula (Eq. (2)) for the matching probability. Given that the text is biased we have replaced $1/\sigma^m$ by p^m, where p is the relative frequency of

Attack	m	Occs.	Prob. char	Nec. k	Max. k	Fract. of text
Chained who	4	4	0.004382	225	500	8.21%
Sensitive commands	10	2	0.007187	580	620	14.50%
Chained whois	4	1	0.001402	1425	1570	5.74%

Table 1. Main parameters for the three search patterns.

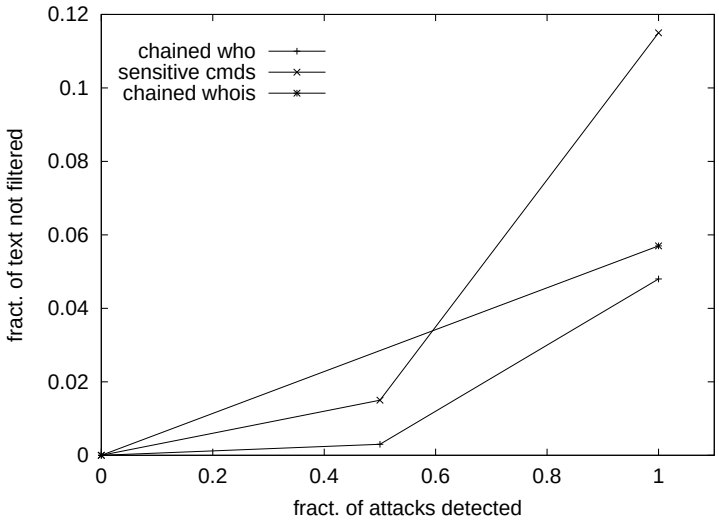

Figure 14. Fraction of attacks detected versus fraction of text left for further processing.

the character that forms the pattern (all the attacks are repetitions of a single character, otherwise we can just multiply the probabilities of the participating characters).

Together with the maximum k recommended by the model we have computed the fraction of the text that the filter selects (for that k) as a candidate for further evaluation. This is simply the $m+k$ characters preceding every match, avoiding to count multiple times the overlapping areas.

Table 1 shows that using the maximum k recommended by the model selects just 6% to 15% of the text to be processed by a more costly algorithm. Moreover, we show in the column of "necessary k" the minimum k value that is necessary to detect all the attacks present in the audit file. This turns out to be below (and generally close to) the maximum k recommended by the model. Therefore, the model can be used to obtain a good estimation of the k value to use in order to detect all real attacks. Of course, it is also possible to use specific knowledge of the application to determine the appropriate k.

8.1.1. False negatives

Another interesting experiment is related to the evolution of the number of attacks detected as a function of the fraction of text not filtered out. That is: what is the percentage of text that we will need to evaluate with a more complex algorithm in order to detect a given fraction of the real attacks?

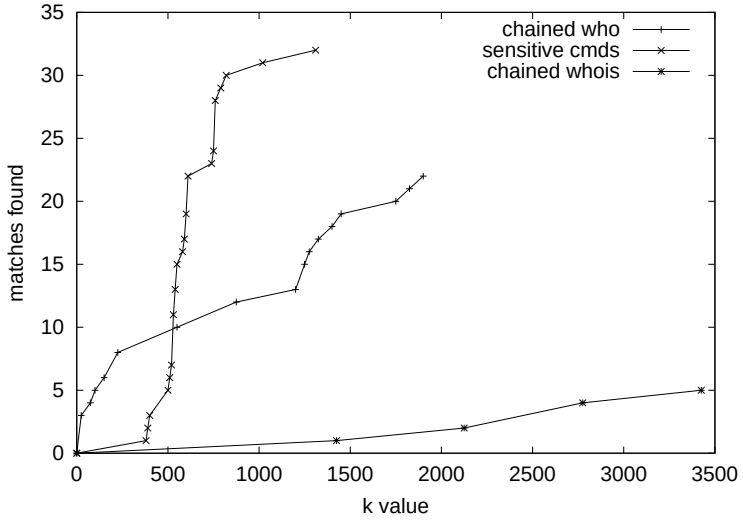

Figure 15. Number of matches found as a function of k.

Regarding the false negatives, we evaluated, for our three particular patterns, the fraction of text filtered as a function of the fraction of attacks detected (see Figure 14). As can be seen, the curve is concave, which suggests that considering a very small fraction of the text permits to detect most of the attacks. For example, with a k value that leaves just 2% of the text for further evaluation we get 50% of the attacks (and thus 50% of false negatives). We have here a way to balance the false negatives rate and the speed of detection. Of course, in many cases, no false negative is required. In that situation, the value of k determined by the model is an upper bound of the value to be used for the corresponding pattern.

8.1.2. False positives

Regarding the false positives, we studied the evolution of the number of matches as a function of k for the three patterns (see Figure 15). Of course, for some patterns, the use of an excessively large k value leads to many false positives. Let us note that these false positives may be discarded by the more accurate detection algorithm which may analyse the output of our pattern matching mechanism (recall that we give a part of the trail containing a *potential* attack). To limit false positives without allowing false negatives, the value of k determined by the model appears as a near-optimum.

It is interesting to observe that some curves of Figure 15 (most notably *sensitive commands*) behave much as the synthetic curve of Figure 8. We believe that the fitting would be better with a larger text, since currently it is not large enough to obtain smooth averages.

9. 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 can be better than the other depending on the number of insertions allowed and the pattern length.

We have presented analytical and experimental results concerning the performance of the new algorithms. As an example, we illustrate the case of 4-characters patterns searched for 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 for 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, a 75-fold improvement.

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.

Concerning a detailed tuning of implementation choices, a number of potential optimizations could bring large improvements in practice. For example, in the multipattern filter algorithm, if the patterns have different length, we just 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. Also, we used simple heuristics to group subpatterns in the superimposed automata. These can be improved to maximize common characters too. Finally, the multipattern filter is limited to patterns of size $m(\lceil \log_2(k+1) \rceil + 1) \leq w$. Automaton and pattern partition techniques [4] can be incorporated to search for 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. A weakness of our methods that could be addressed is the need that the patterns have more or less the same length.

In particular, more sophisticated models of attacks may yield more complex pattern matching problems, involving for example regular expression searching allowing insertions or permitting transpositions (swaps) of events in the pattern, as well as a limited number of missing pattern events. We remark that the counting filter is well suited to deal with transpositions, while fast algorithms for regular expression searching based on bit parallelism [24] could be adapted for these more complex problems. For permitting a given number of deletions as well as insertions, where both operations are counted and thresholded separately, there is a dynamic programming solution in [26] applied to a different problem.

References

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

[2] Baeza-Yates, R.: Text Retrieval: Theory and Practice, *12th IFIP World Computer Congress*, I, Elsevier Science, September 1992.

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

[4] Baeza-Yates, R., Navarro, G.: Faster Approximate String Matching, *Algorithmica*, **23**(2), 1999, 127–158.

[5] Baeza-Yates, R., Navarro, G.: New and Faster Filters for Multiple Approximate String Matching, *Random Structures and Algorithms (RSA)*, **20**, 2002, 23–49.

[6] Bishop, M.: A standard audit log format, *Proc. 19th National Information Systems Security Conference*, 1995.

[7] Boasson, L., Cegielski, P., Guessarian, I., Matiyasevich, Y.: Window Accumulated Subsequence Matching is linear, *Annals of Pure and Applied Logic*, **113**, 2002, 59–80, Previous version in *ACM PODS'99*.

[8] Das, G., Fleischer, R., Gasieniec, L., Gunopulos, D., Kärkkäinen, J.: Episode Matching, *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, LNCS 1264, 1997.

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

[10] Garvey, T., Lunt, T.: Model-based intrusion detection, *Proc. 14th National Computer Security Conference*, October 1991.

[11] Grossi, R., Luccio, F.: Simple and Efficient String Matching with k Mismatches, *Information Processing Letters*, **33**(3), 1989, 113–120.

[12] Jokinen, P., Tarhio, J., Ukkonen, E.: A Comparison of Approximate String Matching Algorithms, *Software Practice and Experience*, **26**(12), 1996, 1439–1458.

[13] Kendall, K.: *A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems*, Master Thesis, MIT, Dept. of Electrical Engineering and Computer Science, June 1999.

[14] Kumar, S.: *Classification and Detection of Computer Intrusions*, Ph.D. Thesis, Dept. of Computer Science, Purdue University, August 1995.

[15] Kuri, J., Navarro, G.: Fast Multipattern Search Algorithms for Intrusion Detection, *Proc. of the 6th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*, IEEE CS Press, 2000.

[16] Kuri, J., Navarro, G., Mé, L., Heye, L.: A Pattern Matching Based Filter for Audit Reduction and Fast Detection of Potential Intrusions, *Proc. of the 3rd International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, LNCS v. 1907, 2000.

[17] Lunt, T.: A survey of intrusion detection techniques, *Computers and Security*, **12**, 1993.

[18] Mé, L.: Gassata, a genetic algorithm as an alternative tool for security audit analysis, *Proc. of the 1st International Workshop on the Recent Advances in Intrusion Detection (RAID'98)*, 1998.

[19] Muth, R., Manber, U.: Approximate Multiple String Search, *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, 1996.

[20] Myers, G.: A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Progamming, *Journal of the ACM*, **46**(3), 1999, 395–415.

[21] Navarro, G.: Multiple Approximate String Matching by Counting, *Proc. 3rd South American Workshop on String Processing (WSP'97)*, Carleton University Press, 1997.

[22] Navarro, G.: *Approximate Text Searching*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Chile, December 1998, Technical Report TR/DCC-98-14.

[23] Navarro, G.: A guided tour to approximate string matching, *ACM Computing Surveys*, **33**(1), 2001, 31–88.

[24] Navarro, G.: NR-grep: a Fast and Flexible Pattern Matching Tool, *Software Practice and Experience (SPE)*, **31**, 2001, 1265–1312.

[25] Navarro, G., Baeza-Yates, R.: Improving an Algorithm for Approximate Pattern Matching, *Algorithmica*, **30**(4), 2001, 473–502.

[26] Navarro, G., Baeza-Yates, R., Arcoverde, J.: Matchsimile: A Flexible Approximate Matching Tool for Searching Proper Names, *Journal of the American Society for Information Science and Technology (JASIST)*, 2003, To appear. Earlier version in *Proc. SBBD 2001*.

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

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