RAIRO-Inf. Theor. Appl. **40** (2006) 303-313
DOI: 10.1051/ita:2006010

# STRING DISTANCES AND INTRUSION DETECTION: BRIDGING THE GAP BETWEEN FORMAL LANGUAGES AND COMPUTER SECURITY

Danilo Bruschi[1] and Giovanni Pighizzini[1]

**Abstract.** In this paper we analyze some intrusion detection strategies proposed in the literature and we show that they represent the various facets of a well known formal languages problem: computing the distance between a string $x$ and a language $L$. In particular, the main differences among the various approaches adopted for building intrusion detection systems can be reduced to the characteristics of the language $L$ and to the notion of distance adopted. As a further contribution we will also show that from the computational point of view all these strategies are equivalent and they are amenable to efficient parallelization.

**Mathematics Subject Classification.** 68M99, 68Q17, 68Q45.

## Introduction

Intrusion detection systems, initially introduced by Anderson [2] and Denning [7], are security devices attempting to identify (in quasi real time) and isolate computer systems intrusions. A very broad classification generally adopted distinguishes between HIDS (Host Intrusion Detection System) and NIDS (Network Intrusion Detection Systems). Host-based IDSs mainly monitor operating system activities on specific hosts, in order to detect intrusion attempts, while network-based IDSs examine network traffic.

Any category of IDS can be further divided into two subcategories on the basis of the mechanisms adopted for detecting malicious activities. More precisely, we distinguish between signature-based IDS (also referred to as misuse detection) and anomaly detection IDS. A misuse detection IDS detects attacks as instances of attack signatures, *i.e.*, sets of rules or filters which characterize a malicious event. Anomaly detection instead focuses on normal system behaviors, rather than attack

[1] Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, via Comelico, 39, 20135 Milano, Italy; {bruschi,pighizzi}@dico.unimi.it

behaviors, *i.e.*, a normal behavior profile is created for any activity performed on the system, which has to be monitored, and any deviation from such a profile is flagged as a potential attack. The misuse detection approach, adopted for almost all the commercial products, enable a very accurate detection of known attacks, but it is ineffective against previously unseen attacks, and it can be bypassed by a slight variation of a known attack. On the other hand, anomaly detection can solve such a problem but it is not still clear how to precisely define and capture the "normal" behavior profile of the activities to be monitored.

In [9, 10] Forrest *et al.* introduced a methodology for building anomaly detection HIDS based on the observation that the behavior of any program can be characterized by the sequence of system calls it makes. The HIDS they proposed works as follows. Let $P$ be the program to be monitored. During a training period, sequences of system calls are collected by executing $P$ many times in a sterile environment. In the meantime characteristic patterns of the collected system calls are placed in a database $D$, which will be subsequently used for intrusions detection. For detecting $P$'s misbehavior the following strategy is followed. Any execution of $P$ in a production environment is monitored and system calls are again collected in a string $x$. At runtime they are compared against the contents of the database $D$. The Hamming distance between $x$ and the database content is computed, when such a value is greater than a given threshold, an alarm is raised. It turns out that in such a context the problem of detecting computer intrusions is equivalent to compute the Hamming distance of regular languages.

Forrest's *et al.* idea has inspired many authors who proposed many variants of the original model, in order to obtain more efficient and more precise (*i.e.*, which recognize broader classes of intrusions) anomaly detection HIDS (see Sect. 3 for further details). In the following we will show that all these approaches are computationally equivalent. More precisely, they can be reduced to the problem of computing distances between strings and languages, using various notions of distance among those provided in literature. As a further contribution we will show that from the computational point of view all the distance notions considered are equivalent and that they can be computed by efficient (in terms of the classical complexity theory) parallel algorithms.

The paper is organized as follows. In Section 1 some preliminary notions about distances between languages and computational models will be recalled. In Section 2 we recall the most prominent results about intrusion detection systems related to our work and we will show the connections between the various notion of intrusion detection system and distances. In Section 3 we will show that the problem of computing all the notions of distances considered can be efficiently parallelized. In Section 4 some final remarks will be provided.

## 1. Preliminaries

Given a finite alphabet $\Sigma$, we denote by $\Sigma^*$ the free monoid of strings over $\Sigma$, and by $\epsilon$ the empty string. Given a string $x \in \Sigma^*$, we denote by $|x|$ its length and by $x_i$ the $i$th symbol of $x$, $i = 1, \dots, |x|$. We recall that a string $z \in \Sigma^*$ is a *subword* or *factor* of $x$ if and only if there exist two strings $u, v \in \Sigma^*$ such that $x = uzv$. If $u = \epsilon$ ($v = \epsilon$, respectively) then $z$ is a *prefix* (*suffix*, resp.) of $x$.

## 1.1. Distances

A distance over $\Sigma^*$ is a function $d : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$ such that, for all $x, y, z \in \Sigma^*$ the following holds

i) $d(x, y) = 0$ if and only if $x = y$;
ii) $d(x, y) = d(y, x)$; and
iii) $d(x, y) \leq d(x, z) + d(z, y)$.

Strings over $\Sigma^*$ can be compared symbol by symbol. In particular, one can define the *Hamming distance* between $x$ and $y$ as the number of positions where the symbol of $x$ is different from the symbol of $y$. More formally:

**Definition 1.1.** Given $x, y \in \Sigma^*$, the *Hamming distance* between $x$ and $y$, denoted by $d_H(x, y)$ is defined as:

$$d_H(x, y) = \begin{cases} \#\{i \mid x_i \neq y_i\} & \text{if } |x| = |y| \\ \infty & \text{otherwise.} \end{cases}$$

The above definition of the Hamming distance is given by considering substitution operations transforming a string into another one. If we consider, besides substitutions, the other two *edit operations* of insertion and deletion, we get another notion of distance, called *edit distance*. More precisely, we can consider the binary relation $\vdash$ over $\Sigma^*$ by setting for all $x, y \in \Sigma^*$, $x \vdash y$ if and only if there exist $u, v \in \Sigma^*$, $a, b \in \Sigma$, with $a \neq b$, such that either condition is satisfied

i) $x = uav$, $y = ubv$ (substitution);
ii) $x = uav$, $y = uv$ (deletion);
iii) $x = uv$, $y = ubv$ (insertion).

Note that $\vdash$ is symmetric. As usual, we denote by $\overset{k}{\vdash}$ the composition of $\vdash$ with itself, $k$ times.

**Definition 1.2.** Given two strings $x, y \in \Sigma^*$, the *edit distance* $d_e(x, y)$ between $x$ and $y$ is the minimum number of edit operations which transform $x$ into $y$, *i.e.*,

$$d_e(x, y) = \min\{k \mid x \overset{k}{\vdash} y\}.$$

Two strings can be also compared by considering their longest common prefix, suffix and subword, respectively. This leads to the following notions:

**Definition 1.3.** Given $x, y \in \Sigma^*$, the *prefix, suffix* and *subword distance* between $x$ and $y$, denoted respectively by $d_p(x,y)$, $d_s(x,y)$ and $d_f(x,y)$, are defined as:

$$\begin{aligned} d_p(x,y) &= |x| + |y| - 2\max\{|z| \mid x, y \in z\Sigma^*\} \\ d_s(x,y) &= |x| + |y| - 2\max\{|z| \mid x, y \in \Sigma^* z\} \\ d_f(x,y) &= |x| + |y| - 2\max\{|z| \mid x, y \in \Sigma^* z \Sigma^*\}. \end{aligned}$$

Each distance over $\Sigma^*$ can be extended to a distance between strings and languages. In particular, the distance between a string $x$ and a language $L$ is defined as the minimum of the distances between $x$ and the strings belonging to $L$. Thus

**Definition 1.4.** Let $\Sigma$ be a finite alphabet, $d : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$ a distance, $x \in \Sigma^*$ a string, and $L \subseteq \Sigma^*$ a language. The *distance* between $x$ and $L$ is defined as:

$$d(x, L) = d(L, x) = \begin{cases} \min\{d(x,y) \mid y \in L\} & \text{if } L \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

We recall that it is also possibile to define distances between languages in a standard way, by resorting to the Hausdorff distance. However, this notion is beyond the scope of this paper. For a discussion and results on this topic we address the interested reader to [4].

Sometimes, the edit distance is defined by considering a weight function associating a cost with each edit operation. Also for other distance notions considered in the paper, it is possible to consider weighted versions. Our results can be extended to such versions, however, for the sake of simplicity, here we consider only non-weighted versions.

### 1.2. Computational models and complexity classes

The model of parallel computation we will consider is represented by uniform families of circuits. More precisely, a *family of circuits* is a set $\{C_n \mid n \in N\}$ where $C_n$ is a circuit for inputs of size $n$. $\{C_n\}$ is logspace uniform if the function $n \to C_n$ is computable on a deterministic Turing machine in logarithmic space. $\mathrm{NC}^k$ ($\mathrm{AC}^k$, $\mathrm{SAC}^k$, respectively) denotes the class of problems solvable by logspace uniform families of bounded (unbounded, semiunbounded[1]) fan-in Boolean circuits of polynomial size and $O(\log^k n)$ depth.

Unbounded fan-in circuits are equivalent to parallel random access machines with both concurrent read and concurrent write (CRCW PRAM). In particular the class $\mathrm{AC}^1$ coincides with the class of functions computed by CRCW PRAM in $O(\log n)$ time, using a polynomial number of processors. For further notions

[1]In a circuit *semiunbounded* the fan-in of the AND gates is bounded, while the fan-in of the OR gates is unbounded.

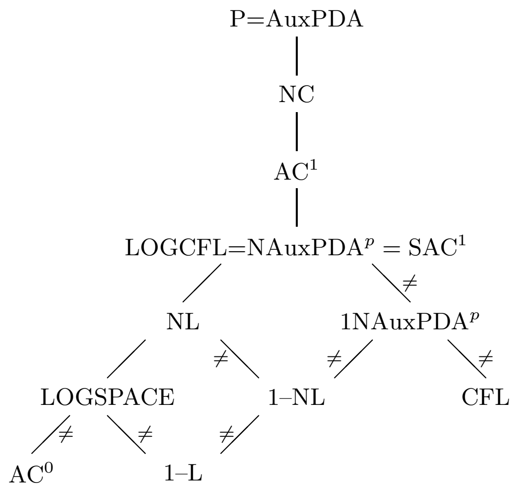

FIGURE 1. Complexity classes.

of parallel computation and arithmetic circuits the reader is referred to Cook [6], Karp and Ramachandran [13].

A *one-way nondeterministic auxiliary pushdown automaton* (1–NAuxPDA) (see Brandenburg [3]) is a nondeterministic Turing machine having a one-way, end-marked, read-only input tape, a pushdown tape, and a two-way, read/write work tape *with a logarithmic space bound.* (For more formal definitions see, *e.g.*, Hopcroft and Ullman [12].)

We recall that without any bound on the running time, two-way nondeterministic and deterministic auxiliary pushdown automata are equivalent, and they characterize the class P (Cook [5]). On the other hand, two-way nondeterministic auxiliary pushdown automata working in polynomial time characterize the class LOGCFL of languages reducible in logarithmic space to context-free languages. As proved in [19], the class LOGCFL coincides with the class of languages recognized by semiunbounded circuits of polynomial size and logarithmic depth, *i.e.*, SAC$^1$ circuits. By 1–NAuxPDA$^p$ we denote the family of language accepted in polynomial time by 1–NAuxPDA. Finally we recall that NL is the class of languages accepted by logspace bounded nondeterministic Turing machines, while 1–NL is the class of languages accepted by logspace bounded one-way nondeterministic Turing machines.

In Figure 1, the relationships among these classes and other complexity classes considered in the literature are summarized.

## 2. Intrusion detection

In this paper we are interested in the host based intrusion detection model initially introduced by Forrest *et al.* As mentioned earlier, such a model, also known as $N$-gram model, is based on the intuition that the "normal" behavior of a program can be characterized by the sequences of system calls it uses during its executions in a sterile environment. The characteristic patterns of such sequences are placed in a database, and they are subsequently used for detecting intrusions in production environments. An essential insight of such an approach was that the database could consist of short substrings, called $N$-grams, that may occur during a program execution. Thus to detect intrusions, sequences of system calls of a given length are collected and compared against the contents of the database, using the Hamming distance. More precisely, given a language $L \subseteq \Sigma^*$ describing the normal behavior of a process $p$ and a string $x$ describing the current behavior of $p$, the HIDS computes $d(L, x)$; if the result is greater than a threshold, experimentally defined, an alarm is raised. In the original model, $L$ is a regular language and the adopted distance function is the Hamming distance.

One of the main drawbacks of the $N$-gram model is its accuracy, *i.e.*, it is not able to recognize particular forms of attacks and, as shown in [11], it is characterized by a relatively high degree of false alarms. Many contributions appeared in the literature suggesting improvements to the $N$-gram model. Here we are particularly interested in [16, 18].

In [16] the behavior of a program is described by a deterministic FSA (Finite State Automaton), which contrarily to $N$-grams is more suited to capture common program structures such as branches, joins and loops, thus contributing to reduce the degree of false alarms. The FSA is built by monitoring the normal program execution at runtime. The states of the automaton are represented by the distinct program counter values at which a system call is made, while transitions are labeled with the syscall name. Such an automaton is then used to detect intrusions in the following way. Anytime the monitored program executes a syscall $s$, a check is performed for verifying if the location from where $s$ was made corresponds to a FSA state and if there exists a transition from the current state labeled with $s$; if not, a transition to a "sink" state is performed. When the number of such transitions, or equivalently the anomaly counter, exceeds a threshold, an alarm is raised[2]. Thus in this case the HIDS operates on a regular language $L$, while the notion of distance adopted for detecting intrusions attempts is the prefix distance.

The model previously described is not able to deal with a specific form of computer attack firstly described in [18] and known as impossible path problem (for further details see also [8]). In order to improve the resistance against this type of attacks Wagner *et al.* introduced in [18], among others, a new HIDS model called *abstract stack model.* Such a model, is still based on the assumption that the

[2]To be more precise the authors of [16] associate different weights with different kinds of anomalies, and use such weights to compute the anomaly counter. It is not difficult to see that this more general case can be handled by adopting a proper FSA.

“normal” behavior of a program can be characterized by the sequence of system calls used, but it derives such sequences *via* static analysis (instead of runtime as the models previously described) of the source code, along with a fixed model of the operating system. A program is modeled as a transition system which can be simulated by a nondeterministic pushdown automata. In the detection phase, intrusions are detected by recognizing strings of system calls which could not have been generated by the underlying transition system. Any time the monitored program makes a system call, one or more steps of the nondeterministic transition system are simulated; if none of them reaches an admissible transition then an alarm is raised. In this case the HIDS operates on a context-free language $L$, while the notion of distance adopted is the prefix distance.

In [14] the $N$-gram model has been extended to work with distributed applications. In such a context it has been argued by the authors that instead of tracing syscalls it was natural to trace calls from client to server, and that variable length $N$-grams were more suitable to capture the anomalous behavior of applications. It turns out that the Hamming distance was no more suitable for comparing process current behavior against normal behaviors, and in such a case the authors proposed to adopt the suffix distance.

So far, the Forrest’s approach has been used for implementing anomaly based HIDS, however it can also implement misuse detection HIDS. In such a case, the language $L$ will contain all the traces of the malicious agents. The string $x$ describing the current behavior of a process $p$ will be compared against strings contained in $L$ in order to verify whether it contains some of them. The detection of intrusions will be performed by computing the subword distance between the regular language $L$ and $x$.

In conclusion, the various approaches adopted for the construction of anomaly based HIDS can be unified by the problem of computing the distance between a string $x \in \Sigma^*$ and a language $L \subseteq \Sigma^*$ for a suitable alphabet $\Sigma$. The differences among the various approaches are determined by the characteristics of $L$ and the notion of distance chosen by the various authors.

In the following, we will provide a computational classification of such a problem and individuate a class of languages for which it can be efficiently (in terms of the classical complexity theory) solved.

## 3. Efficient computation of distances for languages in 1–NAuxPDA$^p$

In [15], it was shown that the edit distance of any language accepted in polynomial time by one-way auxiliary pushdown automata can be efficiently computed. More precisely, it belongs to the parallel complexity class AC$^1$. In [1] a similar result was proved for the *maximal word function* of a such a language. In this section, we are able to show a better result for prefix, suffix, and subword distance. In fact, we show that those distances are in the parallel complexity class SAC$^1$.

Let us start by presenting the following preliminary lemma that states a linear upper bound on the length of the string belonging to a language and with minimal distance with respect to a given string.

**Lemma 3.1.** *Given a nonempty language $L \subseteq \Sigma^*$, there exists a constant $\beta$ such that for any $\sigma \in \{e, p, s, f\}$, for each $x \in \Sigma^*$ and for each $y \in L$ with $d_\sigma(L, x) = d_\sigma(y, x)$, it holds that $d_\sigma(L, x) \leq |x| + \beta$ and $|y| \leq 2|x| + \beta$.*

*Proof.* Let $x_0$ be a string of minimal length belonging to $L$ and $\beta = |x_0|$. It is not difficult to observe that $d_\sigma(L, x) \leq d_\sigma(x_0, x) \leq |x| + \beta$.

Now, observe that $d_\sigma(y, x) \geq |y| - |x|$, for each string $y$. If $d_\sigma(y, x) = d_\sigma(L, x)$, this implies that $|y| \leq 2|x| + \beta$. □

Given a language $L$ and the constant $\beta$ of Lemma 3.1, in order to compute the prefix distance for $L$ we introduce the following relation:

$$\begin{aligned} R_p = \{(i, j, x) \quad | \quad & \exists z, w, v \in \Sigma^* \text{ s.t. } x = zw, zv \in L, |z| = i \text{ and} \\ & 0 \leq |v| = j \leq 2|x| + \beta\}. \end{aligned}$$

We now prove the following result, concerning the set $R_p$:

**Lemma 3.2.** $d_p(L, x) = \min\{|w| + j \mid \exists z \in \Sigma^* \text{ s.t. } x = zw \text{ and } (|z|, j, x) \in R_p\}$.

*Proof.* Let $y \in L$ such that $d_p(L, x) = d_p(y, x)$ with $|y| \leq 2|x| + \beta$. Hence, there exist strings $z, w, v$ such that $x = zw$, $y = zv$ and $d_p(y, x) = |v| + |w| = d_p(L, x)$. It is easy to verify that the triple $(|z|, |v|, x)$ belongs to $R_p$. Hence $d_p(L, x) \geq \min\{|w| + j \mid \exists z \in \Sigma^* \text{ s.t. } x = zw \text{ and } (|z|, j, x) \in R_p\}$.

Conversely, let $(|z|, j, x) \in R_p$, with $x = zw$, for some string $w$, such that $|w| + j$ gives the minimum on the right side of the equality. Then there is a string $v$ of length $j$ such that $zv \in L$. Hence, $d_p(L, x) \leq d_p(zv, x) \leq |w| + |v| = |w| + j$. □

We now consider the following language encoding the relation $R_p$:

$$L_p = \{z\#w\#^j \mid \exists v \in \Sigma^* \text{ s.t. } zv \in L \text{ and } 0 \leq |v| = j \leq 2|zw| + \beta\}.$$

It is immediate to observe that a string $z\#w\#^j$ belongs to the language $L_p$ if and only if the triple $(|z|, j, zw)$ belongs to the relation $R_p$.

Now, we study the relationships between the complexities of languages $L$ and $L_p$. In particular, we prove the following result:

**Theorem 3.3.** *Given $L \subseteq \Sigma^*$, let $L_p$ be the language above defined.*

- $L \in$ 1–NAuxPDA *implies that* $L_p \in$ 1–NAuxPDA.
- $L \in$ 1–NAuxPDA$^p$ *implies that* $L_p \in$ 1–NAuxPDA$^p$.
- $L \in$ 1–NL *implies that* $L_p \in$ 1–NL.

*Proof.* First suppose that $L$ is accepted by the 1–NAuxPDA $M$ We define a 1–NAuxPDA $M_p$ accepting $L_p$ that, on an input string of the form $z\#w\#^j$, works in three phases, as described in the following.

In the first phase, $M_p$ directly simulates a computation of $M$ on the input prefix $z$. During this phase $M_p$ also keeps tracks, using its work tape, of the length of $z$. In the second phase $M_p$ scans the factor $|w|$ to count its length. With this information $M_p$ can compute, using its logarithmic worktape, the number $2|zw|+\beta$. In the third phase, $M_p$ scans the input suffix $\#^j$. In this phase $M_p$ continues the simulation of $M$ of the first phase, by guessing, for each scanned symbol, an input symbol for $M$. If during this phase $M_p$ discovers that $j > 2|zw| + \beta$, then it stops and reject. Otherwise, let $v$ be the string of $j$ symbols guessed by $M_p$ in the third phase. $M_p$ accepts at the end of the computation if and only if the simulated computation of $M$ on input $zv$ is accepting.

It is immediate to verify that if $M$ works in polynomial time then also $M_p$ works in polynomial time. Moreover, if $M$ does not use the pushdown store, namely, it is a one-way machine working in logarithmic space, then $M_p$ is a machine of the same kind.

This completes the proof of the theorem. □

We informally recall that a function $f : \{0,1\}^* \to \{0,1\}^*$ is NC$^1$ *reducible* to a function to $g : \{0,1\}^* \to \{0,1\}^*$ if and only if $f$ can be computed by a family of NC$^1$ circuits, extended with *oracle gates* that compute values of $g$, subject to the restriction that no two oracle gates lie on the same input-output path (for more details see, *e.g.*, [13]). Under a suitable encoding, the notion of NC$^1$-reducibility can be extended in order to consider not only binary inputs and outputs.

**Lemma 3.4.** *The prefix distance for $L$ is* NC$^1$*-reducible to the language $L_p$.*

*Proof.* (outline) Given an input string $x$, the circuit realizing the reduction uses the oracle gate $G_{i,j}$, for $0 \le i \le |x|$ and $0 \le j \le 2|x| + \beta$, in order to check whether or not $(i, j, x) \in R_p$, *i.e.*, $z\#w\#^j \in L_p$, where $zw = x$ and $|z| = i$.

The output of the circuit, *i.e.*, $d_p(L, x)$, is the minimum of all integers $n - i + j$ such that the oracle gate $G_{i,j}$ gives answer "yes".

Because the minimum of a polynomial set of numbers can be computed in AC$^0$ and, hence, in NC$^1$ [17], it is not difficult to verify that this is an NC$^1$-reduction. □

Using Lemma 3.4 we are now able to prove the main result of this paper:

**Theorem 3.5.** *For each language $L \in$ 1–NAuxPDA$^p$, the prefix, suffix and subword distances of $L$ are in* SAC$^1$.

*Proof.* It is immediate to see that SAC$^1$ is closed under NC$^1$-reductions. Hence, if $L \in$ 1–NAuxPDA$^p \subseteq$ SAC$^1$ then by Lemma 3.4 it turns out that its prefix distance is in SAC$^1$.

For the suffix and subword distances the proof can be given using similar steps. In particular, in the case of the suffix distance it is useful to consider the following relation:

$$R_s = \{(i,j,x) \mid \exists z,w,v \in \Sigma^* \text{ s.t. } x = wz, vw \in L, |z| = i, \text{ and } 0 \le |v| = j \le 2|x| + \beta\},$$

while in the case of the subword distance the relation:

$$R_f = \{(i,j',j'',x) \mid \exists w',w'',v',v'',z \in \Sigma^* \text{ s.t. } x = w'zw'', v'zv'' \in L, |z| = i, |v'| = j', |v''| = j'', \text{ and } 0 \le j' + j'' \le 2|x| + \beta\}. \quad \square$$

Using a similar technique, it is easy to prove the following:

**Theorem 3.6.** *For each language $L \in$ 1–NL, the prefix, suffix and subword distances of $L$ are in* NL.

## 4. Conclusions

We have revisited the notion of host intrusion detection system from a formal languages point of view. We have shown that the various models of HIDS based on the approach proposed in [9] represent instances of the problem of computing the distance between a string $x$ and a language $L$, using various notion of distance as well as referring to different type of formal languages. We also showed that such a problem is highly parallelizable. Ongoing research efforts are trying to evaluate the practical impact of such results.

*Acknowledgements.* The authors thank the anonymous referees whose hints contributed a lot to improve the quality of the final version of the manuscript. In particular, a referee suggestion helped us to improve our original result.

## References

[1] E. Allender, D. Bruschi and G. Pighizzini, The complexity of computing maximal word functions. *Comput. Compl.* **3** (1993) 368–391.

[2] J.P. Anderson, *Computer security threat monitoring and surveillance.* Tech. Rep., James P. Anderson Company, Fort Washington (1980).

[3] F. Brandenburg, On one-way auxiliary pushdown automata, in *Proc. 3rd GI Conference. Lect. Notes Comput. Sci.* **48** (1977) 133–144.

[4] C. Choffrut and G. Pighizzini, Distances between languages and reflexivity of relations. *Theoret. Comput. Sci.* **286** (2002) 117–138.

[5] S. Cook, Characterization of pushdown machines in terms of time–bounded computers. *J. ACM* **18** (1971) 4–18.

[6] S. Cook, A taxonomy of problems with fast parallel algorithms. *Inform. Control* **64** (1985) 2–22.

[7] D.E. Denning, An intrusion detection model. *IEEE Trans. Software Engineering* **13** (1987).

[8] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong, Anomaly detection using call stack information, in *Proc. IEEE Symposium on Security and Privacy.* IEEE Press (2003).

[9] S. Forrest, S. Hofmeyr, A. Somayaji and T. Longstaff, A sense of self for Unix processes, in *Proc. IEEE Symposium on Security and Privacy.* IEEE Press (1996).

[10] S. Forrest, S. Hofmeyr, A. Somayaji and T. Longstaff, Intrusion detection using sequences of system calls. *J. Comput. Security* **6** (1998) 151–180.

[11] A.K. Ghosh and A. Schwartzbard, A study in using neural networks for anomaly and misuse detection, in *Proc. USENIX Security Symposium.* USENIX Association (1999).

[12] J. Hopcroft and J. Ullman, *Introduction to automata theory, languages, and computations.* Addison-Wesley, Reading, MA (1979).

[13] R. Karp and V. Ramachandran, A survey of parallel algorithms for shared-memory machines, in *Handbook of Theoretical Computer Science, Vol. A.* North Holland (1990).

[14] C. Marceau, Characterizing the behavior of a program using Multiple length N-grams, in *Proc. New Security Paradimg Workshop.* ACM Press (2000) 101–110.

[15] G. Pighizzini, How Hard is Computing the Edit Distance? *Inform. Comput.* **165** (2001) 1–13.

[16] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni, A fast automaton-based method for detecting anomalous program behaviors, in *Proc. IEEE Symposium on Security and Privacy.* IEEE Press (2001).

[17] Y. Shiloach and U. Vishkin, Finding the maximum, merging and sorting in a parallel computation model. *J. Algorithms* **2** (1981) 88–102.

[18] D. Wagner and D. Dean, Intrusion detection *via* static analisys, in *Proc. IEEE Symposium on Security and Privacy* (2001).

[19] H. Venkateswaran, Properties that characterize LOGCFL. *J. Comput. Syst. Sci.* **43** (1991) 380–404.