PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 49, NO. 1–2, PP. 3–23 (2005)

FILTERING FALSE ALARMS: AN APPROACH BASED ON EPISODE MINING

Ferenc BODON and Zoltán HORNÁK

Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok kőrútja 2, Hungary
e-mails:bodon@cs.bme.hu, hornak@mit.bme.hu

Received: Oct. 24, 2005

Abstract

The security of computer networks is a prime concern today. Various devices and methods have been developed to offer different kinds of protection (firewalls, IDS's, antiviruses, etc.). By centrally storing and processing the signals of these devices, it is possible to detect more cheats and attacks than simply by analysing the logs independently. The most difficult and still unsolved problem in centralized systems is that vast numbers of false alarms. If a harmless pattern, which caused by a safe operation is identified as an alarm, then it is a nuisance and requires human invention to be handled properly.

In this paper we show how we can use data mining to discover the patterns that frequently causes false alarms. Due to the new requirements (events with many attributes, invertible parametric predicates) none of the previously published algorithms can be applied to our problem directly. We present the algorithm ABAMSEP, which discovers frequent alert-ended episodes. We prove that the algorithm is correct in the sense that it finds all episodes that meet the requirements of the specification.

Keywords: data mining, episode mining, computer security, remote supervision system.

1. Introduction

Nowadays it is essential and a basic need to connect the computer network of a company to the World Wide Web. The original purpose of the Internet was to support people in the education with a decentralized network, where the effectiveness (speed, and reliability) was relevant and security was less important. The widespread of WWW opened the gate for new audience and new applications but also posed new demands and stressed that part of the system that received not much attention in the beginning. Unfortunately, additional solutions bring up more problems than those that were taken into consideration in the planning phase. This may be the main reason for being the security of Internet a hot topic in the scientific community.

A wide variety of security devices (virus checkers, firewalls, coding and policy methods, Intrusion Detection Systems, etc.) is available on the market, but each of them just attempts to fill in a security gap. They provide partial solutions and, as they communicate with each other in a limited way, their capabilities are limited. A centralized system that watches every part of the system could collect more data,

could be more efficient in the case of intrusions than a standalone system. We call such system a *Centralized Remote Supervision System*.

In the Remote Supervision System data coming from different security devices placed at different points of the network are collected in the center. It is similar to a traditional security guard who is sitting in front of a monitor wall and can see the monitors of the cameras and the signals of all the protection systems at the same time. Theoretically, by growing the number of the protection systems (and hence the available information) incidents can be handled more effectively. On the other hand we have to face the problem of handling the huge amount of data.

The sensitivity of the typical network monitoring security devices can be set within a wide range. If the parameters are set to high levels, then a device reports an alarm at every event that is a little bit suspicious. In fact, most (but not all!) of the "little bit suspicious" events are harmless, which is the reason for setting the sensitivity parameters to lower values in the practice. In general overloaded system administrators have capacity only to analyse the dangerous and critical events, however for analysing and tracing them back, they need to know the preceding events as well. If the sensitivity is low, real attacks could be ignored, because none of the security devices find them suspicious enough. In the other extreme (high sensitivity) we have to cope with the huge amount of data and the large number of false alarms (an alarm is false if no real attack can be associated with it). Data mining gives us a helping hand in analysing big volumes of data to discover frequent rules of false alarms.

In our approach we collect as much information as we can. By using episode mining algorithms frequent patterns that precede an alarm can be analysed. This makes it possible to automatically discover the reason of frequent false alarms. Our goal is to develop a method that can infer the hidden patterns from the central database. If we can match a known pattern of false alarm to the event sequence preceding an alarm, then we degrade this alarm to false alarm. Of course before accepting a rule for false alarms, the approval of a professional person is needed. This is necessary for adequate human control.

In this paper the architecture of the system and the technical details are not discussed, we are focusing on the mathematical model and the episode mining algorithm. For further details on the overall system the reader should consult [10].

2. Related Work

We shortly review the known episode mining algorithms. The first published algorithm that could cope with large datasets of event sequences was APRIORIALL [2]. It introduced the notion of frequent sequential pattern as a generalization of frequent itemsets known from the association rule mining field. Episodes were defined as sequences of itemsets, and the algorithm found those episodes that were contained in many (more than a given support threshold) sequences. The algorithm GSP [15] and SPADE [16] solve the same problem much faster (in addition, they can handle time constraints).

Another algorithm that finds frequently occurring serial and parallel episodes in one given sequence was presented in [13]. Similarly to our approach, it uses fixed size windows to define the containment relation. In its model the events are atomic, hence its method is not adaptable to the context, where events are determined by parameters.

From our point of view the most promising episode mining algorithm that can handle events with attributes was presented in [12, 8]. We mainly adapted the approach of [12] in our mathematical model. However, we are not looking for episodes and their minimal occurrences, but rather for episodes, which occurred in windows that often ended by alarm. Furthermore, we allow the building blocks of episodes (i.e. the predicates) to be more general by letting them parametric.

The purpose of the Remote Supervision System project was to study the adaptability of data mining techniques to filter false alarms coming from different security devices. Our final goal was to implement a prototype system that proves our hypothesis that data mining is a powerful tool in this field. Our second aim was to construct an efficient and scalable algorithm. It is needless to say, that the system ready for public use has to be fast. However, in our first approach, we avoided the use of sophisticated data structures and other techniques to speed up the programme. We merely wanted to show, that the approach is working and "tuning" of the prototype was left as a work for the future.

3. False Alarms

One cannot give an overall description for the reason of false alarms. Warning messages usually reflect suspicious situations that might be results of attacks or attempted attacks. But if they are not the consequences of such malicious actions, the cause can be almost anything: a misspelled password, a wrongly executed command, configuration problems of network settings, incompatibility of products, software bugs or even a rarely used – otherwise normal – feature of a programme.

A false alarm may be generated for example if someone has a bad IP address configuration on his PC. It will produce several warnings from simple "no network connection" to "possible intruder: alien computer in the system". If the source of this problem is traced, then there is no reason to send repeatedly warning messages that reflect the same problem. We expect the data mining approach to provide rules on events and/or on their attributes, which describe the reason for such frequent unwanted alarms.

In several cases the reasons for unwanted alarms are consequences of the behaviour of software or network elements that cannot be modified. For example if a software component regularly wants to connect to its service portal, looking for updates and the company policy prohibits this activity, then there will be a large number of warnings about someone trying to break the regulation. In many cases there is no option to turn this feature off, the only way to filter out this false alarm is based on an appropriate rule. The goal is to create rules that filter out only those

warnings that are caused by that specific software component. We definitely do not want to give a chance for an attacker to abuse this rule and hide his activity behind a similar alarm.

The task of the data mining algorithm is hard because it should be "open" to discover new and weird causes of alarms, i.e. it has to consider every possibly important attributes of events. On the other hand, the attributes that differentiate false alarms from true positives are the most important. Unfortunately this latter requirement cannot be handled by data mining techniques, since in general we have enough number of samples only for false alarms and not for actual attacks.

4. A Formal Definition of Episode Mining

Among the various data mining approaches the episode mining framework seems to be most suitable for our purposes. Episodes are searched in a sequence of events that are determined by their attributes. Let $R = \{A_1, \ldots, A_n\}$ be a set of attributes, where the domain of A_i is D_i. We denote the set $D_1 \times D_2 \times \ldots \times D_n \times \mathbb{R}$ by $\mathcal{E}$.

Definition 4 1. *An event over attribute set R is an element of $\mathcal{E}$ and we denote it by an $n+1$ tuple $e = (a_1, \ldots, a_n, t)$, where $a_i \in D_i$ and t is a real number, which we call the time of the event.*

In the rest of the paper the time of event e is referred as $e.T$ and the attribute $A \in R$ of e as $e.A$. Some examples for attributes used in the common security message format are: `Type`, `Analyser.Process.Name`, `Create.Time`, `Targe.Node.Address`, `Target.Service.Port`, `Source.User.User-ID`. To handle the very different messages of various security devices we defined a common, XML based file format (called SMEF, Security Message Exchange Format) and converted all incoming messages to this form.

The *alarm function* $W : \mathcal{E} \to \mathbb{N}$ plays an important role in our model. If $W(e) = 0$, then e is said to be a *normal* event, otherwise it is an event that generates *alarm* of type $W(e)$.

An *event sequence* is sequence of events over R, where events are ordered by time. We denote the event sequence of length l by $S = \langle e_1, \ldots, e_l \rangle$; here $e_i \in \mathcal{E}$ and $e_1.T \leq e_2.T \leq \ldots \leq e_l.T$.

Filtering the Event Sequence

The event sequence that is processed by the episode mining algorithm is not the whole raw data coming from the devices. First the list of messages is cleaned and filtered to be more suitable for data mining. This filtering returns an event sequence (or more precisely, several event sequences) that we expect to be smaller than the whole data and we concentrate only on those events that are in relation with the alarms. Consequently, the aim of the filtering is to reduce the complexity. Imagine

a user who has a harmless habit that regularly generates alarms. Obviously, we want to discover the pattern of this habit to ignore its alarms in the future. In general, traffic of a network can be so heavy that the elements of the pattern get far from each other, numerous of other irrelevant events can be inserted between them. Discovering a pattern whose elements are far from each other needs much more computational capacity than discovering a pattern whose elements are next to (or very close to) each other. Hence patterns that belong to a user are easier to be discovered if we filter the original sequence of messages by a function that makes selection e.g. according to the IP addresses. In the last section we study formally the complexity-reducing effect of these filter functions.

Episodes

The habits or patterns are defined by episodes. An episode, which describes the preceding causes of a false alarm can be formalized as a conjunction of several conditions.

Definition 4 2. *Let $\mathcal{X} := \{x_1, \ldots, x_k\}$ be variables that can take events as values (event variables). We say that a triple $p(\mathcal{X}, <, \Phi)$ is an episode of size l, if $\prec$ is an order over the time of the event variables, and Φ is a conjunction of unary predicates, that refer to the attributes of the variables, so*

$$\Phi = \bigwedge_{i=1}^{l} \phi_i,$$

where the ϕ_i are given predicates applied to an attribute of an event variable.

Without loss of generality, we can presume that for any $i > j$, the inequality $x_i.T < x_j.T$ does not hold. If $\prec$ is a total order, then $p(\mathcal{X}, \prec, \Phi)$ is a *serial episode*. If the order is trivial, then the episode is *parallel*. If the episode is neither serial nor parallel, then it is *composite*.

For example the warning about a badly configured IP address we discussed earlier may be filtered by the episode $p(\mathcal{X} = \{x_1, x_2, x_3\}, \prec, \Phi)$, where

$$\Phi = (x_3.APN = \text{"idslogd"}) \wedge (x_3.CN = 404) \wedge (x_3.TNAA = 236.182.6.22) \wedge$$

$$(x_2.APN = \text{"swlogd"}) \wedge (x_2.CN = 404) \wedge$$

$$(x_2.TNAE = \text{"08:00:07:A9:B2:FC"}) \wedge (x_2.TNAA = 236.182.6.22) \wedge$$

$$(x_1.APN = \text{"eventlog"}) \wedge (x_1.SNAA = 236.182.6.22) \wedge (x_1.CN = 206)$$

and $x_1.T < x_2.T < x_3.T$. This episode describes the following situation:

- A message that a network service is started with IP address 236.182.6.22 comes from a PC.

- A gateway sends a message that the network card 08:00:07:A9:B2:FC has an invalid IP address 236.182.6.22.
- A message is sent from the network IDS that a possible alien computer is connected to the network. By the way, this message is an alarm so if x_3 is e_3, then $W(e_3)$ returns positive value.

Note that this episode filters out only this type of alarms related to this specific computer.

Definition 4 3. *The episode $p'(\mathcal{X}', \prec', \Phi')$ is a subepisode of episode $p(\mathcal{X}, \prec, \Phi)$ (denoted by $p' \subseteq p$), if there exists injection $f : X' \to X$ such that every predicate in Φ' that is applied to an $x \in \mathcal{X}'$, can be found in Φ as well applied to $f(x)$. Furthermore if $(x_i, x_j) \in \prec'$, then $(f(x_i), f(x_j)) \in \prec$ is also true. If the size of p' is less than the size of p, then p' is a proper subepisode of p. We denote this relation by $p' \subset p$.*

It is useful to restrict the episodes that we want to discover. We can presume that an episode $p(\{x_1, \ldots, x_k\}, \prec, \Phi)$ is always continuous in the sense that at least one predicate applies to each variable. Otherwise there exists an episode $q(\{x_1, \ldots, x_{k'}\}, \prec, \Phi)$, such that $k' < k$ and p, q are *isomorphic*. Episodes p and q are isomorphic if p is subepisode of q and q is subepisode of p. For every episode p there exists a continuous episode that is isomorphic to p, hence we can restrict our attention to continuous episodes. In the following, every episode is considered to be continuous.

Definition 4 4. *The episode p' is an immediate subepisode of p, if there exists no episode p'' such that $p' \subset p''$ and $p'' \subset p$.*

For example the episodes $p(\{x_1, x_2\}, \prec, \beta(x_2) \wedge \alpha(x_1))$ and $p'(\{x_1, x_2\}, \prec, \beta(x_1) \wedge \gamma(x_1))$ are immediate subepisodes of $p''(\{x_1, x_2\}, \prec, \beta(x_2) \wedge \gamma(x_2) \wedge \alpha(x_1))$. In the case of the first episode f may be the identical mapping of the variables, in the second case $f(x_1) = x_2$ suffices. Obviously, an immediate subepisode of p contains all but one predicates of p.

Invertible Parametric Predicates

The known algorithms that can handle events with attributes [12, 8] work with predefined, given predicates. An episode is a conjunction of such predicates. However, we expect more from our algorithm. It should generate the predicates themselves and then the episodes from these predicates as well. For this we provide "types" of the predicates. The predicate types are defined in the form of *parametric invertible predicates*. For example, if we think that the predicate that checks the equality of IP address may be important, then we don't want to give 2^{32} different predicates that check a given IP address, but rather provide only one general predicate.

Definition 4 5. *A parametric predicate $v : D \times T \to \{true, false\}$, which applies to the attribute $x.A$ ($A \in R$) of the variable x, is a predicate, whose value depends*

on the value of the parameter $q \in T$. The parametric predicate is invertible, if for every event e there exists a unique q such that $\nu(e.A, q)$ is true.

When we want to discover episodes that contain predicates, which apply to attributes with large domain (for example IP address), then we have to add the parametric predicate

$$\nu(x.A, q) = \begin{cases} \text{true} & \text{if } x.A = q \\ \text{false} & \text{otherwise} \end{cases}$$

to the given predicates. In the next section we present an algorithm that can handle these parametric predicates. Of course, the parameters are set in the episodes that are returned by the algorithm.

Since there are many special events, not all attributes are set or can be interpreted in the actual situation. Regarding the value of a parameter on an event, where an attribute is not applicable, we consider a predicate that applies to a missing attribute as false for any value of its parameter. Please note that with a fixed q value the predicate $\nu(x.A, q)$ is regarded as a traditional unary predicate. It is important, that a predicate with different parameter values gives different unary predicates. We also refer to a parametric predicate with a fixed parameter as a *parameter-predicate pair* and a predicate with non-fixed parameter as a *predicate type*.

4.1. Support and Alarm Support

Definition 4 6. *The event sequence $S = \langle e_1, \ldots, e_l \rangle$ contains the episode $p(\{x_1, \ldots, x_k\}, \prec, \Phi)$, if there exists different events $e_{j_1}, \ldots e_{j_k} \in S$ such that in $p(\{e_{j_1}, \ldots e_{j_k}\}, \prec, \Phi)$ Φ is true and $\prec$ holds.*

If the sequence S contains episode p, then we say "p *occurs* in S" or "p is true in S".

Definition 4 7. *An m window of the event sequence $S = \langle e_1, \ldots, e_l \rangle$ that ends with an alarm of type w is an event sequence $S' = \langle e_j, \ldots, e_{j+m-1} \rangle$, where $1 \leq j \leq l - m + 1$ and $W(e_{j+m-1}) = w$.*

The set of windows of S defined above is denoted by $aw(S, m, w)$.

Definition 4 8. *The support of episode p in $aw(S, m, w)$ is the number of windows that contain p:*

$$supp_{S,m,w}(p) = \left|\{S' \in aw(S, m, w) | S' \text{ contains } p\}\right|$$

An episode is *frequent*, if its support is higher than a given support threshold (min_supp), otherwise it is *infrequent*.

The frequent episodes are not necessarily important in practice. There can be many that have no connection with alarm situation, but they occur in many windows

that end with an alarm. Such universally frequent episodes are out of interest in our context.

If episode $p(\{x_1, \ldots, x_k\}, \prec, \Phi)$ is contained in a window such that x_1 generates alarm, then this episode may be important because the conditions described by the episode may have been considered improperly to be an attack by some security device. We shall focus on such episodes and their occurrences.

Let us define the term *alarm support*.

Definition 4 9. *The alarm support of a serial episode* $p(\{x_1, \ldots, x_k\}, \prec, \Phi)$ *in the* $aw(S, m, w)$ *is defined by* $\text{alarm\_supp}_{S,m,w}(p(\{x_1, \ldots, x_k\}, \prec, \Phi)) = \big|\{S' = \langle e_1, \ldots, e_m\rangle \in aw(S, m, w) | \exists e_{j_2}, \ldots, e_{j_k} \in S' \setminus \{e_m\}$ *different events such that in* $p(\{e_m, e_{j_2}, \ldots e_{j_k}\}, \prec, \Phi)\Phi$ *is true* $\}$

An episode is *alarm frequent*, if its alarm support is greater than a given threshold (min_supp).

Definition 4 10. *The expression* $p[m, w] \Rightarrow \{real\ alarm, false\ alarm\}$ *is an episode rule, if p is an episode, m is an integer number and w is an alarm type. The interpretation of the rule* $p[m, w] \Rightarrow$ *false alarm is the following: if p occurs in an event sequence of S of width m that ends at an alarm of type w, then the alarm is false, otherwise it is a real alarm.*

Our final goal is to determine episode rules that filter out false alarms. The data mining algorithm discovers alarm frequent episodes and an expert (security specialist) sets the right-hand-side (false alarm or real alarm) of the rules. Obviously this step cannot be automated since it requires domain knowledge (knowledge about the local network, about the security devices, about the users, etc.). We expect that determining the alarm frequent episodes can help the security specialists to handle the vast number of alarms effectively.

4.2. The Aim of Data Mining

After clarifying the basic definitions we can set the model and define the aim of data mining in the Remote Supervision System. Then is given a filtered event sequence S that has to be processed off-line. The invertible, parametric, unary predicates $\alpha(x.A, q_\alpha)$, $\beta(x.B, q_\beta)$, ..., the window width (m) and the alarm type (w) are provided by an expert (system administrator, security specialist). Based on this set of parameters we have to determine the alarm-frequent serial episodes in S.

Our first task is to determine the values of the parameters so that from the predicates obtained, frequent episodes can be built. The output of the data mining module (the alarm-frequent episodes) is examined exhaustively by the expert, who finally approves the false alarm filtering episode rules. After the episode rules are set, the on-line processing of the network traffic can be started. If an alarm of type w arrives and its preceding m wide window contains episode p (more precisely the episode rules $p[m, w] \Rightarrow false\ alarm$ exists), then the alarm is determined to be false and automatically filtered out.

We restrict our search to serial episodes, however the model and the algorithm can be extended to handle parallel episodes as well, at the expense of performance degradation. These generalizations are not discussed in this paper. In the following event variables are always referred as $x_1, \ldots, x_k$ and the order on time is $x_k.T < \ldots < x_1.T$. If only continuous, serial episodes are concerned, the conjunction of the predicates unambiguously determines the episode itself. So for the sake of simplicity, an episode is understood to be the conjunction of the predicates (we write $p = \bigwedge_{i=1}^{l} \phi_i$).

5. The Algorithm ABAMSEP

The detailed algorithm ABAMSEP (APRIORI-Based Algorithm for Mining Serial Episodes with parametric Predicates) is based on algorithm APRIORI [1]. It discovers frequent serial episodes and handles invertible parametric predicates. The pseudo-code is given in the next page.

The algorithm has two phases. First, the parameters of the interesting predicates are determined, and those windows are found where alarm-frequent episodes may occur. Next, these windows are scanned and the frequent episodes are discovered.

So in the beginning we determine those predicate-parameter pairs that can be true on an event that generated alarm of type w. From these predicates we can immediately generate alarm-frequent episodes consisting of only one condition. The occurrences of these episodes will be the last events of those windows, where alarm-frequent episodes can be found. This set of windows is a subset of $aw(S, m, w)$ so let us denote it by $aw'(S, m, w)$ ($aw'(S, m, w) \subseteq aw(S, m, w)$).

In order to determine frequent episodes in $aw'(S, m, w)$ we need some further evaluations. The following property holds for every frequent episode.

Property 5 1. *If an episode p is frequent in some windows of the event sequence S, then all subepisodes of p are also frequent in these windows.*

This follows from the fact that if an episode occurs in an event sequence, then the subepisodes occur as well. This property suggests to adapt the scheme of algorithm APRIORI.

We scan every event of $aw'(S, m, w)$ one-by-one and determine those predicate-parameter pairs that are true on the actual event. Notice, that a predicate-parameter pair can be regarded as an episode of size 1. Every predicate-parameter pair has a counter, and if the pair is true on an event we increase this counter. The counter can be increased just once in a window although it may be true on more than one event of the window. After reading through the event sequence we select those predicate-parameter pairs that have support higher than *min_supp*. The frequent episodes of size 1 will consist of them. In the following only these frequent predicate-parameter pairs are considered. As we mentioned earlier, these predicates after the parameters are fixed, can be regarded as traditional predicates. Without loss of generality, we assume that these predicates are ordered.

Algorithm 1 algorithm ABAMSEP

Require: $S = \langle e_1, \ldots, e_l \rangle$: event sequence ordered by time,
m : width of the window.
min_supp : support threshold
$\alpha, \beta, \ldots$: parametric predicates
w : alarm type
Ensure: P^w : set of the alarm-frequent episodes

I. PREPROCESSING:
for all event $e_i \in S : W(e_i) = w$ **do**
determine those predicate-parameter pairs that are true on e_i
end for
determine $rep\_aw'(S, m, w)$
generate C_1
$i \leftarrow 1$

II. MAIN CYCLE:
repeat
determine the support of elements of C_i
$P_i \leftarrow \{c | c \in C_i, c.support \geq min\_supp\}$,
delete C_i
$C_{i+1} \leftarrow$ candidate_generation(P_i)
for all $p \in P_i$ **do**
if $p.support\_w \geq min\_supp$ **then**
$P_i^w \leftarrow P_i^w \cup p$
else
delete p
end if
end for
OPTIONAL STEP: delete nonmaximal episodes from P_{i-1}^w
$i \leftarrow i + 1$
until $\{|C_i| > 0$ AND $P_{i-1}^w > 0\}$
$P^w \leftarrow \bigcup_i P_i^w$

The next step is to generate candidate episodes of size 2 from frequent episodes of size 1. An episode can be *candidate* if all of its subepisodes are frequent. Note that this is just a necessary condition for an episode to be frequent, therefore for each candidate the support should be determined in an additional step. To do this, after candidate generation the support counting method is evoked. In general candidate episodes of size $i + 1$ are generated from the frequent episodes of size i. The candidate generation is detailed in section 5.1. After the candidate generation, we need only the alarm-frequent episodes of size i, the others can be deleted. The next step is to determine the support of the candidates of size $i + 1$, and delete those that have support less than the support threshold (min_supp). By repeating this

process ($i = 1, 2, \ldots$) we can determine all alarm-frequent episodes. The algorithm terminates, if no new candidate is generated.

The output of the algorithm is the set of alarm-frequent episodes. The problem with this solution is that too many useless episodes are generated. For example an alarm-frequent episode of size 5 and variable number 5 has $2^5 - 2$ subepisodes that are also alarm frequent. Consequently, it is useful to return to the expert only the maximal (with respect to $\subseteq$) alarm-frequent episodes.

5.1. Candidate Generation

The candidate generation is similar to the method proposed in algorithm APRIORI. The differences stem from the fact that APRIORI works with itemsets while here we are working with episodes. Candidate generation has two phases: *join* and *prune*.

5.1.1. Join Phase

A candidate c of size $i+1$ is generated from two frequent episodes (p_1,p_2) of size i. Without loss of generality we can assume that p_1 has l variables, p_2 has k variables and $l \geq k$. We join the two episodes, if by deleting the predicate ($\mu(x_l.A)$) from p_1 that has the largest order among those that apply to the variable x_l, we obtain the same episode as we get if we delete the predicate ($\nu(x_k.B)$) from p_2, which has the largest order among those that apply to the variable x_k. Thus, p_1 and p_2 must have $i - 1$ common predicates that apply to the variables $x_1, \ldots, x_k$. Three different cases are possible:

1. p_1 is equal to p_2 (so $l = k$ and $\mu = \nu$). We join an episode with itself if only one predicate applies to x_l ($= x_k$). If this condition holds, then we generate the candidate $c := p_1 \wedge \mu(x_{k+1}.B)$.
2. If $p_1 \neq p_2$ and more than one predicates apply to the variable x_k in p_2, then $c := p_1 \wedge \nu(x_k.B)$ is the candidate. Obviously if predicate ν applies to x_k in p_1 (even with different parameter), then we can immediately delete the candidate. The reason for this is, that if the parameters are the same, then the candidate is not of size $i + 1$, otherwise it will not occur in any window (invertibility).
3. Otherwise $l = k$ and only one predicate applies to the variable x_l in p_1, since the episodes are continuous (each variable is contained in at least one predicate). In this case 3 candidates are generated: $c' := p_1 \wedge \nu(x_k.B)$, $c'' := p_1 \wedge \nu(x_{k+1}.B)$, $c^* := p_2 \wedge \mu(x_{k+1}.A)$. Again, if predicate ν applies to x_k in p_1, then c' can be deleted.

The episode pair (p_1, p_2) generates the same candidate as the pair (p_2, p_1) does. We suppose that an order on the episodes can be defined (for example lexicographic order that is defined based on the ordering of the predicates). Two episodes are joined if and only if p_2 is larger than p_1 with respect to the order.

Let us consider some examples for the join phase (the attributes of the variable are omitted for the sake of simplicity).

Table 1. Example for joining

p_1	p_2	*candidate*
$\gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	$\alpha(x_1) \wedge \beta(x_1) \wedge \delta(x_1)$	not joinable
$\gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	$\gamma(x_3) \wedge \delta(x_2) \wedge \alpha(x_1)$	not joinable
$\gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	$\beta(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$	$\gamma(x_3) \wedge \beta(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$
$\beta(x_2) \wedge \gamma(x_2) \wedge \alpha(x_1)$	$\beta(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$	$\beta(x_2) \wedge \gamma(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$
$\gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	$\delta(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	$\gamma(x_3) \wedge \delta(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$
		$\delta(x_4) \wedge \gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$
		$\gamma(x_4) \wedge \delta(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$

It is instructive to look at the possible candidates generated from the pair $p_1 = \mu(x_1.A)$, $p_2 = \nu(x_1.B)$. Here $l = k = 1$ and if $\mu \neq \nu$, then the candidate pairs are :$\mu(x_1.A) \wedge \nu(x_1.B)$; $\mu(x_2.A) \wedge \nu(x_1.B)$; $\nu(x_2.B) \wedge \mu(x_1.A)$. If $\mu = \nu$, then the first candidate is deleted. We remark that the two episodes are always immediate subepisodes of the candidate generated.

5.1.2. Prune Phase

The objective of this phase is to prune the candidates that have an immediate subepisode of size i that is infrequent, i.e. it is not among the frequent episodes.

Let us consider some examples with two different frequent episode sets of size 3, which are given in Table 2. Frequent episodes are found in the first column, the candidates after the join phase in the second. If the candidate is pruned, then *yes* can be found in the 3^{th} column, otherwise *no*. If the candidate is pruned, then its immediate subepisode that is infrequent is shown in the 4^{th} column.

5.2. Determining the Support

To determine the support of the candidate episodes, we present a simple algorithm that is easy to implement. Using tries or hashing techniques can further accelerate it. For details see [1, 5, 6, 3, 4].

The support of the candidates has to be calculated so that the frequent ones can be selected, and the infrequent ones are pruned. Each window of $aw'(S, m, w)$ has to be examined and those candidate episodes have to be found that are true in the window. A candidate episode of size k occurs in a window if there exists k different events such that all predicates that apply to the variable x_j are true on the j^{th} event ($1 \leq j \leq k$).

Table 2. Example for pruning

frequent episodes of size 3	candidates after join	is pruned	infrequent subepisodes
$\gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$			
$\beta(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$	$\gamma(x_3) \wedge \beta(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$	yes	$\gamma(x_2) \wedge \beta(x_1) \wedge \delta(x_1)$
$\gamma(x_3) \wedge \delta(x_2) \wedge \alpha(x_1)$			
$\gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$			
$\delta(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	$\gamma(x_3) \wedge \delta(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	yes	$\gamma(x_2) \wedge \delta(x_2) \wedge \alpha(x_1)$
$\delta(x_3) \wedge \gamma(x_2) \wedge \alpha(x_1)$	$\delta(x_4) \wedge \gamma(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	no	
$\delta(x_3) \wedge \gamma(x_2) \wedge \beta(x_1)$	$\gamma(x_4) \wedge \delta(x_3) \wedge \beta(x_2) \wedge \alpha(x_1)$	yes	$\gamma(x_3) \wedge \delta(x_2) \wedge \beta(x_1)$

We use a *greedy algorithm* to find the episodes of size k that occur in the given event sequence $S = \langle e_1, \ldots, e_m \rangle$. Let us read through the event sequence from the end to the beginning. A finite automaton can represent each episode. All automata are in the initial (0^{th}) state before we process the event sequence. When event e is considered, the automaton of the candidate episode $p(x_1, \ldots, x_k)$ jumps to the next state from state i, whose predicates that apply to the variable x_{i+1} are true on e. Otherwise we don't jump. The state k is the accepting state.

A boolean variable has to be added to each automaton. It is set when the last (from the end the first) event is processed. Its value is *true* for those automata that jumped to the first state after this event (the one that generated alarm) is processed. Otherwise the variable is set to *false*.

When the left end of the window is reached, all instances of the automata are deleted. We increase the support of those candidates, whose automata is in the accepting state. If in addition the boolean variable is true, then the alarm support is also incremented.

In order to implement the automata a reference to the episode and a variable that stores the state index of the automata need to be handled. This two pieces of data and the boolean variable can be stored in a list. We add a triple to this list only when their imaginary automaton jumps to the first state.

Please observe that the (costly) disc operations are carried out in the first two steps of the preprocessing, and in the first step of the main cycle.

5.3. An Optional Step

A partially ordered set can be built from the alarm episodes, where the alarm-frequent episodes are under a border. The elements of the border are the maximal alarm-frequent episodes. It would be useless to present all the frequent episodes to the user since the border has fewer elements, and any frequent episode can be

obtained from the maximal ones.

The alarm-frequent episode p is *maximal*, if there exists no frequent episode p' such that p is a proper subepisode of p'. Maximal episodes can be filtered in two ways. First, we can filter the output of algorithm ABAMSEP; second we can weave the filtering process into the algorithm. The advantage of the second solution is that it decreases the memory need, since less episodes are stored. We applied this approach in the implementation.

After infrequent candidates of size i are deleted, those frequent episodes of size $i-1$ can also be pruned that are subsets of some frequent episode of size i. Hence, if the user is interested only in the maximal episodes, then the following line have to be inserted into the main cycle (by definition $p_0^{(w)}$ equals to $\emptyset$!)

Algorithm 2 Removing nonmaximal episodes

```
for all p1 ∈ P^w_{i-1} do
  for all p2 ∈ P^w_i do
    if (p1 ⊂ p2) then
      P^w_{i-1} ← P^w_{i-1} \ p1
    end if
  end for
end for
```

5.4. Completeness and Redundancy

By the following lemma the greedy algorithm (presented in section 5.2) properly counts the support of an episode.

Lemma 5 1. *The greedy algorithm finds all episodes that occure in a given event window.*

Proof. Suppose that there exists a candidate episode $p(x_1, \ldots, x_k)$ that is true in event window $S' = \langle e_1, \ldots, e_m \rangle \in aw'(S, m, w)$, but the greedy algorithm did not find it to be true. According to the assumption there exist different events $e_{i_1}, \ldots, e_{i_k} \in S'$ such that $p(e_{i_1}, \ldots, e_{i_k})$ is true. Since the greedy algorithm did not find the episode after processing S', the automaton that represents the episode stopped in a state (k'), with $k' < k$. Thus there exist different events $e_{i'_1}, \ldots e_{i'_{k'}} \in S'$, such that all predicates that apply to x_1 are true on event $e_{i'_1}$, and all predicates that apply to x_2 are true on event $e_{i'_2}$ and so on. The algorithm finds these events starting from the back, and due to the greedy nature of the method the relations $i_1 \geq i'_1$, $i_2 \geq i'_2$, ..., $i_{k'} \geq i'_{k'}$ must hold. We are searching for occurrences of serial episodes, it follows from the above that there exists an event $(e_{i_{k'+1}})$ that is in the window, it is before $e_{i_{k'}}$, and all predicates that apply to variable $x_{k'+1}$ are true on it. The greedy algorithm will clearly find this event. This is a contradiction. □

Theorem 5 2. *The algorithm ABAMSEP is complete: it finds all alarm-frequent episodes.*

Proof. The proof is based on induction of the size of episodes. In the first step we check each event in windows $aw'(S, m, w)$, and all predicates are found from whom a frequent episode of size 1 can be built. Let us suppose that we found all frequent episodes of size $l \geq 1$, but an episode p of size $l + 1$ and variable number k was not found. According to Lemma 5 1, if a frequent episode is generated as a candidate episode, then its support is calculated exactly. If p were not found to be frequent, then it should not be generated at all.

3 different cases can occur: (1) at least two predicates apply to x_k, (2) one predicate applies to x_k, and at least two to x_{k-1}, (3) one predicate applies to x_k and one to x_{k-1}.

In the first case p is in the form of $\alpha(x_k) \wedge \alpha'(x_k) \wedge p'$, where $\alpha \neq \alpha'$. However, p should have been generated by joining episodes $p' \wedge \alpha(x_k)$ and $p' \wedge \alpha'(x_k)$).

In the second case p is $\alpha(x_k) \wedge \alpha'(x_{k-1}) \wedge p''$, where at least one predicate applies to x_{k-1} in p''. In this case p is obtained if episodes $p'' \wedge \alpha(x_k)$ and $p'' \wedge \alpha'(x_{k-1})$ are joined.

In the third case $p = \alpha(x_k) \wedge \alpha'(x_{k-1}) \wedge p^*$, where the largest variable in p^* is x_{k-2}. Here, by joining $p^* \wedge \alpha(x_{k-1})$ and $p^* \wedge \alpha'(x_{k-1})$ episodes we obtain p. If $\alpha = \alpha'$, then $p_1 = p_2$, hence it is a case of self-join, where the condition holds (ie. only one predicate applies to the largest variable). Each case leads to contradiction, hence the statement follows. □

Consequence 5 1. *Each candidate is generated once in algorithm ABAMSEP.*

Proof. It is immediate from the proof of the Theorem 5 2. For any candidate we can uniquely determine the two subepisodes that generated it, hence it cannot happen that two different episode pairs generate the same candidate. □

Candidate generation algorithms that do not generate the same candidates in different ways are called *nonredundant* in the literature. Nonredundant candidate generation is a requirement of an efficient frequent pattern mining algorithm.

5.5. Theoretical Remarks on the Time and Memory Need of the Method

Let us denote the size of the largest alarm episode by $|p_{max}|$, and as earlier, the length of the sequence by l and the width of the window by m. We analyse the time and memory needs with the assumption that the event sequence is on disk.

Operations in the memory are much faster than operations on the disk, therefore disk access is of primary concern. It is easy to determine how many times does the algorithm read through the event sequence. ABAMSEP is a levelwise algorithm, it reads through the database as many times as the size of the largest episode. If the number of the given predicates is u, then this is at most $m \cdot u$, because the number

of variables cannot be more than the size of the window and a predicate can apply only once to any variable (this is a consequence of invertibility). We infer that the number of disk access is linear in the parameters l, m and u.

The candidates and the actual window are stored in the memory. Insufficient space in the main memory slows down very sharply the processing of the candidates. It is impossible to estimate, which episode counter should be increased before processing a window, hence swapping the candidates to and from the disk would take a lot of time. Therefore algorithm ABAMSEP does not possess the 'graceful degradation' property, similarly to all other frequent patter mining algorithms [2, 13, 1, 14, 9, 16].

Procedure that finds the supported candidates in a window is executed as many times during a single reading of the sequence as many elements $aw'(S, m, w)$ has. However, this can be slower than reading in the sequence from the disk. We know that when an event is processed, we have to check the state of all instances of the automata, hence support determination is proportional to the number of the candidates. If $|p_{max}| < m$, then in the worst case just the number of episodes with $|p_{max}|$ variables can be $(u \cdot |p_{max}|)^{|p_{max}|}$, since $u \cdot |p_{max}|$ different predicates can apply to each variable. If $|p_{max}| \geq m$, then the number of the candidates can be even more than u^m. Consequently, determining the support can be proportional to $l \cdot u^m$.

This exponential growth is not as bad as it seems. Every data mining algorithm, where the aim is to find frequent objects, shows similar characteristic [1, 2, 14, 9, 16, 13]. Fortunately, the theoretical bounds on time and memory complexity and real performance are often far from each other. When the algorithm is slow, then the parameters are probably set improperly and too many episodes are generated. Generating too many episodes should be avoided since these episodes have to be examined one-by-one by an expert. The test results presented in the next section supports the following hypothesis of ours: when the mining yields manageable results, then the algorithm finds the episodes in acceptable time.

6. Experimental Results

Implementation of the proposed algorithm was developed within the framework of a research project supported by the Hungarian Ministry of Education in cooperation with ICON Ltd, a Hungarian IT specialized company. During this project a common message format was elaborated. We collected large volume of log files from different security products and the execution experiments come from this work.

In the figures the influence of parameters on the run-time and the candidate number can be seen. The parameters examined were the support threshold, the width of window and the number of invertible predicates. In the previous section we showed that theoretically there is an exponential growth in the run-time.

In the test system events were generated by 20 different devices. Each event had 11 attributes. The first attribute returned the type of the event, i.e.: entry of a

computer to the network, signal from a virus checker, entry of a user to the network, or system event. Other attributes were: process name, creation time, classification name, detection time, source node address, source service port, target node address, target service port, target file and target file path.

The raw database consisted of 2400 events, the filtered sequence was of length 600. 100 alarms were hidden in the data: 50 randomly and 50 were inserted with predefined events. These events were generated so that episodes could be retrieved from them. These episodes had 4, 5 or 6 variables and the size of them varied from 18 to 28. Random events were inserted between the predefined events such that the episodes could be discovered by using windows of size 10.

We implemented the algorithm on a Linux operating system (Red Hat Linux version 7.2). The tests ran on a configuration with Athlon XP 1700+ processor and 256 DDR operative memory.

The ABAMSEP algorithm worked properly. If the window size parameter was set to 10 or more, then each episode was successfully discovered. Obviously if the window size was less than 10, then some episodes were infrequent, hence left unnoticed.

Further test results and the description of the test environment can be found in [11].

Before presenting the results, we would like to draw the reader's attention to a very important feature. In most data mining applications, we are used to low correlation of a large number of items. Even if the size of dataset is very large, the number of frequent items is still manageable. Unfortunately this is not the case with security events. There are a few types of events, only the values of parameters change.

Let us first examine the number of candidates of different sizes (*Fig. 1*). We can see that by increasing the number of predicates the number of the candidates rapidly grow. After it reaches the peak the number of candidates decreases. Similar characteristics were observed and analysed in the case of frequent itemset mining [7]. This roots from the similarity with respect to the inclusion anti-monoton, i.e. if a set/episode is a candidate, then all its subsets/subepisodes are candidates as well.

The *Figs. 2, 3, 4* show how the window width, the predicate number and the support threshold affect the run-time. Please note, that the exponential increase in the run-time roots from the problem itself and not from the solution. The search space is exponential in the number of predicates, thus if we set min_supp to zero, then all possible episodes become frequent and simply outputting the results requires exponential operation. The exponential run-time characteristic is present in all frequent pattern mining algorithms.

We can see that the retrieval speed is getting lower if we increase the width of the window or the number of the predicate types, or decrease the support threshold. One may find the run-times too slow, however we have to emphasize that this primary implementation did not include any accelerating techniques. Simple data structures (like lists) were used where not even ordering and binary search was implemented. Evidence can be found in the literature that by using sophisticated data structures (like trie) the run-time drops to its fraction [6, 3, 4].

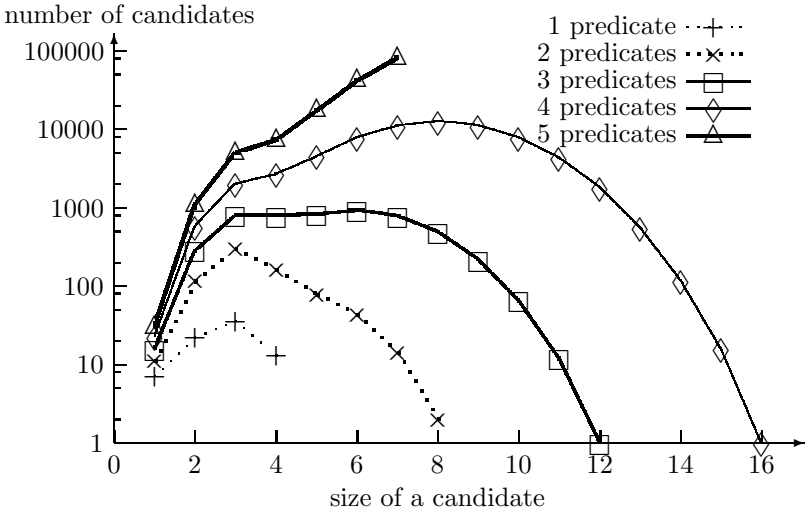

Fig. 1. Number of candidates of different size

7. Consequences and Future Research

The aim of this research was to make an in depth investigation on the improvement possibilities of a Remote Supervision System. Such systems seem to be the most effective systems in computer security, therefore this kind of research is of importance. We intended to battle and defeat the most dangerous enemy, the large number of possibly false or unimportant alarms. We have won the battle, however, the end of the war is still far away. In our work we proved that data mining is a powerful weapon. An efficient and scalable algorithm was proposed that makes it possible to automatically filter many false alarms.

Several simplifications have been made in our model in order to keep the complexity of the algorithm acceptable even when large event sequences are processed. Our solution can be improved in many ways. Episodes can be generalized so that more complex patterns can be found. The efficiency of our existing algorithm can also be improved. Here we shortly discuss some avenues of further research.

- Algorithm ABAMSEP is searching for serial episodes only. However, parallel and more complex episodes are also of interest. Candidate generation and support count can be easily extended to handle parallel episodes. The time complexity immediately increases as soon as more general episodes are searched for. We suggest that a middle way solution i.e. serial episode that is made of small parallel episodes could be still manageable.
- Episodes were defined as sets of conditions where the conditions were given by unary predicates. As soon as higher level predicates (for example binary) are allowed in the conditions neither the candidate generation nor the support

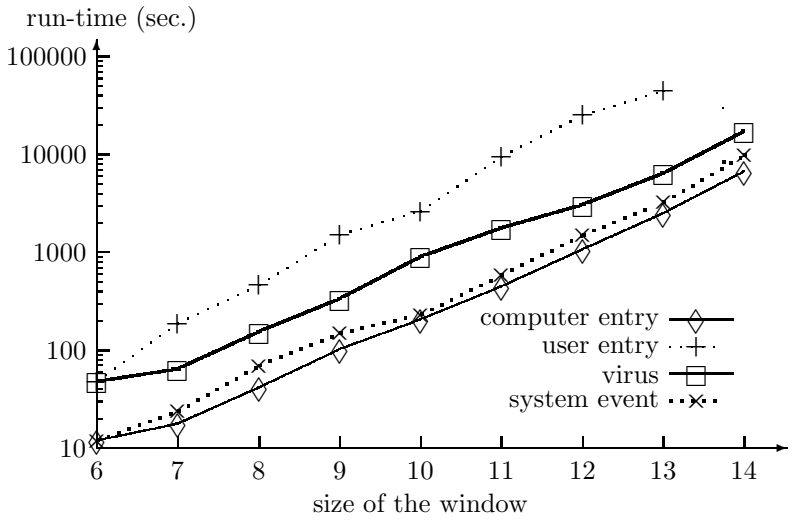

Fig. 2. Run-time as the function of the window size

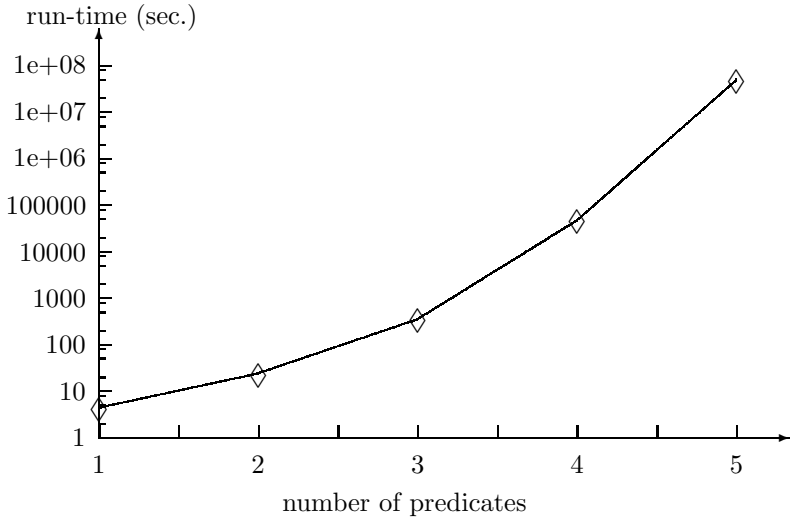

Fig. 3. Run-time as the function of the predicates' number

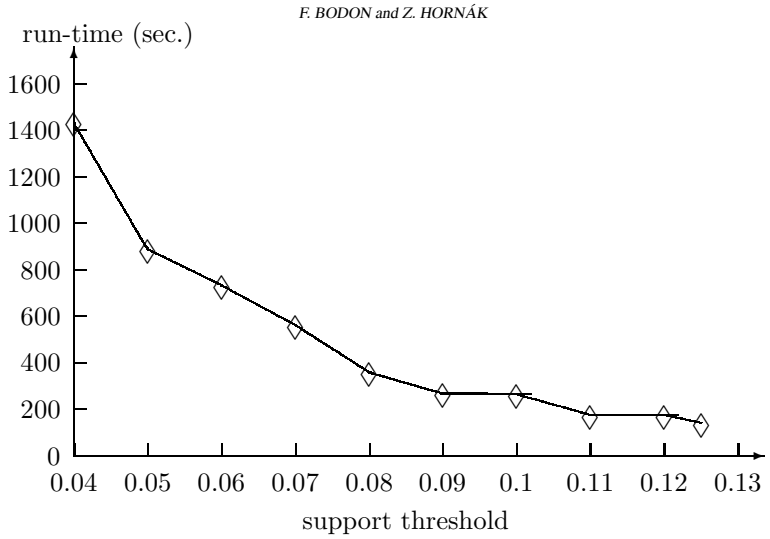

Fig. 4. Run-time as the function of the support threshold

count could be solved so easily.

- We have proposed that a filtered sequence and not the raw data should be processed by the data mining algorithm. Filters could be efficiently implemented in the system, and produce the filtered sequence very fast. We know that some binary predicates can be substituted if a proper filter is used. For example the binary predicate $x_1.IP = x_2.IP$ is implicitely included in the episodes if we filter the raw data according to the same IP addresses. However there are binary predicates that cannot be substituted by any reasonable filter. A theoretical and practical research on the limitations and realization of the filters is still ahead.
- The prototype implementation of our algorithm ABAMSEP does not include any techniques for fast operation. Obviously, support count could be speeded up greatly by using tries or hash-trees to store the candidates.

We see that many interesting and important open problems can be posed. We believe that we have proved here that data mining algorithms can be applied in the security supervision of IT systems by discovering the sources of false alarms. If the number of false alarms can be decreased thanks to the rules determined, then the sensitivity parameters of security devices can be set to high and the number of recognized attacks will increase as well.

References

[1] AGRAWAL, R. – SRIKANT, R., Fast Algorithms for Mining Association Rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, ed., *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pp. 487–499. Morgan Kaufmann, 12–15 1994.

[2] AGRAWAL, R. – SRIKANT, R., Mining Sequential Patterns. In P. S. Yu and A. L. P. Chen, ed., *Proc. 11th Int. Conf. Data Engineering, ICDE*, pp. 3–14. IEEE Press, 6–10 1995.

[3] BODON, F., A Fast Apriori Implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[4] BODON, F., Surprising Results of Trie-based Fim Algorithms. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 1. November 2004.

[5] BODON, F. – RÓNYAI, L., Trie: An Alternative Data Structure for Data Mining Algorithms. *Computers and Mathematics with Applications*, 2002.

[6] BORGELT, C., Efficient Implementations of Apriori and Eclat. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[7] GEERTS, F. – GOETHALS, B.– BUSSCHE, J. V. D., Tight Upper Bounds on the Number of Candidate Patterns. *ACM Trans. Database Syst.*, 30(2):333–363, 2005.

[8] HATONEN, K. – KLEMETTINEN, M. – MANILLA, H. – RONKAINEN, P. – TOIVONEN, H., Knowledge Discovery from Telecommunication Network Alarm Databases. In S. Y. W. Su, ed., *Proceedings of the twelfth International Conference on Data Engineering, February 26–March 1, 1996, New Orleans, Louisiana*, pp. 115–122, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[9] HUHTALA, Y. – KINEN, J. – PORKKA, P. – TOIVONEN, H., Efficient Discovery of Functional and Approximate Dependencies Using Partitions. In *ICDE*, pp. 392–401, 1998.

[10] KERÉNYI, K., Applying Data Mining Methods in Computer Remote Inspection. Master's thesis, Department of Measurement and Information Systems, Budapest University of Technology and Economics, 2002.

[11] KUCZY, CS., Filtering False Alarms with Data Mining Methods in the Case of Computer Networks. Master's thesis, Department of Measurement and Information Systems, Budapest University of Technology and Economics, 2003.

[12] MANNILA, H. – TOIVONEN, H., Discovering generalized episodes using minimal occurrences. In *Knowledge Discovery and Data Mining*, pp. 146–151, 1996.

[13] MANNILA, H. – TOIVONEN, H. – VERKAMO, A., Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[14] SILVERSTEIN, C. – BRIN, S. – MOTWANI, R., Beyond Market Baskets: Generalizing Association Rules to Dependence Rules. *Data Mining and Knowledge Discovery*, 2(1):39–68, 1998.

[15] SRIKANT, S. – AGRAWAL, R., Mining Sequential Patterns: Generalizations and Performance Improvements. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, ed., *Proc. 5th Int. Conf. Extending Database Technology, EDBT*, volume 1057, pp. 3–17. Springer-Verlag, 25–29 1996.

[16] ZAKI, M., Sequence Mining in Categorical Domains: Incorporating Constraints. In *CIKM*, pp. 422–429, 2000.