JYVÄSKYLÄ STUDIES IN COMPUTING 22

Alexandr Seleznyov

# An Anomaly Intrusion Detection System Based on Intelligent User Recognition

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Agora-rakennuksessa (Ag Aud. 2)
syyskuun 21. päivänä 2002 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in the Building Agora, (Ag Aud. 2), on Semptember 21, 2002 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2002

# An Anomaly Intrusion Detection System Based on Intelligent User Recognition

JYVÄSKYLÄ STUDIES IN COMPUTING 22

Alexandr Seleznyov

# An Anomaly Intrusion Detection System Based on Intelligent User Recognition

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2002

Editors
Seppo Puuronen
Department of Computer Science and Information Systems, University of Jyväskylä
Pekka Olsbo, Marja-Leena Tynkkynen
Publishing Unit, University Library of Jyväskylä

URN:ISBN:9513912876
ISBN 951-39-1287-6 (PDF)

ISBN 951-39-1192-6 (nid.)
ISSN 1456-5390

# ABSTRACT

Seleznyov, Alexandr
An Anomaly Intrusion Detection System Based on Intelligent User Recognition
Jyväskylä: University of Jyväskylä, 2002, 186 p.
(Jyväskylä Studies in Computing,
ISSN 1456-5390; 22)
ISBN 951-39-1287-6
Finnish summary
Diss.

Recently computer systems have become a critical part of network-connected system, possessing essential economic and human values to individuals and organizations. This key role of the systems has increased the requirements for their protection. They have to be more resistant against malicious activities. Intrusion detection is aimed at detecting and preventing such activities. It forms the last line of defense in the overall protection scheme of a computer system. It is useful not only in detecting successful breaches of security, but also for monitoring attempts to breach security, which provides important information for timely countermeasures. Thus, intrusion detection systems are useful even when strong preventive steps are taken to protect computer systems. In anomaly detection, computer systems compare current events with expected or predicted events. In this thesis, a typical decision problem in anomaly detection is transformed into three scenarios: what event is going to happen in the future, when, and how much danger it may cause. In the thesis we concentrate on the first two scenarios. We use three layers of information representation for constructing a time-probabilistic network for online event learning and prediction. Online prediction usage in our approach provides us with the possibility to automatically reveal possible dangerous sequences of events, while it does not arouse the person's suspicions. This thesis describes an approach aimed at catching and representing peculiarities of user behavior and recognizing them later. It also presents an architecture for the intrusion detection system based on the temporal-probabilistic network approach. This hybrid architecture combines anomaly and misuse detection models attempting to make the resulting architecture free of the specific disadvantages of these models when used separately. Based on the described architecture and devised user verification approach a prototype was built. The prototype was tested on a number of real users proving viability of the approaches discussed in this thesis.

**Keywords**: network security, intrusion detection, online learning, probabilistic network, behavioral pattern, user verification

**ACM Computing Review Categories**

C.2.3 Computer Systems Organization: Computer-Communication Networks: Network Operations: *Network monitoring, Public networks*
C.5.3 Computer Systems Organization: Computer System Implementation: Microcomputers: *Personal computers, Workstations*
D.4.6 Software: Operating Systems: Security and Protection: *Access controls, Information flow controls, Invasive software*
I.2.4 Computing Methodologies: Artificial Intelligence: Knowledge Representation Formalisms and Methods: *Relation systems, Temporal logic*
I.2.6 Computing Methodologies: Artificial Intelligence: Learning: *Knowledge acquisition, Parameter learning*
I.5.1 Computing Methodologies: Artificial Intelligence: Pattern Recognition: *Statistical*

**Author's address** Alexandr Seleznyov
Department of Computer Science and Information Systems
University of Jyväskylä
P.O.Box 35, FIN-40351 , Jyväskylä, Finland
E-mail: alexandr@it.jyu.fi

**Supervisor** Seppo Puuronen
Department of Computer Science and Information Systems
University of Jyväskylä
Finland

**Reviewers** Dr. Steven Furnell
Department of Communication and Electronic Engineering
University of Plymouth, UK

Prof. Ravi Sandhu
Department of Information and Software Engineering
George Mason University, USA

**Opponent** Dr. Dipankar Dasgupta
Computer Science Division
University of Memphis, USA

## ACKNOWLEDGMENTS

I am deeply indebted to my academic adviser, Seppo Puuronen for his constant help, support, and guidance. Special thanks to Vagan Terziyan for his time and energy in teaching me how to perform research and survive as a researcher in academic community; for his encouragement to overcome the temptations to throw it all away. Without their valuable contribution I would not be able to finish this work.

Many thanks to COMAS graduate school, which has provided funding for these studies.

Computer Science and Information Systems department of the University of Jyväskylä was very helpful in providing me with excellent working environment, without which it would be difficult to perform my research. I would also like to thank professor Kalle Lyytinen for his valuable comments and support during these studies.

I am thankful to the examiners of my Ph.D. thesis, Dr. Steven M. Furnell (Network Research Group, University of Plymouth, United Kingdom) and Dr. Ravi Sandhu (Department of Information and Software Engineering, George Mason University, USA) for their time and insightful comments. I would like to sincerely thank Dr. Dipankar Dasgupta (Mathematical Science Department, University of Memphis, USA) for being my opponent.

This work has greatly benefited from extensive numerous reviews of Dr. Steven M. Furnell, which he made for my Licentiate and Ph.D. theses. His contribution to the quality of this study is very much appreciated.

Many thanks to Oleksiy Mazhelis who was heavily involved into the process of development and testing of the software prototype used in this work. Without his help it would have taken significantly longer time to finish this thesis.

I thank professors Seppo Puuronen and Jari Veijalainen for their valuable help with preparation of the Finnish summary.

Finally, I would like to thank my parents for encouraging me to begin this work. I also thank my wife Katja for tolerating many evenings and weekends spent in writing this thesis.

Jyväskylä
August 2002

# CONTENTS

## List of Figures

## List of Tables

# 1 INTRODUCTION

This chapter presents an introduction as well as a brief description of our work. It outlines the research area, provides the motivation for the present work, as well as stating the objectives of the thesis.

## 1.1 Research Area

Our society is becoming increasingly dependent on the rapid access and processing of information. More information is being stored and processed on computers. The fast expansion of inexpensive computers and computer networks has increased the problem of unauthorized access and tampering with data. An increased connectivity not only provides access to larger and more varied resources of data more quickly than ever before, it also provides an access path to the data from virtually anywhere on the network (Power, 1998). Thus, the systems should be more resistant against misuse activities. These activities may be successful due to many reasons, for example, hardware or software failures, incorrect system administration, etc. Software bugs represent a great danger, since software designers do not learn from past mistakes, they still reproduce "classical" programming mistakes (such as buffer overflow in sendmail (Sendmail Mail Program, 2000)). In many cases, the security controls themselves introduce weaknesses and protocols developed for secure communications tend to provide unintended services (Focardi and Gorrieri, 2000). For example, it has recently been discovered that SSH protocol (Ylönen *et al.*, 2002) designed to provide secure communications between two hosts provide such unintended services. Despite encryption and authentication mechanisms it uses, SSH leaks inter-keystroke timing information, which makes an eavesdropper able to learn the lengths of user passwords and keystrokes that the user is typing during SSH sessions (Song *et al.*, 2001). Lowe estimated that approximately half of the protocols published fail to achieve their goals in some respect (Lowe, 1996).

Lately, the amount of successful intrusion incidents has grown quite high: even 90% of all major companies and government agencies have detected at least

one major intrusion incident during the last twelve months, 78% detected employee internal abuse of access rights and privileges (Power, 2002). Nowadays the Internet is becoming more and more popular constantly creating new applications that at the same time brings about new possibilities with which to abuse it. Recently it has become a place for electronic terrorism or vandalism. It is also used as one of the primary means that people use to try to influence world opinion and as a result we can see cyber-fights between groups of people sharing opposite opinions. As a result, in 2001 an increase of 15 per cent in computer penetrations was observed relatively to 2000 (computerheadline.com, 2002). Thus, computer protection against attacks is a very important problem and it is unlikely to be completely solved in the near future.

In this section we introduce the computer security area and describe the security problems that are currently being considered as the most important ones.

### 1.1.1 Computer Security

According to (Russel and Gangemi, 1991), a computer security infrastructure is based on the following three security services: *confidentiality*, *integrity*, and *availability* in a computer system. Confidentiality requires that the information stored in a computer or transmitted over a network may be accessible only to those authorized for it. Integrity provides a mechanism for protecting information against accidents or malicious tampering. Availability means that the computer system remains working without degradation of access and provides resources for authorized users when they need it. For instance, it must assure protection against denial of service attacks, which consume an abnormally large portion of available resources, denying access to other users. A secure computer system protects its data and resources from unauthorized access, tampering, and denial of service.

There is a fourth security service - *accountability*. It is an incorruptible record of activity on a system that positively identifies the users and actions involved. A high importance has been recently attached to it (Avizienis *et al.*, 2001), (Powell *et al.*, 2001). One reason is that the number and complexity of intrusion detection systems have grown; therefore, these systems require a more detailed and complete audit log. Another reason is a usage of log files as evidence in court; thus, they must not have been tampered with and they must have as full a description of a case as possible. As well as unauthorized users, authorized users sometimes make mistakes, or even commit malicious acts. In these cases, a system administrator needs to determine what has been done, by whom, and what was affected; thus an audit is the only way to achieve these results.

The security services described above provide preventive measures for ensuring the security of the system by helping to avoid security policy violations that can occur. A security policy defines the requirements of acceptable usage of the computer resources and establishes the correct procedures for their us-

age. Policy plays three major roles: makes clear what to protect and why, states the responsibilities for that protection, and defines a ground on which to interpret and solve any conflicts. It also defines security priorities, since different organizations have different security concerns and must set their priorities by establishing the proper policies. For example, in a banking environment, the integrity and accountability are the most critical concerns, while in some military systems that processes classified information, confidentiality may be the first priority, and availability the last.

### 1.1.2 Security Problems

Computer systems have become an essential part of the critical systems that have a crucial importance and, hence, they must have as high a tolerance as possible against misuse. In many cases, such as the Internet worm attack of 1988 (Spafford, 1989) or the Melissa virus (Melissa Virus, 1999), network intruders have easily overcome the intrusion prevention mechanisms, such as authentication and authorization, designed to protect systems. Nowadays, when e-commerce is gaining wide acceptance, it also becomes a popular target for intruders. For instance, computer hacking by organized criminal groups in Russia and the Ukraine have already resulted in more than one million stolen credit card numbers (CNN.com/SCI-TECH, 2002).

According to the ISO standard 10181 (ISO, 1991) the threats for which protection is needed include:

- Masquerading - both users and hosts. Usually, it is solved by various identification and authentication procedures using cryptographic techniques.
- Unauthorized use of resources. It is solved by access control mechanisms and policies.
- Unauthorized disclosure of information. This is usually solved by encryption applied to the data during transfer over the network.
- Tampering with information and/or resources. This is solved by access control as well as usage of cryptographic techniques to assure the integrity of information.
- Repudiation of actions, proving the origin and validity of actions and messages. It is solved by the cryptographic techniques, such as digital signature.
- Denial of service - usually solved by the correct service implementation.
- Auditing and accountability - logging facility. The system must provide enough information that describes internal events for external review to prove a system's accountability.

An intruder, in order to make an intrusion successful, may exploit the threats described above. Basically, there are different kinds of attacks that were defined in the literature (Anderson, 1980), (Mell, 1999), (Kumar, 1995), (Krsul, 1998). They group attacks basing on different characteristics (such as attack's end-effect or methods involved in it). A classification scheme presented earlier by Smaha (1988) is general enough to encompass all the above schemes and it classifies the intrusions by their types into the following six types:

- Attempted break-in: often detected by the abnormal behavior profiles or violations of security policy.
- Masquerade attack: often detected by the abnormal behavior profiles or violations of security policy.
- Penetration of the security control system: usually detected by monitoring the specific patterns of activity.
- Leakage: often detected by atypical usage of I/O resources.
- Denial of Service: often detected by the abnormal usage of the system resources.
- Malicious use: often detected by the abnormal behavior profiles, violations of security policy, or the use of special privileges.

The described in this section types of potential threats and attacks present a great danger to networked computer systems. Therefore, it is necessary to use a tracking mechanism to detect or even prevent current and future attacks originated from outside, as well as insider abuses. In the next section we are going to describe systems and employed by them techniques aimed at preventing or/and detecting threats described in this section.

### 1.1.3 Intellectual Attacks

There is a threat that was not covered in the previous section. It is an *intellectual attack* (or *meta-attack*). It is a distinct challenge presented to intrusion detection and it is one of the most dangerous ways to subvert the anomaly intrusion detection system. An insider or other well-informed user may issue this kind of attack. It uses the main problem of statistical approaches in anomaly detection - a gradual training possibility; i.e. after some malicious actions the anomaly intrusion detection systems may recognize an intrusive behavior as normal in the future. For the anomaly detection system that uses online learning techniques intellectual attacks represent a great danger. It is almost equal to putting a trojan horse into the system, but much more dangerous since it is much more difficult (or almost impossible) to detect.

It is supposed that the intruder has full knowledge of the system's defenses, including the anomaly detection system and its user profiles. Having

such knowledge, the intruder may attempt to adapt his or her behavior, to that recorded in a profile for a valid user, and avoiding notice by the anomaly detection system. On the second step after initially conforming to expected behavior (not necessary for insider) the intruder needs to mask anomalous behaviors. He tries to subvert the detection system by training it gradually, changing to malicious behavior in such a way that it does not appear to be suspicious (force abnormal learning). In other words the intruder hides his malicious behavior as normal changes in behavior on the part of a valid user and the adaptive anomaly detection system accepts this and updates its profiles. After this the intruder has a safe passage to the system, since it will be accepting this kind of malicious behavior as normal in the future. We call this process an *intellectual attack*.

From the first look it seems that the intellectual attack may refer several classes of attacks in a scheme given in Section 1.1.2. This classification provides a grouping of intrusions based on their end effect and the method of carrying out the intrusions. If we conduct further analysis it is possible to note that it is not possible to refer the intellectual attack directly to any of these classes. It is obviously not *denial of service*. It can not refer either to *attempted break-in* nor to masquerade attack since a user can train his own profile. There might be some implicit *leakage* of information, for example, when the user is training the system he/she might get some feedback to his/her actions when the workstation locks or a system administrator during the incident investigation contacts the user. However, leakage means that the system provides unintended services (due to flawed design or bad implementation) and this is not the case here. It is also not a *penetration of the security control system* since in this case it is not necessary to penetrate it in order to subvert. It might be *malicious use* of resources, but the user is not directly abusing anything that he/she has access to. Therefore, we can see that the intellectual attack does not refer to malicious use in this sense. Thus, it is a very complicated and dangerous kind of attack to detect and should be paid enough attention to in the future.

Summarizing this section we can say that in general the introduced problem is one of distinguishing real concept drift (introduced by valid user) from abnormal training and in the following chapters we propose our methods in solving this problem.

## 1.2 Intrusion Detection

Most computer systems provide an authentication mechanism as their first line of defense. However, this only defines access restrictions to an object in the system, but does not restrict how a subject may manipulate with the object itself, if it has the access to it (Denning, 1982). Access control cannot prevent unauthorized information flow through the system because such flow can take place with authorized accesses to the objects. Also the access controls and protection models are not helpful against insider threats or compromise of the authentica-

tion module (Kumar, 1995). If a password is weak and is compromised, access control measures cannot prevent the loss or corruption of information that the compromised user was authorized to access. A dynamic method, such as behavior tracking, is therefore needed to detect and perhaps prevent the breaches in security.

An intrusion is defined in (Heady *et al.*, 1990) as any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource. An earlier study done by Anderson (1980) characterizes the intrusion as a threat and defines it to be the potential possibility of a deliberate unauthorized attempt to access information, manipulate information, or render a system unreliable or unusable. In other words, an intrusion is a violation of the security policy of the system (Kumar, 1995). The definitions above are general enough to encompass all the threats mentioned in the previous section.

Any definition of intrusion is, of necessity, imprecise, as the security policy requirements do not always translate into a well-defined set of actions. Whereas, a policy defines the goals that must be satisfied in a system, detecting the breaches of policy requires knowledge of steps or actions that may result in its violation (Kumar, 1995).

Due to hardware or software failures, an incorrect system administration, or software bugs, computer systems are likely to remain not properly secured in the nearest future. Thus, there must be means to detect security breaches, i.e., identify the intruders and intrusions and where possible collect the imperturbable evidence to establish a criminal case. The intrusion detection system is a tool that is capable of filling this role and it usually forms the last line of defense in the overall protection scheme of a computer system (Allen *et al.*, 1999). The intrusion detection systems use a number of generic methods for monitoring the exploitations of vulnerabilities (Sundaram, 1998). They are useful not only in detecting successful breaches of security, but also in monitoring attempts to breach security, which provides important information for timely countermeasures. Thus, the intrusion detection systems are useful even when strong preventive steps are taken to protect computer systems, placing a high degree of confidence in their security (Kumar and Spafford, 1995).

There are two major categories of intrusion detection systems which exist: misuse intrusion detection and anomaly intrusion detection (Smaha, 1993). The systems of the first category are based on the detection of intrusions that follow well defined patterns of attack, exploiting known systems and applications software vulnerabilities (Kumar, 1995). A misuse intrusion detection system has knowledge about poor or unacceptable behavior, which it directly searches for (Smaha, 1992). It is not possible to implement an online learning (meaning an automatic creation of attacks' signatures) for such systems; hence these kinds of systems are unable to recognize attacks that are not precisely encoded in the system. Moreover, it is extremely difficult to perform proper testing of such systems due to an insufficient amount of information about the real intrusion

cases (Allen *et al.*, 1999) and an impossibility to create all possible variations of attacks.

The intrusion detection systems of the second category are based on the detection of the anomalous behavior or the abnormal use of the computer's resources (Kumar and Spafford, 1994). For example, if a user uses a computer only during working hours and only from his office, a connection established using his/her userid during the night from a remote host is absolutely anomalous and, moreover, may be an intrusion.

The main problem with anomaly intrusion detection is that it is based on the assumption that all intrusive activities are necessarily anomalous, which is not always true. In real life the set of intrusive activities only intersects the set of anomalous activities instead of being exactly the same. Another issue is that an anomaly intrusion detection system may be trained to accept intrusive activities. Also the selection of threshold levels so that the misclassification rate would be minimal is difficult. Anomaly detection systems are also computationally expensive because of the overhead of keeping track of, and possibly updating several system profile metrics.

The basic principles of detecting intrusions by identifying "abnormal" behavior are outlined by Anderson (1980). He presents a threat model that classifies intrusions as external penetrations, internal penetrations, and misfeasance. External penetrations are defined as intrusions that are carried out by the unauthorized computer system users. Internal penetrations are those that are carried out by the authorized users of computer systems who are not authorized for the data that is compromised; and misfeasance is defined as misuse of authorized data and other resources by otherwise authorized users.

This classification, as well as the one described in Section 1.1.2, provides a grouping of intrusions based on their end-effect and the method of carrying them out, but the main techniques for detection of these intrusions are the same: the anomaly detection, and the precise monitoring of well-known attacks in the misuse detection.

After Anderson's work (1980), the intrusion detection area has been extensively developed and many approaches have been suggested, for example (Liepins and Vaccaro, 1989), (Lunt *et al.*, 1989), (Heberlein *et al.*, 1991), (Ilgun *et al.*, 1995). We are going to consider them in more detail in the next chapter.

## 1.3 Requirements to Intrusion Detection Systems

There are certain key points that any intrusion detection system should address, regardless of what model it is based on. The following issues are identified as desirable for an intrusion detection system (based on (Crosbie and Spafford, 1995)):

- It must *run continually*. The system must be reliable enough to be run in the system being observed.

- The system should require a reasonable amount of *computer resources* and employ minimum manual and ad-hoc elements in its architecture.
- It must be *fault tolerant*. It must survive a system's crash and should not have to rebuild its knowledge base at the restart.
- It should be *survivable*. The system should not stop working if some of its components stop working for any reason.
- It should be *flexible* enough. Using different usage patterns - the defense mechanism should adapt easily to these patterns.
- It should be *scalable* to monitor a large number of hosts.
- It should allow *dynamic reconfiguration* without restarting the system.
- It must resist direct *physical subversion*. The system should monitor itself to ensure its integrity and the integrity of users' profiles.
- It should handle *concept drift* (Schlimmer, 1987) - normal changes in user behavior, i.e. distinguish an abnormal behavior from a normal user behavior changing and then opportunely updating the users' profiles (for anomaly detection).
- It should detect *intellectual attacks* (mainly important for systems that use an online learning). A malicious person can train the system to accept abnormal behavior as normal. Knowing how the system works, he tries to pose an abnormal behavior as a normal behavior changing and updating a profile to accept it as normal in the future.
- It should avoid *misclassification errors*.

The above list of desirable characteristics is very demanding and challenges the researchers. The last point raises an issue about the type of errors that can occur in the system. If we consider the traditional security mechanisms as a wall which protects our resources from attackers, there are four kinds of events that an intrusion detection system should react to (Lindqvist, 1999): probing/provocation, circumvention, penetration, and insider. Everything else is considered as misclassification errors. These can be categorized as:

- Not intrusive but detected as an intrusion (false positives). The activity is not intrusive, but because it is anomalous or it partially matches a pattern of intrusion, the system reports it as intrusive. Since the used approaches are not ideal, the intrusive activity in anomaly detection is a subset of anomalous activity; therefore, the intrusion detection system sometimes falsely reports an intrusion. The occurrences of this type of error should be minimized, since it may not be possible to completely

eliminate them, the useful information has to be provided to the operators.

- Intrusive but not detected (false negative). Since an activity is not anomalous or it does not match any of the misuse signatures, the system fails to detect an intrusion and falsely reports its absence. A false negative error occurs when an action proceeds even though it is an intrusion. The false negative errors are more serious than the false positive errors because they give a misleading sense of security.

- Subversion errors. They occur when an intruder modifies the operation of an intrusion detector or profiles (by modifying it physically or training) in a way that the system produces false negative errors.

The requirements described above should be carefully taken into account while developing an intrusion detection system. Additional attention must be paid to the described above categories of the potential errors since they may give a misleading sense of security by missing actual attacks or otherwise require large amounts of additional work to investigate false intrusion cases.

## 1.4 Research Objectives

Both intrusion detection models (anomaly and misuse) have disadvantages that are peculiar to the specific model. In this thesis, we have developed a hybrid model that combines these two models. The anomaly and misuse detection models are combined in a way that, working cooperatively, each model can completely or partially eliminate the shortcomings of another. Thus, at the end, we expect to achieve better parameters of the resulting model, satisfying the common requirements for such systems.

In this thesis we attempt to provide the answer to the question: *Is it possible to detect the computer intrusions using the machine learning algorithms adapted for online user verification?*

A complete statement of the thesis is:

*A person's interaction with a computer consists of different activities that he/she performs in order to achieve his goals. These activities consist of actions. Each action causes a series of events in the operating system. Each user performs similar activities, which are expressed by repeated sets of actions and which differ on a per-user basis. This gives the possibility to differentiate an intruder from a valid user.*

The main objective of this thesis is to build a portable intrusion detection approach, taking the anomaly detection as basis and implementing online machine learning techniques, instead of or together with the statistical ones, for online user verification. We have concentrated on the following goals:

- overcoming the existing disadvantages of anomaly intrusion detection systems (IDS);

- improving the following parameters:
  - intrusion detection probability - the probability that an intrusion will be detected and the probability that any detected case of abnormal activity will be a case of intrusive activity;
  - detection time - time between the beginning of an intrusive activity and its actual detection;
  - guarantee online intruder recognition - real time disclosure of the real source of an intrusive activity;
  - protection against *intellectual* attack (malicious learning) - it is protection against an intrusion the main goal of which is to force the intrusion detection system to learn in a way that it will recognize the intrusive activity as normal in the future;
  - recognition based on incomplete information.

In this thesis, the main research objective is reached by dividing it into the following research directions:

- usage of AI methods and tools for anomaly detection;
- development of special event representation for the AI methods and tools used;
- development of special methods for knowledge discovery based on proposed information representation;
- development of special methods for protection against undesirable or malicious learning (cross validation of knowledge for discovering mutual inconsistencies);
- choose thresholds in classification algorithms to minimize a false positives alarm rate.

Basically, the research, being presented in this thesis, can be divided into two phases. The first phase is the development of an information representation method for intrusion detection. This method includes temporal aspects of knowledge representation of the users' behavior to construct behavioral patterns. The representation uses Allen's temporal interval algebra (Allen, 1983), (Allen and Ferguson, 1994) to describe the temporal relations between events caused by a user. This representation is also useful in a concept drift (Schlimmer, 1987) handling when the set of training samples is reduced by removing old data that is no longer used for classification. In previous works, a classification was based on the sequences of actions where each action was followed by another, which means that they are not suitable to be used with modern operating systems and auditing facilities, where many processes may be running at the same time. In

this work, the information representation method is developed in a way that allows actions to overlap and thus, may be used in GUI environments.

According to one definition given by Garfinkel and Spafford (Garfinkel and Spafford, 1991), a secure computer system can be depended upon to behave as it is expected to. By analogy with this definition, the approach being developed is based on the assumption that the user's behavior includes regularities, which can be detected and coded as a number of patterns. The information derived from these patterns could be used to detect the abnormal behavior and to train the system.

The second phase uses the proposed representation to develop a new approach to the anomaly detection employing machine learning, in order to catch and then recognize the diversity of a certain user's actions. The pattern encoding and matching mechanisms are based on the probabilistic networks approach. It is adapted to use the information representation developed during phase one to catch the temporal behavioral aspects.

Typically, in anomaly detection, computer systems are trying to compare the current events with the expected or predicted events (Liepins and Vaccaro, 1989). So we then transform the decision problem into three scenarios: what, when and how much. *What* is happening - this is the problem of the identification of the different aspects of a user's behavior. A person's interaction with a computer consists of different *activities* that he performs in order to achieve his goals. These activities consist of *actions*. Each action causes a series of *events* in the operating system, which are usually stored in audit trail logs. Each user performs similar activities which are expressed by repeated sets of actions and which differ on a per-user basis. Thus, the answer to the question *"what?"* would be a recognition of a certain user's activity. To recognize a certain activity it is necessary to collect related events and group them into an action. Related actions, in turn, are grouped into activities, which is the answer to the given question. In other words, the first scenario represents the translation of events, which are mostly an operating system/logging facility specific and are not very informative and representative as parts of a user behavior model, to activities that are described in an independent way and they tell what is happening from a user's behavior point of view.

*When?* - in order to answer this question, we try to observe the temporal aspects of a user's behavior (when an activity is happening and how long) by analyzing and tracing it in its temporal context using Allen's temporal algebra (Allen, 1983), (Allen and Ferguson, 1994) to describe relations between the temporal intervals or actions. Addressing the second scenario we discover temporal regularities between the actions. Therefore, having a model of a user's behavior (in profile) it is possible to predict how long a certain action is going to be and when the next action is expected (the first scenario defines what kind of action to expect).

The answer to the question *"how much?"* would help us to determine the possible danger, by identifying objects that were manipulated in an intrusive sequence. This is the third scenario, but in this thesis we are going to concentrate mostly on the first two.

Dealing with these scenarios has made it possible to capture more behavioral aspects, which, in turn, leads to additional flexibility in managing the information stored in the profiles.

## 1.5 Structure of the Thesis and Paper Summary

Chapter 2 depicts related work in intrusion detection. We describe several key approaches to intrusion detection. None of them uses temporal-probabilistic networks directly to represent and detect intrusions.

This thesis includes ten research papers. The first paper *"A Hybrid Model for Intrusion Detection"* (Seleznyov, 2000a) and the second paper - *"HIDSUR: A Hybrid Intrusion Detection System based on Real-time User Recognition"* (Seleznyov and Puuronen, 2000) constitutes Chapter 3. It discusses the problem of building architecture for the intrusion detection systems. Many attempts have been made to build an intrusion detection system that satisfies the common requirements for such a system. The intrusion detection systems that are developed for research or commercial purposes have a number of problems, which are usually inherent to a certain architecture that the system is based on. Sometimes, developers combine two architectures in hybrid systems in order to merge advantages that both these architectures possess. Unfortunately, quite often, they combine shortcomings of both architectures as well, which usually does not significantly increase the overall performance of the resulting system. In this chapter, we propose our architecture for an intrusion detection system based on online learning. It combines the anomaly and misuse detection in hybrid architecture in a way that helps to overcome some of their disadvantages. We use temporal-probabilistic trees as a main representation for the creation and maintenance of user profiles in the system.

The third paper is *"Anomaly Intrusion Detection Systems: Handling Temporal Relations Between Events"* (Seleznyov and Puuronen, 1999). Its topic is discussed in Chapter 4. This chapter discusses a temporal knowledge representation of a users' behavior that is used to construct the behavior patterns. These are used to decide whether a current behavior follows a certain normal pattern or differs from all known users' behavior patterns. The representation uses Allen's temporal interval algebra to describe the temporal relationships between events caused by the user. We also discuss how our representation is used to help in the concept drift when the set of training samples is reduced by removing old data that is no longer used for classification.

Chapter 5 discusses the topic described in paper four *"A Methodology to Detect Anomalies in User Behavior Basing on its Temporal Regularities"* (Seleznyov,

2001). In this chapter we have formulated an anomaly detection problem as one of user behavior classification in terms of incoming multiple discrete sequences. Although, here we focus on user-oriented anomaly detection. Monitoring multiple streams of discrete events, such as GUI events, system call traces, keystrokes, a system learns in order to classify (or recognize) a user according to his/her behavior. By developing our approach we aim to eliminate, as much as possible, manual and ad-hoc elements from the creation and manipulation of the user profiles by introducing online learning. We develop an approach that allows creating and maintaining users' behavior profiles relying not only on sequential event information but taking into account events' lengths and possible relations between them. Information about user "normal" behavior is accumulated in user profile. We define it as a number of predefined classes of actions with accumulated temporal statistics for every class, and matrix of possible relations between classes. Every class contains a number of instances, i.e. a number of patterns that are allowed for this class. In other words, an instance of a certain class contains temporal information that is peculiar for a certain pattern. A relation matrix describing possible relations between classes gives us the possibility not only to check the "normality" of each action in the incoming sequence of events, but also to check whether current relationships between the actions are "normal" for a certain user.

Chapter 6 is based on papers five-seven: *"Temporal-Probabilistic Network Approach for Anomaly Intrusion Detection"* (Seleznyov, Terziyan and Puuronen, 2000), *"Detecting Abnormal Behavior Using Temporal-Probabilistic Networks"* (Seleznyov, Mazhelis and Puuronen, 2000), and *"Learning Temporal Regularities of User Behavior for Anomaly Detection"* (Seleznyov *et al.*, 2001). It continues the discussion from the previous chapter towards building a portable approach that is able to catch and represent a model of user behavior as fully as possible. Here a typical decision problem in anomaly detection is transformed into the following scenarios: *what* event is going to happen in the future and *when*. In this chapter, we develop algorithms that cope with these scenarios. An online prediction usage in our approach gives us the possibility to reveal possible dangerous event sequences while it does not arouse the intruder's suspicions and to estimate the possible damage to the system.

A problem of importance in information systems security is to differentiate normal behavior changes from malicious learning attempts. Being able to recognize these types of behavior efficiently the system is able to minimize misclassification errors caused by inconstancy of the behavior and prevent undesirable or malicious learning of systems that use online learning to detect the presence of an intruder, masquerading as a valid user, or abusive actions of a legitimate user. We discuss this topic in paper eight - *"Temporal-Probabilistic Network Approach for Anomaly Intrusion Detection: Detecting Abnormal Learning"* (Seleznyov, 2000b) and continue the discussion in Chapter 7 where we use an anomaly detection approach based on a kind of probabilistic network presented in the form of a

tree to develop a method of detecting the abnormal learning. We define and use an *information context* in this chapter to achieve a more complete coverage of a user's behavioral aspects. Finally, the usage of a coefficient of reliability punishment mechanism was developed in order to distinguish between malicious learning and natural user behavioral change (concept drift).

The ninth paper - *"Learning Temporal Patterns for Anomaly Intrusion Detection"* (Seleznyov and Mazhelis, 2002) is put as a foundation of Chapter 8. This chapter extends the questions raised in the paper - main problems we were faced with when designing the prototype and our solutions to them. The prototype architecture is presented and important parts of it are discussed in detail.

Chapter 9 is based on paper ten - *"An Anomaly Intrusion Detection System Based on Online User Recognition"* (Seleznyov *et al.*, 2002). They consider temporal patterns of user behavior in detail showing that they are actually present in user behavior and it is possible to use them together with sequential patterns in order to reliably differentiate a legitimate user from an intruder. Chapter 9 describes experimental settings, discusses our choice of parameters, and presents and analyzes performance results of the prototype.

Chapter 10 concludes the thesis. It discusses the limitations of the described studies and outlines future research directions.

Additionally, in Appendix 1 we present an idea described in the paper - *"Using Temporal-Probabilistic Network Approach for Automatic Pattern Generation for Misuse Detection"* (Seleznyov, 2000c). Here we show one possible way to solve the problem of misuse intrusion detection, i.e. that they are not able to recognize future attacks since misuse detectors only search for previously encoded patterns. The approach is a part of our hybrid intrusion detection model. We use anomaly detection based on time-probabilistic tree network for the detection of abnormal sequences in an input stream and then we use grouping and filtering techniques for picking out the key events in these sequences. Finally, the key events are encoded in a pattern for misuse detection.

## 1.6 Summary

Intrusion detection is an important component of the security controls and mechanisms provided in a system. It usually forms the last line of defense against security threats. These mechanisms are intended to detect breaches of policy that cannot be easily detected other than using intrusion detection methods or prevented using an access control. Intrusion detection is usually based on one of two models: the anomaly or the misuse model. Both models make assumptions about the nature of intrusive activity that can be detected.

# 2 RELATED WORK IN INTRUSION DETECTION

A full description of existing intrusion detection systems is beyond the scope of this dissertation. There are at least 92 intrusion detection systems that have been developed to date (Sobirey, 2002). Therefore, the main goal of this chapter is not to describe all specific systems, but to outline the main approaches used in their implementations. None of them use the probabilistic trees directly, to represent and recognize the variations of user behavior.

## 2.1 Introduction

Basically, there are two main approaches to intrusion detection: misuse detection and anomaly detection. Sometimes, their combination is referred to as a separate approach - hybrid detection (Mounji, 1997).

The misuse intrusion detection systems detect intrusions that follow well-defined patterns of attack, exploiting known systems and applications software vulnerabilities. They have knowledge about intrusive or unacceptable behavior, which they directly search for (Smaha, 1992).

The anomaly intrusion detection systems are based on the detection of the anomalous behavior or the abnormal use of the computer's resources (Kumar and Spafford, 1994). The basic principles of detecting intrusions by identifying abnormal behavior are outlined by Anderson (1980).

## 2.2 Misuse Detection

The misuse intrusion detection refers to the detection of intrusions by precisely encoding them into patterns or signatures and seeking them in a current event stream. These signatures specify the features, conditions, and relationships between events that lead to an intrusion. Therefore, a certain signature refers to a certain intrusion and if the signature is found among events, it indicates an intrusion. A partial signature satisfaction may be considered to be an intrusion

attempt. In the following sections, we describe the main approaches to misuse detection.

### 2.2.1 Expert Systems in Intrusion Detection

Expert systems are developed in a way that separates the rule-matching phase from the action phase. Every detection rule in the rulebase represents a particular scenario and detects its occurrence by matching the corresponding signature against a current event stream. Matching is done according to audit trail events. The intrusion detection rules should be made general enough to capture all variations of the same attack.

In Snapp and Smaha (1992) an example of the use of such systems in intrusion detection is described. This system encodes knowledge about attack cases as *if-then* rules in CLIPS (Giarratano, 1992) and asserts the facts corresponding to audit trail events. When all the conditions on the left side of a rule are satisfied, the actions on the right side are performed. The IDES intrusion detection system uses P-BEST a general rule-based expert system (Lunt *et al.*, 1992) (see Figure 2.1 for example of decision engine). Some other systems as, for example, OSIRIS (Baur and Weiss, 1988) use Prolog rules to encode the intrusion signatures.

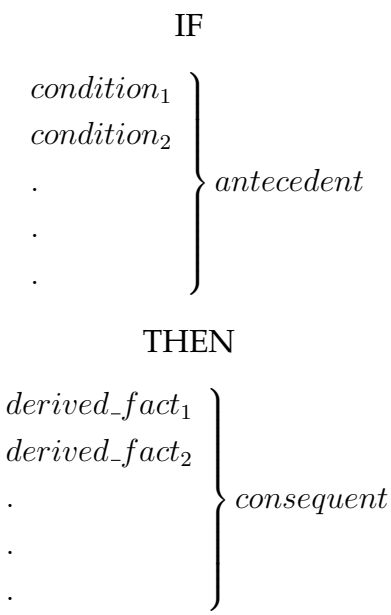


Figure 2.1 A production rule structure for an expert system

There are some practical problems associated with rule-based intrusion detection:

- The expert system has to be formulated by a security professional and thus the system is only as strong as the security personnel who program it (Lunt, 1993).

- It is also very difficult to manipulate rules in rule base, since all actions must take into account the inter-dependencies between different rules (Snapp and Smaha, 1992).

- There is no recognition of the sequential ordering of data, because of the various conditions that make up a rule are not recognized to be ordered (Sundaram, 1998).

### 2.2.2 State Transition Analysis

In the state transition analysis approach, developed in STAT (Eckmann *et al.*, 2001), the monitored system is represented as a state transition diagram. In this diagram, an attack pattern corresponds to the system's state and have Boolean assertions associated with them that must be satisfied to transit to that state and, as the data is analyzed, the system makes transitions from one state to another. Arcs represent the events required for changing a state (Ilgun *et al.*, 1995).

Consider an example (Ilgun *et al.*, 1995). Table 2.1 presents a penetration scenario for 4.2 BSD UNIX operating system (CERT advisories, 1999). The main goal of this exploit is to obtain the root privileges. In this scenario, the attacker exploits a flaw in the *mail* utility, in which the *mail* fails to reset the *setuid* bit of the file to which it appends the message and changes the owner. As a result, the attacker is able to gain the root privileges.

Table 2.1 A penetration scenario

| Step | Command | Comment |
|---|---|---|
| 1. | %cp /bin/csh /usr/spool/mail/root | -assumes no root mail file |
| 2. | %chmod 4755 /usr/spool/mail/root | -makes a *setuid* file |
| 3. | %touch x | -creates an empty file |
| 4. | %mail root < x | -mails root empty file |
| 5. | %/usr/spool/mail/root | -executes a *setuid*-to-*root* shell |
| 6. | root% | |

The above scenario succeeds when the following assertions hold:

1. the attacker must have "write" access to a mail directory;

2. the attacker must have an "execute" access to *cp*, *mail*, *touch*, and *chmod*;

3. a root's mail file must not exist or must be writable;

4. the attacker cannot be *root*.

According to these assertions, it is possible to build a state transitional diagram of the penetration scenario (Figure 2.2).

The considered approach can detect the cooperative attacks and attacks that span across multiple user sessions. However, the attack patterns can only

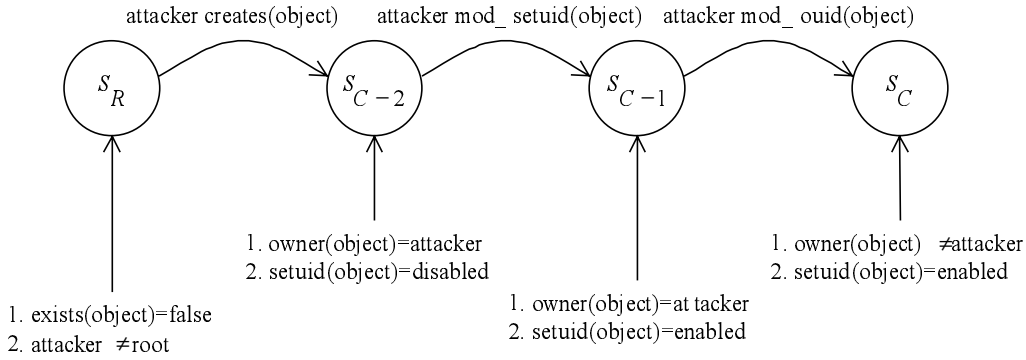


Figure 2.2 A state transition diagram of the penetration example

specify a sequence of events; so more complex ways of specifying events are not permitted. Furthermore, there are no general-purpose methods to prune partial matches of attacks other than through assertion primitives built into the model. And finally, approaches based on the state transition analysis cannot detect denial of service attacks (Sundaram, 1998).

### 2.2.3 Keystroke Monitoring

Keystroke monitoring is a technique that monitors a user's keystrokes for an attack pattern. The attack pattern is a specific keystroke (i.e. sequences of events) that indicates an attack (Lane and Brodley, 1997a), (Lane and Brodley, 1997b).

Unfortunately, there is an unlimited number of ways of expressing the same attack at the keystroke level. Furthermore, the feature of shells (like *bash, ksh,* and *tcsh*), in which a user is able to define aliases, defeats the technique and forces it to use an alias expansion and a semantic analysis of the keystrokes.

Due to this technique analyzes only the keystrokes, unsupervised attacks that are a result of malicious program executions cannot be detected. Also, the operating systems do not offer much support for keystroke capturing; therefore, a facility for the keystroke capturing must be provided additionally, and, finally, in some countries keystroke monitoring is considered illegal, without special permission. For example, in USA due to ambiguities of current laws (they are outdated sometimes) it may be illegal to conduct keystroke monitoring of a user, even if it is only conducted for the purpose of detecting system intruders. Moreover, according to the 18 U.S.C. Section 2512 et seq. (U.S.C., 2002), the design, manufacture, possession, advertisement, and use of the keystroke monitoring software is illegal.

### 2.2.4 Model-Based Intrusion Detection

Originally, this approach was proposed by Garvey and Lunt (1991) and is a variation of misuse intrusion detection that combines the models of misuse with evidential reasoning to support the conclusions about the occurrence of misuse.

This approach assumes that there is a database of attack scenarios, each of which is a sequence of events making up the attack. At any moment, some subset of attacks scenarios are considered as the likely ones by which the system might be under attack. Then the system attempts to verify these scenarios by seeking an audit trail for evidence.

The events in the audit trail are monitored and it is possible to find the intrusion attempts by looking at events that infer a certain intrusion scenario. The model consists of three important modules (Garvey and Lunt, 1991). The *anticipator* predicts the next steps in the scenario by using active models and scenario models (a knowledge base with intrusion scenarios). The *planner* determines how the hypothesized behavior is reflected in the audit data and translates it into a system-dependent audit trail match. It uses the predicted information to determine what to search for next. The interpreter then searches for this data in the audit trail. The system proceeds collecting evidence of the attack until a threshold is crossed; at this point, it alerts the system administrator.

This approach is based on a mathematical theory of reasoning in the presence of uncertainty. It can potentially reduce the substantial amount of noise present in the audit data, since the interpreter and planner know what they are searching for. Also the planner provides an independence of representation from the underlying audit trail representation (Kumar, 1995).

In contrast to the advantages above, this approach creates more difficulties for the person creating the intrusion detection model to assign meaningful and accurate evidence to various parts of the graph representing the model. This evidence must be distinguished and may not be associated with any other normal behavior.

### 2.2.5 Pattern Matching with Colored Petri Nets

Kumar (1995) proposes an intrusion detection approach, which provides a pattern matching-based computational model.

The model of matching consists of:

- *context representation* that allows the matching to correlate various events,
- *semantics* that accommodates the possibility of several intrusion patterns being mixed in the same event stream,
- *specification of actions* that provides execution of specified actions when the pattern is matched.

This model has its own classification hierarchy to categorize the intrusion signatures based on the structural interrelationships among events used to represent the signature. This hierarchy is independent of any underlying matching techniques. Defining the requirements, that patterns in all categories of classifications must meet (specification of context, actions, and invariants of intrusion patterns), the model of matching was devised for the misuse detection.

The model based on a variation of Colored Petri Nets (CP-nets) represents and detects the intrusion patterns. Each signature is represented by a Colored Petri Net, in which a context is modelled as colors of tokens in each state. The matching is done against the audit trail and performed by moving tokens from (possibly several) initial states to the only final state.

Each CP net has several variables assigned to it. The value of each variable can only be assigned once. Each token has its own copy of each variable and it maintains it itself because each token can make its own binding as it flows from state to state.

A CP net has a set of directed arcs that connect states to transitions and vice versa. Each transition is associated with an event type, which must occur in the input stream in order to fire this transition. A single transition may be associated with more than one kind of event. A transition is enabled if each of its input states contains at least one token. Optional expressions (guards) may be placed to each transition, which permits assignment to the token local variables that flow through the transition. It is possible to assign event data fields, expressions, and user-defined functions to the guards. A transition fires if it is enabled and an event of the same type as transition's label that satisfies the guard as the transition occurs. After this a set of consistent tokens is unified, and a copy of this unified token is placed in each output state of the transition.

States of a CP net may be associated with actions performed to each token that comes into the state. This allows countermeasures to be defined when a partial signature match is encountered.

In order to illustrate the use of this approach, Figure 2.3 depicts a graphical representation of the penetration scenario that uses a mail security vulnerability exploitation described in Section 2.2.2. The path in this diagram represents the activity:
> cp /bin/sh /usr/spool/mail/root
> chmod 4755 /usr/spool/mail/root

The node labeled $s4$ represents a final node where an intrusion alarm is made. It must contain at least one token before the alarm will fire.

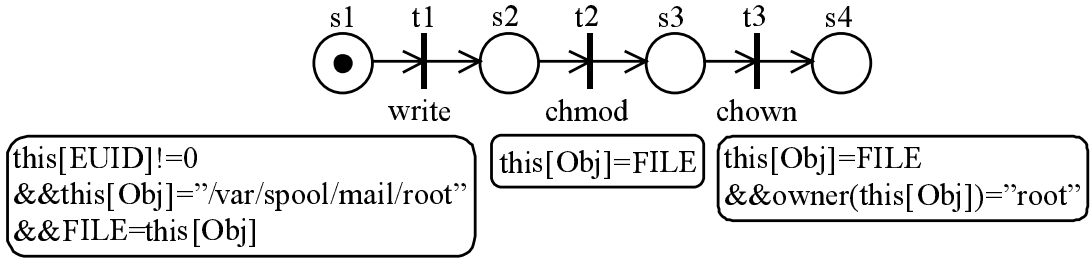


Figure 2.3 A simple pattern of attack represented as Colored Petri Net

An intrusion detection system called IDIOT (Intrusion Detection In Our Time) has been developed based on this approach (Crosbie *et al.*, 1996).

This approach has several problems. It is very difficult to extract and identify the crucial elements of exploitations and translate them into general descriptions to detect exploitation variations. It requires human expertise, since it is very difficult to automate this process. It also requires the consistency of audit information and sometimes, it requires information that may not be provided by the protection mechanism and audit trails, e.g. operating system facilities; therefore, additional auditing mechanisms have to be implemented.

## 2.3 Anomaly Detection

In this section, we make an overview of the techniques that make their decision based on the predicted or expected behavior from the observed behavior. These techniques do not base their decision on the occurrence of specific landmark activities.

### 2.3.1 Statistical Approaches

An anomaly detector based on statistical approaches, observes the activity of subjects and generates profiles for the user and the system as in NIDES (Javitz *et al.*, 1993), (Lunt *et al.*, 1992) or applications as in SAFEGUARD (Anderson *et al.*, 1995). As the system continues running, the anomaly detector periodically generates a value that is a measure of the abnormality of the profile.

A statistical profile is a set of metrics $M_1 ... M_n$, each of them is related to a particular aspect of behavior. It may be CPU usage, I/O usage, file access, typing rate, error rate, etc. Each metric is associated with a threshold beyond which the activity is considered abnormal.

In NIDES (Lunt *et al.*, 1992) a set of $S$ represents the abnormality values of the profile measures $M$ respectively, and a higher value of $S_i$ indicates greater abnormality. Combining the functions of the individual $S_i$ values may be a weighted sum of its squares $a_1 S_1^2 + a_2 S_2^2 + ... + a_n S_n^2$, where $a_i > 0$ and reflect the relative weight of the metric $M_i$.

In general, the measures $\{M\}$ may not be mutually independent, and may require a more complex function for combining them. The anomaly measures are just numbers without a well-defined theoretical basis for combining them (Kumar, 1995).

The advantage of the statistical anomaly intrusion detection is that well-studied techniques in statistics can often be applied, which are able to adaptively learn the behavior of users. However, intruders may gradually train these systems so that intrusive events are considered as normal. Also, it is very difficult to choose measures to monitor and set their thresholds. Finally, relationships between events are missed because of insensitivity of statistical measures, thus a purely statistical intrusion detection system may miss intrusions that are indicated by the sequential interrelationships among events.

### 2.3.2 A Generic Model of Intrusion Detection

Dorothy Denning (1987) proposed a model of intrusion detection independent of the system, type of input, and the specific intrusions to be monitored and it can be viewed as a generalization of most intrusion detection systems built up to date.

The system based on this model looks for audit-recorded events, network packets, or any other observable activity. These events serve as the basis for the detection of abnormality in the system (Denning, 1987).

The central point in this model (Figure 2.4) is the abstract concept of the activity profile, which is the global state of the intrusion detector. This profile is a description of normal activities in terms of statistic metrics: the event counters metric accounts for the number of audit records occurring during a period of time and satisfying some properties; the internal timer metric measures the time separating related audit records; the resource usage metric quantities resources consumed by an event (Mounji, 1997).

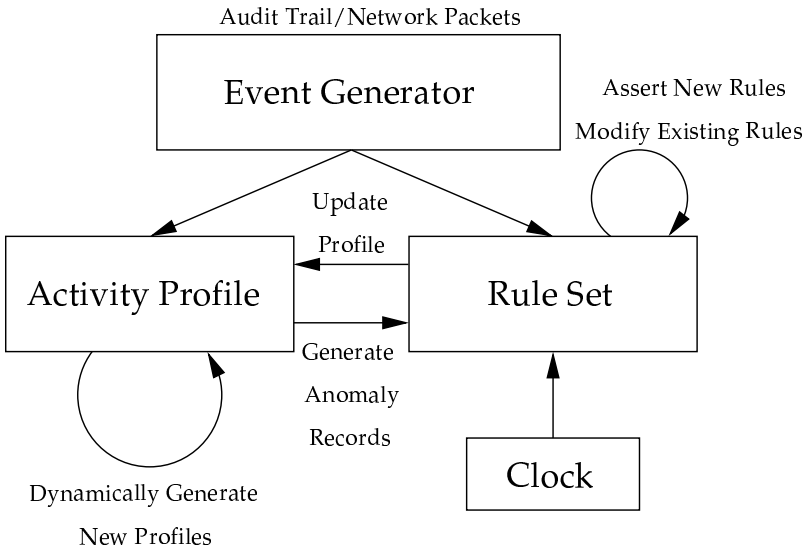


Figure 2.4 A generic intrusion detection model

Each of these metrics may be associated with a statistical model that characterizes the normal distribution of the metric. The activity profile can also generate new profiles dynamically for newly created subjects and objects based on the pattern templates. If new users are added to the system, or new files created, these templates make new profiles for them. Denning (1987) uses a rule-based component to specify an action to be taken when an audit record is anomalous.

### 2.3.3 Data Mining

The main goal of introducing data mining in intrusion detection is to develop an automated approach for building the intrusion detection models. It is aimed

to be applied on a variety of audit data sources, to automatically generate intrusion detection models. Therefore, an intrusion detection is considered, as a data analysis process of finding the normal usage patterns from the audit data (Lee *et al.*, 1998), (Lee *et al.*, 1999a), and (Lee *et al.*, 1999b).

Data mining approach eliminates the need to manually analyze and encode the intrusion patterns, as well as the guesswork in selecting statistical measures for the normal usage profiles. More importantly, the same data mining tools can be applied to the multiple streams of evidence, each from a detection module that specializes on a specific type(s) of intrusion to train the combined detection model that considers all the available evidence.

Data mining generally refers to the process of extracting the descriptive models from the large stores of data (Holsheimer and Siebes, 1994). The recent rapid development in data mining has made available a wide variety of algorithms, drawn from the fields of statistics, pattern recognition, machine learning, and databases. Basically, they may be divided into three categories:

- *Classification*: maps a data item into one of several predefined categories. These algorithms normally output classifiers, for example, in the form of decision trees or rules.

- *Link analysis:* determines relations between fields in the database records. The correlation of system features in audit data, the correlation between command and argument in the shell command history data of a user, can serve as the basis for constructing the normal usage profiles.

- *Sequence analysis*: models sequential patterns. These algorithms can discover what time-based sequence of audit events frequently occur together.

Data mining usage for intrusion detection is a very young, but quite promising topic in the intrusion detection field.

### 2.3.4 Probabilistic Networks

Sometimes, in order to combine anomaly measures, the intrusion detection systems use Bayesian or other belief networks. A Bayesian network (Acid and de Campos, 1996) allows the representation of causal dependencies between random variables in a graphical form and permit the calculation of the joint probability distribution of the random variables, by specifying only a small set of probabilities that relate only to neighboring nodes. This set consists of the prior probabilities of all the root nodes and the conditional probabilities of all non-root nodes (Cheeseman *et al.*, 1991). The Bayesian networks represent causal dependencies between the parent and the child. Usually, each node represents a binary random variable with values representing either its normal or abnormal condition. If the values of some of these variables may be monitored, it is possible to use the Bayesian network calculus to determine $P(Intrusion|Evidence)$.

However, in general, it is not trivial to determine the apriori probability values of the root nodes and the link matrices for each directed arc (Charniak, 1991).

Bayesian classification (Charniak, 1991) is a technique of unsupervised classification of data. Autoclass (Cheeseman and Stutz, 1995) uses the Bayesian statistic techniques to search for classes in the given data and determines the most likely processes that generate the data. It does not partition the given data into classes but finds a probabilistic membership function of each data record in the most likely determined class. It may automatically determine the most probable number of classes of the given data. It does not require any measures, stopping rules, or clustering criteria and continuous and discrete attributes may be freely mixed (Cheeseman and Stutz, 1995).

The Bayesian classification technique permits the determination of the optimal number of classes by grouping users with similar profiles, and thus naturally classifying a set of users. As almost all statistical methods, it also suffers from the same generic failings of statistical systems, such as the difficulty in determining the right anomaly thresholds and the user ability to gradually train the system.

### 2.3.5 Predictive Pattern Generation

A predictive pattern generation-based system (Cheeseman and Stutz, 1995), (Lee *et al.*, 2000), (Teng *et al.*, 1990) uses a dynamic set of rules for detecting intrusions, while expert systems use a static predefined set. These rules are inductively generated based on the sequential relationships and temporal properties of the observed events and they are modified dynamically during the learning phase. Only rules with a high accuracy of prediction (if it is correct most of the time) and a high level of confidence (if it can be successfully applied many times in the observed data) remain in the system.

By revealing regularities in past event sequences, the inductive engine is able to determine that some event types are more likely to occur next in the input stream, than other types of events, and it assigns a probability to each most likely event. Here is an example of an inductively generated rule:

$$E_1, ..., E_k \ :- \ (E_{k+1},\ P(E_{k+1})),\ ...\ ,\ (E_n,\ P(E_n)) \tag{2.1}$$

This means that if the input stream contains the event sequence $E_1, ..., E_k$, the events $E_{k+1}, ..., E_n$ with probabilities $P(E_{k+1}), ..., P(E_n)$ are more likely to be found in the rest of the input stream. An example of a rule generated by TIM (Teng *et al.*, 1990) may be:

$$E_1 \rightarrow E_2 \rightarrow E_3 \Rightarrow (E_4 = 90\%,\ E_5 = 10\%) \tag{2.2}$$

where $E_1 - E_5$ are security events. This shows that for the pattern of observed events $E_1$ followed by $E_2$ followed by $E_3$, the probability of seeing $E_4$ is 90% and that of $E_5$ is 10%.

According to this method, if a sequence of events matches the left side of a rule, then the next event is considered anomalous if it is not one of the predicted events, form the right side. The ability to effectively predict future events depends on training the system on patterns that are the most representative of the normal user behavior, which is a difficult task. A primary weakness of this approach is that the inductively generated rules may not cover all possible normal user behaviors and produce many false alarms.

The strengths claimed for this approach are:

- better handling of a wide variety of user behavior;
- ability to separate a few relevant security events rather than the entire session that has been labeled suspicious;
- better sensitivity to intellectual attacks. Malicious person who attempts to train the system can be discerned more easily because of the semantics built into rules.

### 2.3.6 Neural Networks

The core of this approach is to train the neural net on a sequence of information units (events) (Fox *et al.*, 1990). A neural network consists of many simple processing elements (units) that interact using weighted connections (Kondratoff and Michalski, 1990). The net is trained using the current and past commands. The length of the past commands history is determined by the size of the window of past commands that the neural net takes into account in predicting the next command. Once the neural net is trained on a set of representative command sequences of a user, it encodes the knowledge in its structure, by creating on learning stage connections between the units and assigning weights for them (Gent and Sheppard, 1992).

The intrusion detection approaches that use the neural nets have been developed (Debar *et al.*, 1992), (Fox *et al.*, 1990), and (Gent and Sheppard, 1992) and showed that the neural network-based anomaly detection overcomes the difficulty of devising the statistical features, and consequently, the problem of selecting a good subset of the features is irrelevant. They handle the noisy data quite well and do not depend on statistical assumptions about the nature of the data.

The drawbacks of the neural net approach are:

- they do not provide explanatory information for detected anomalies (Debar *et al.*, 1992),
- the net topology and its weights are determined only after considerable trial and error (Debar *et al.*, 1992),

- the command history size is an independent variable in the neural net design. If it is too low, the net will do poorly, if it is too high, the net will suffer from irrelevant data (Mounji, 1997).

### 2.3.7 Instance-Based Learning

The instance-based model (Aha *et al.*, 1991) classifies data according to its relation to a set encountered exemplar instances. It stores historical examples of user behavior to reference when classifying newly encountered behavioral data.

This model is represented by a set of instances that exemplify the concept. If a previously unseen case is encountered it is classified according to its relationships to the stored instances. A common scheme to define the relations is $k$-nearest-neighbour classification, in which a new case is classified as the majority of the $k$ stored instances closest to it. In continuous domains, the Euclidian distance is a simplest similarity measure.

To apply the instance-based learning on the anomaly detection it is necessary to define a fixed-length vector representation of the data and the concept of distance in this domain. Cases of the valid user's normal behavior serve as labeled instances of multiple classes.

Significant work in this area was done by Lane and Brodley (1997a , 1997b , and 1999). They proposed an anomaly detection system approach based on the instance-based learning. Figure 2.5 shows schematically the system information flow.

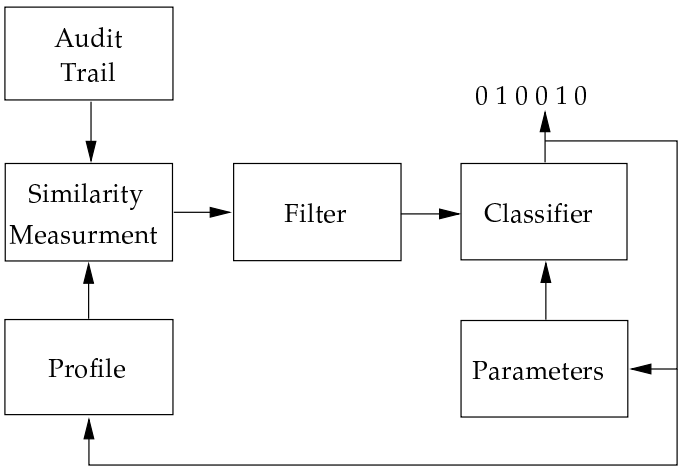


Figure 2.5 Information flow inside the instance-based anomaly detection system

They use a shell command log as a source of user behavior information. This information goes through a parser where it is converted into an internal system format. Then it is compared with a user profile (stored instances) in the similarity measurement part that produces the instantaneous similarity measure stream. This stream is extremely noisy and the classification on its basis is very difficult. To solve this problem a filter is introduced and it serves as a

noise suppressor. After the similarity stream is smoothed it comes into the classifier where the classification takes place. Finally, the system produces a binary stream, which is the detector's estimation of the current state of the input data (”1” - normal, ”0” - abnormal). Feedback is added to the scheme (Figure 2.5) to update profiles and thresholds when user behavior changes with time.

As can be seen from above that the authors presented a framework for instance-based anomaly detection. They claim that it has a relatively short time-to-alarm. However, other advantages and disadvantages hardly depend on which method will be chosen for classification. Also this framework was built using sequential data from a command shell, therefore it cannot be used, as it is, for modern workstations. Users use GUIs for their work. It means that they run numerous applications at the same time, they may have background applications and they may switch between those applications. The events will not be sequential any more, they overlap each other. Thus, the instance-based model, described above, cannot be used as it is.

## 2.4 Limitations of the Existing Approaches

The intrusion detection approaches that have been developed to date basically may be divided into two types: misuse and anomaly. In this section, we consider some limitations of these approaches that most intrusion detection systems are based on; we describe them according to their affiliation to a certain type.

The main assumption of the misuse intrusion detection is that there are attacks that can be precisely encoded in a manner that captures variations of the activities that exploit the same vulnerability. The primary limitation of this approach is that it only looks for known weaknesses, and may not be of much use in detecting unknown future intrusions. It is difficult for such a system to learn (Kumar, 1995); hence these kinds of systems are unable to recognize attacks that are not precisely encoded in the system. Misuse detection is harder to automate since it requires the application of many rules (as in NIDES (Lunt *et al.*, 1989)) or searching for many patterns (as in Shieh and Gligor (1991) or Kumar and Spafford (1994)). It is extremely difficult to implement automatic pattern learning; therefore an intrusion pattern database requires constant manual updating. Moreover, it is almost impossible to perform a proper testing of such systems due to an insufficient amount of information about the real intrusion cases.

Misuse detection depends on what is being audited. Current auditing mechanisms do not reveal the input or output data of a program. They catch only system calls that the application makes. This means that often user-level calls to *read* and *write* functions do not always appear in a one-to-one correspondence in the audit trail because of buffered I/O. Furthermore, the passive methods of security breaches like wire-tapping cannot be detected directly because they do not produce a detectable signature (Kumar and Spafford, 1994).

This approach also assumes the integrity of the event data. Thus, those attacks that involve spoofing, which produce the same events but from a different source, cannot be reliably detected.

The main assumption of anomaly intrusion detection is that intrusive activity is a subset of anomalous activity. Ideally, the set of anomalous activities is the same as the set of intrusive activities. Then, flagging all anomalous activities exactly flags all intrusive activities, resulting in no false positives or false negatives. However, the existing approaches are not ideal, therefore, an anomalous activity does not always correspond to an intrusion. There are four possible activities:

1. Intrusive but not anomalous (false negative). Since the activity is not anomalous, a system fails to detect an intrusion and falsely reports its absence. This happens mostly because of an imperfection of the used approaches. Thus, new approaches should be developed or used ones should be improved.

2. Not intrusive but anomalous (false positives). The activity is not intrusive, but because it is anomalous, the system reports it as intrusive. Since used approaches are not ideal, the intrusive activity is a subset of anomalous activity; therefore, the intrusion detection system sometimes falsely reports an intrusion. This means that a used approach does not fully cover a possible set of normal activity. To correct this, new approaches should be developed, which catch and encode more diverse features of systems' or users' normal activities.

3. Not intrusive as well as not anomalous (true negatives). The activity is not intrusive and is not reported as an intrusion.

4. Intrusive and anomalous (true positives). The activity is intrusive and is reported as such because it is also anomalous.

Sometimes, the anomaly detection tends to be computationally expensive (relatively to misuse detection) because several metrics are often maintained that need to be updated against every system activity. Finally, the anomaly intrusion detection systems may be gradually trained to recognize intrusive behavior as normal in the future.

The intrusion detection systems (misuse and anomaly) so far have been written for the single environments and have proved difficult to use in other environments that may have similar policies and concerns. The existing audit trail formats are dependent on machine architectures, which, in turn, requires preprocessing in order to ensure the independence of the intrusion detection system with respect to the underlying layout of audit records. However, preprocessing of the audit data is only a partial solution since the audit data sometimes contains numerous kinds of events, which are not being used for classifications

and must be filtered. Such preprocessing (translation to the unified format) and filtering require additional computational power that entails some performance penalty, which may be critical for real-time intrusion detection systems.

There is even a bigger problem since it is not always possible to directly map all elements in the model of normal or misuse behavior to audit data fields in the audit data. In this case the audit trail does not provide enough (or detailed enough) information that is required by the detector to function properly.

Despite of how much research has been done the anomaly and misuse approaches still exhibit weaknesses that are primarily related to the lack of a complete model of normal and misuse behavior respectively. The completeness for anomaly detection means that it actually covers all possible instances of normal behavior. Similarly, for misuse detection it is necessary to build a complete model of misbehavior.

As it is possible to see from what is said in this section, most shortcomings of the misuse intrusion detection may not be overcome by improving the misuse detection methods. These limitations are introduced by a misuse detection definition. Additionally, a model of intrusive behavior is directly related to a particular security policy of the organization. As the security policy may change over time or differ significantly from one organization (or site) to another a model of misuse behavior must reflect these changes.

Contrary to the misuse intrusion detection, an anomaly detection lacks precise methods, which distinguish normal from abnormal activities with high probability, and are able to build a complete model of user behavior. The reason is that commonly used methods do not take into account all aspects of user behavior.

## 2.5 Summary of Reviewed Intrusion Detection Methods

In this section we present a summary of reviewed before intrusion detection approaches. We summarize their advantages and disadvantages in terms of limitations of the intrusion detection approaches discussed in the previous section. We would like to note that our classification is based on the approaches described in this chapter and their implementations[1]. However, it is possible that there are different implementations of the same approach which may be found with slightly different characteristics. These differences in characteristics may be a result of a combination of different methods in an intrusion detection system's implementation.

We chose five different characteristics upon which we are going to build our summary. The summary in presented in Table 2.2.

[1] The discussed implementations do not combine different detection techniques. Each of them is based on a single intrusion detection approach, which gives the possibility to evaluate the approach itself.

Table 2.2 A summary of characteristics of intrusion detection approaches

| Intrusion detection approach | Easy to program | Easy to manage | Online model update | Completeness of the model | Source independence |
|---|---|---|---|---|---|
| Expert Systems | - | - | - | + | - |
| State Transition Analysis | - | - | - | - | + |
| Keystrokes Monitoring | - | - | - | + | - |
| Model-based | - | - | - | + | + |
| Colored Petri Nets | - | - | - | - | + |
| Statistical | + | + | + | - | - |
| Generic Model | + | + | - | + | + |
| Data Mining | + | - | - | + | + |
| Probabilistic | - | + | + | + | + |
| Predictive Patterns Generation | - | + | + | - | - |
| Neural Networks | + | - | - | + | - |
| Instance-based Learning | + | + | - | + | - |

"+" means that the characteristic is specific for this class of classification techniques;
"-" means that the feature is not present in this class of intrusion detection approaches.

*Easy to program*. By this we imply how easily a model for normal (misuse) behavior may be built for each of the intrusion detection approaches. In other words, is it possible to automatically create (without participation of experts) profiles required by a classifier for a classification with acceptable accuracy.

*Easy to manage*. This means whether it is easy to manage created models when they must be updated. For example, for some systems it is extremely difficult to remove patterns from the profile since they might contain interdependencies, which are difficult to track, and in some systems (for example, based on statistical anomaly detection) it is easy to manage the metrics, stored in a profile, by changing their statistical parameters.

*Online model update*. By this we imply a system's ability to efficiently distinguish normal behavior changes from abnormal learning attempts, and as a result automatically update a model accordingly to the changes in user behavior or take some preventive steps in the case of the system training attempts.

*Completeness of the model*. To classify according to this characteristic we take published performance results of approaches' implementations and analyze whether there were reported problems with descriptions of different sce-

narios: was a model able to represent all cases/patterns of attack or normal behavior.

*Source independence*. This characteristic shows whether an approach is able to handle data produced by different operating systems or logging facilities.

In Table 2.2. we use ”+” and ”-” to identify whether an approach has a certain feature or not. If there is a ”+” in a column it means that a feature represented by the column most likely refers to the advantages of the approach; if there is a ”-” then lacking of this characteristic may be considered as a disadvantage.

The misuse detection approaches are concentrated in the upper half of the table and the anomaly detection approaches in the lower half. As it possible to see from the table that all misuse detection approaches mostly have difficulties with creation, management, and updating models, whereas for anomaly detection approaches ”+”s and ”-”s are more or less mixed. This observation provides additional support to our claims in the summary of limitations for anomaly and misuse intrusion detection systems provided in the previous section.

# 3 ARCHITECTURE FOR AN INTRUSION DETECTION SYSTEM

Here we introduce our architecture of the Hybrid Intrusion Detection System based on Real-time User Recognition (HIDSUR). We outline its main components and show the interactions between them. Later, in the following chapters, we will concentrate on some particular parts of this architecture describing them in detail.

## 3.1 System Architecture

The HIDSUR architecture consists of three main components grouped according to their primary functions: a group of host agents, a control center, and a detection server. Figure 3.1 depicts the information flow between them.

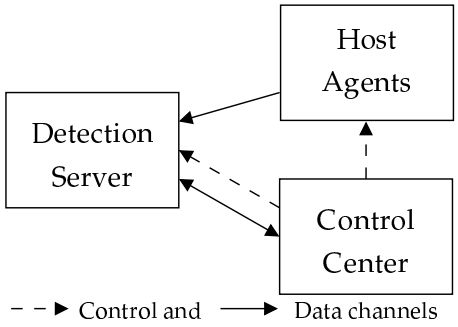


Figure 3.1 Information flow inside HIDSUR

The *control center* is a special part of the system centralizing all control functions. It is responsible for the manipulation of *host agents*, checking the system, i.e. providing services for identification, integrity verification, and execution control. Finally, it makes decisions about tasks distribution over the network. The *detection server* contains all previously learned knowledge and the system detection code. It does not provide any network services except making decisions

about the nature of incoming events. The *host agents* is an auditing facility of the HIDSUR. They are special agents that remain resident in each of the target network hosts and each of them has the responsibility for local gathering and initial information processing in one host. It also has the ability to control the workstation in order to stop an intrusion in progress. There are possibly two basic ways of a response to an intrusion: to lock the workstation, if an intruder has physical access to the workstation, and abort the connection if it is a remote intrusion. However, in the case of remote intrusion, it is more interesting to forward the connection to a "honeypot" (Spitzner, 2000) and perform some online investigation before exposing the detection to the intruder.

Figure 3.2 includes a more detailed description of the HIDSUR structure. In addition to the different structural parts Figure 3.2 shows the main parts of the above discussed two approaches: anomaly and misuse detection, which in the HIDSUR work together. Below we describe the components of the structure, using this logical division into anomaly and misuse subparts of the system.

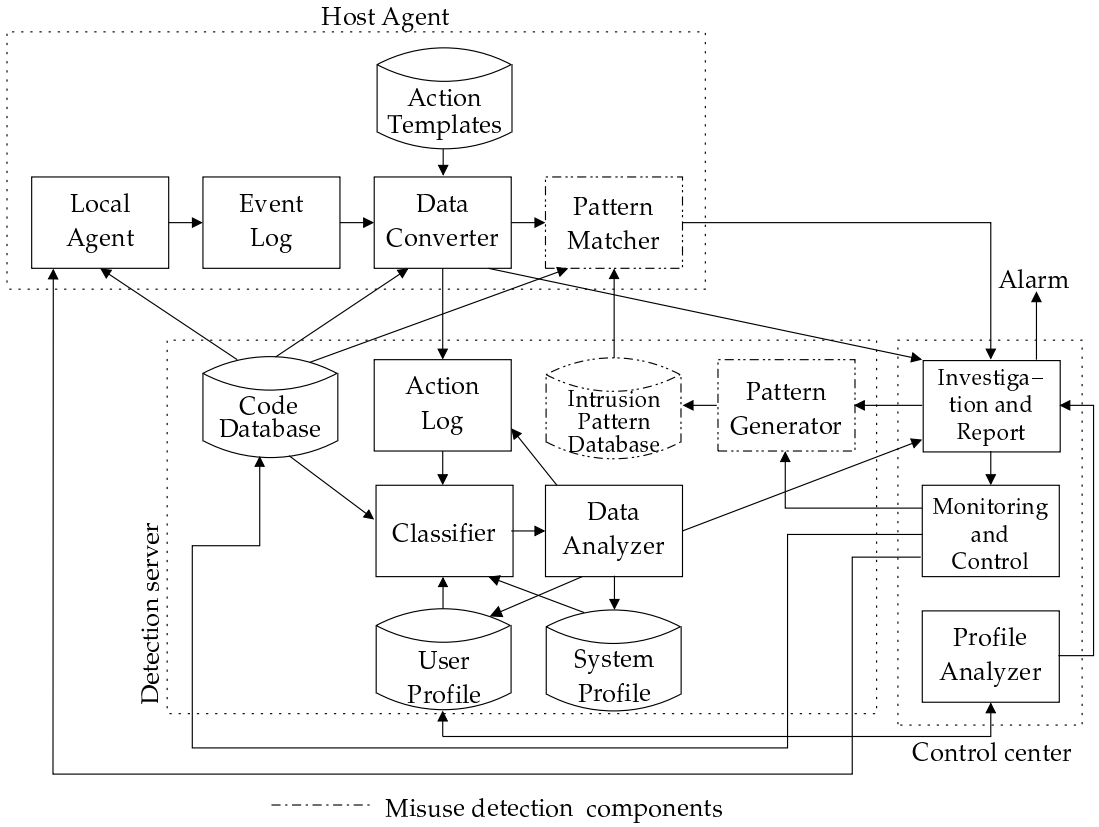


Figure 3.2 HIDSUR

In the following sections we are going to discuss all components of the HIDSUR architecture (Figure 3.2) grouping them by their primary function.

### 3.1.1 Auditing Facility

Here we discuss the elements of the HIDSUR architecture that have the primary function of collecting and initially preprocessing information about user behavior. This is the main functionality of the host agent. It includes: local agent, event log, data converter, event log, pattern matcher and action templates database. All host agent software is run with local daemon privileges. It means that it can not be accessed or tampered with by anyone, unless the workstation is completely compromised (up to root level). The system detects (or at least is supposed to) an intrusion if an intruder gains privileges gradually - starting as a normal user and continuing to upper level of privileges. However, if an intruder logs in directly into root account the system can not notice this because it does not maintain profiles for roots[1].

*Local agent*

The local agent is a loadable kernel module that is permanently resident in the the operating system's memory. It monitors communications between operating system and running applications. It also intercepts an internal kernel message exchange (between kernel's modules). The captured information includes shell commands, GUI events, and system calls. After this the local agent has to collect additional auxiliary information that is required for further processing (but not directly included in event description), such as parent/child process id, etc.

*Event log*

Initially, the host agent stores the information locally into the *event log*, where it waits for further processing. The event stream stored in the event log covers a time interval, length of which depends on the user's activity rate: a more active user creates more events.

*Action templates database*

This is a database containing action templates. By templates we imply "skeleton" for each possible action defining a required and possible auxiliary information. This information is needed later on to convert a stream of events into a stream of actions. It consists of the action's starting and finishing events, required or allowed events between them, allowed relations, etc.

The action template database is created only once, and after this it is necessary to update it whenever a new action(s) is taken for monitoring. We would like to note that the nature of the updates is perfunctory, thus updating the database sometimes would not require much labor. This database is the only element in the HIDSUR architecture that is operating system dependent. It is

[1] It is not possible to create a behavioral model for such users. The first reason is that no one is supposed to do normal work under this account. Second - on the highest level of privileges it is possible to have only one workstation/domain/network managing account, therefore several system administrators usually share the root account.

introduced into the architecture to avoid the necessity of the creation of a separate profile for each operation system that a user normally uses. The database contains descriptions of actions for each operating system where monitoring is performed. When the host agent initializes it determines the type of the operating system and loads locally action templates for this particular operating system, which minimizes the usage of the workstation's memory (RAM).

*Data converter*

It is the data converter part of the host agent, which takes care of forming the actions from the stream of events. It translates the information about user behavior from the operating system's dependent events to an independent description units (actions). It follows the event stream and removes events forming an action when there exists a substream of events that fulfills the requirements of the action. In Chapter 4 we define the structure of information and there we consider the process of information transformation from events to actions in detail.

The data converter of the host agent saves the actions that satisfy defined conditions into the action log of the detection server. In other words, scanning the user event stream it creates a new information *layer (action)* and substitutes the original *event layer* in the stream by highest - *activity*, which is stored in the action log. Thus, this part of information generalization is done locally and this helps to keep the action profile and action log databases of the detection server smaller. It also makes the amount of data transferred between the host agents and detection server smaller. In a way this forms a two-layered structure where the data converter takes events of the original *event layer* and forms the actions of the *action layer*.

The *data converter* also assigns a security significance coefficient to every action instance of the action profile database. These coefficients are initially assigned by a system administrator and they define an intrusion detection system sensitivity to usage of certain actions. They are not the main points for decision-making; however, they are helpful in managing user profiles. For example, if a new dangerous vulnerability is discovered, the system administrator may increase security significance coefficients for some actions that may be used for the exploitation of this vulnerability. This makes the intrusion detection system pay more attention to the usage of these actions. After all workstations' operating systems in the network are patched with new software, the coefficients may be returned to normal.

The data converter keeps event information in a queue until it is able to use it as part of an action, or when a predefined time has elapsed. The value of this time trigger is included in the template of each action and taken from the correspondent database of the detection server, and when it fires (it means that information inconsistency is detected) a request is sent to the *investigation and report component* (most denial-of-service attacks produce inconsistency into the log file).

*Action log*

Actions come from the data converter in the order it is able to form them, i.e. in an arbitrary order, thus, there are possible delays. The main objective of the action log database is to store the actions and keep them in a time order.

### 3.1.2 Anomaly Detector

The *anomaly detector* is based on online learning in order to catch, encode, and then update regularities of user behavior in a knowledge base. It includes system and user profiles, classifier, and data and profile analyzers.

*User profiles*

These profiles contain a model of normal behavior. The model is divided between the user profile and system profile databases depending on the context of action information. The user profile database includes information that is based on previous behavior of the user. This information is presented as patterns of user behavior. They describe what a certain user typically does with his account/workstation.

*System profiles*

The system profile database contains information, which is based on the system's point of view, i.e. what are the notions of the system about the earlier behavior of the user. In other words, the user profile database contains user level behavior patterns and the system profile database system level patterns, correspondingly. A system profile is created relatively to a certain user and it shows the system's usual response to his actions. It may be error messages, amount of the system's resources allocated to a process requested by the user, etc. In other words this profile shows what system resources the user normally requests and how he uses them.

The system and user profiles are not independent. They work cooperatively. They may be considered as high and low level control. In order to minimize the false alarm rate, the system does not report the first abnormal event it encounters. A sequence of abnormal events has to happen in order for it to be reported. It regulates by the coefficient of reliability change and the length of the sequence is determined by how big the deviation is between profiled behavior and this sequence. If this sequence of events is distributed over multiple sessions, it causes very small deviations each time and it may not be detected. Also we use a "closed system" assumption, i.e. the system has examples of "good" behavior on the learning stage. What if there is also bad behavior? It means that it is not going to be detected later. These are two problems that the system profile is aimed to solve. Controlling manipulations of resources and objects in an operating system, it is possible to monitor a number of errors and manipulations when accessing security sensitive resources.

*Classifier*

The classifier takes the stream of the actions included in the action log database as an input and classifies the actions based on the knowledge included in the user profile database of the detection server. The classifier uses a temporal-probabilistic tree to describe a certain user behavior (as described in Chapter 6). It also uses information of the system profile database.

The classifier calculates according to the tree search algorithm, the coefficient of reliability for every action (as described in Chapter 6). Thus, the output of the classifier is a stream of actions where each action has assigned some change to the coefficient of reliability, which shows the probability that the action is originated by the user. In other words it shows how well each action fits into the tree or it is a probability that the specific action belongs to a specific user.

*Data analyzer*

To avoid the appearance of possible mistakes discussed in Section 1.3 we introduce the second level of user data analysis in our model. The data analyzer is developed to fill this role by analyzing coefficients of reliability. If it does not find any suspicions, it reports the action as normal, but if the coefficient of reliability after classifications of some set of actions goes relatively small, the data analyzer tries to find out whether the small value is a result of abnormal behavior or is it probable that the behavior of the user is undergoing a normal change (more detailed description of the algorithm can be found in Chapter 6). If the situation is classified as an abnormal behavior, then the data analyzer reports to the *investigation and report component* of the control center. If the situation is classified as a normal behavior change of the user, then the *data analyzer* updates the user and the system profile databases and returns the actions to be reclassified and the action log updated. If a user was caught doing malicious actions, the system may reassign the security significance coefficients to these actions for this user. Therefore, it will react faster if the user tries to repeat them in the future.

*Profile analyzer*

The profile analyzer is intended to expose *intellectual attacks*. It works asynchronously from the other components. When the system has spare CPU time it activates and analyzes every user profile. Making tree traversal, it calculates the coefficient of reliability for every node in the tree and then for the whole tree. When the coefficient is too low or it grows time after time, then the analyzer makes some actions on the tree. It may split more general patterns into several more specific ones, etc. (the detailed description of algorithms may be found in Chapter 7). It may manipulate the tree parameters to supervise tree changes in the future. If an intellectual attack is detected, the analyzer reports to the *investigation and report* component of the control center.

### 3.1.3 Misuse Detector

The *misuse detection component* co-operates with the anomaly detection component and is intended to support anomaly intrusion detection (in the Figure 3.2 the parts that belong to the misuse detection component are drawn with a broken line). Since the anomaly detection needs to wait to check for consistency and to have several levels of information to analyze which may take a significant amount of time before an intrusion is detected (time-to-detection), the misuse detection part is supplemented to make real-time detection.

If the *investigation and report* component of the control center decides that the current anomaly case is not a false alarm, it has an intrusive sequence of actions.

*Pattern generator*

This component is devised for automatic pattern generation. It takes a stream of abnormal events and creates a signature (pattern of abnormality), which can be detected later on in real-time.

*Intrusion pattern database*

All signatures created by the pattern generator are put into the intrusion pattern database. This database may be easily viewed and managed by a system administrator. Additional, already known intrusion patterns may be uploaded into the database and used for fast misuse detection.

*Pattern matcher*

The *pattern matcher* of the host agent implements one of the pattern matching algorithms (for example, Colored Petri Nets (Kumar, 1995)[2]). It takes actions from the data converter and matches them against the misuse patterns of the intrusion pattern database. If it finds a pattern match it reports an attack to the investigation and report component.

### 3.1.4 Control and Report

This part of the HIDSUR architecture is designed to provide all necessary information about a system's functionality and the tools required to manage it for the system administrators.

*Investigation and report*

Since in case of alarm the system has to provide enough explanatory information to a system administrator, the investigation and report component is designed to investigate and report cases of abnormal behavior. It is intended to expose the real sources of intrusions. For example, since more than 70% of intruders are insiders [3] (Power, 2002), it may use profile cross-validation to find an intruder (try

[2] Colored Petri Net usage was described in Section 2.2.5.

[3] FBI identifies "insider threat" as one of the three most dangerous trends since most damage is done by insiders (FBI WWW page). The Computer Security Institute's 1998 Computer

suspicious stream against profiles of other users and observe if the coefficients of reliability are higher for them than for the current user). The second goal of this component is to create a detailed report to the administrator.

*Monitoring and control*

This component is created to give an interface to a system administrator. It provides actual tools to view and manage processes inside of the system. It consists of a number of visualization and management tools that allow a person to view profiles and their transactions, manage profiles, manage system's constants and parameters, investigate and react to reported cases of abnormal behavior, manage intrusion signatures, etc.

*System code database*

All system code that is crucial for the detection process is stored in the code database of the detection server. System components such as local agents, data converters, pattern matchers, and classifiers, etc. use routines from this database. A centralized storage of the code gives certain advantages. It creates the possibility to store the code in read-only memory (such as read-only disk partition) and protect it from possible misuse tampering. Also the process of system code integrity verification becomes easier in this way. Since the code storage is centralized it gives the opportunity to issue certificates of authority for the routines transferred over the network. These certificates are used for verification by the receiver that the obtained code is authentic and integral.

If the system administrator recompiles some system routine, the control center issues a message that the code database was updated and all the other components that use this database become aware of the need to update themselves dynamically. Thus, it allows for a dynamic reconfiguration and system code change without restarting it.

## 3.2 Networked Architecture Components and their Interactions

In this section we discuss the system components location and their operation over the local area network. Figure 3.3 shows one possible task division over a network.

The *control center* that centralizes all the control functions, also makes all decisions about task distribution over the network. The *control center* includes three modules, two of which are operating synchronously: the *investigation and report* and the *monitoring and control* component. These modules are connected to the network through the *network interface*, which provides authentication and encryption between the system components. The *detection server*, which includes all the centralized databases, is also connected to the local area network by the network interface component.

Crime Survey (conducted jointly with the FBI) reported the average cost of an outsider (hacker) penetration at $56,000, while the average insider attack costs a company $2.7 million.

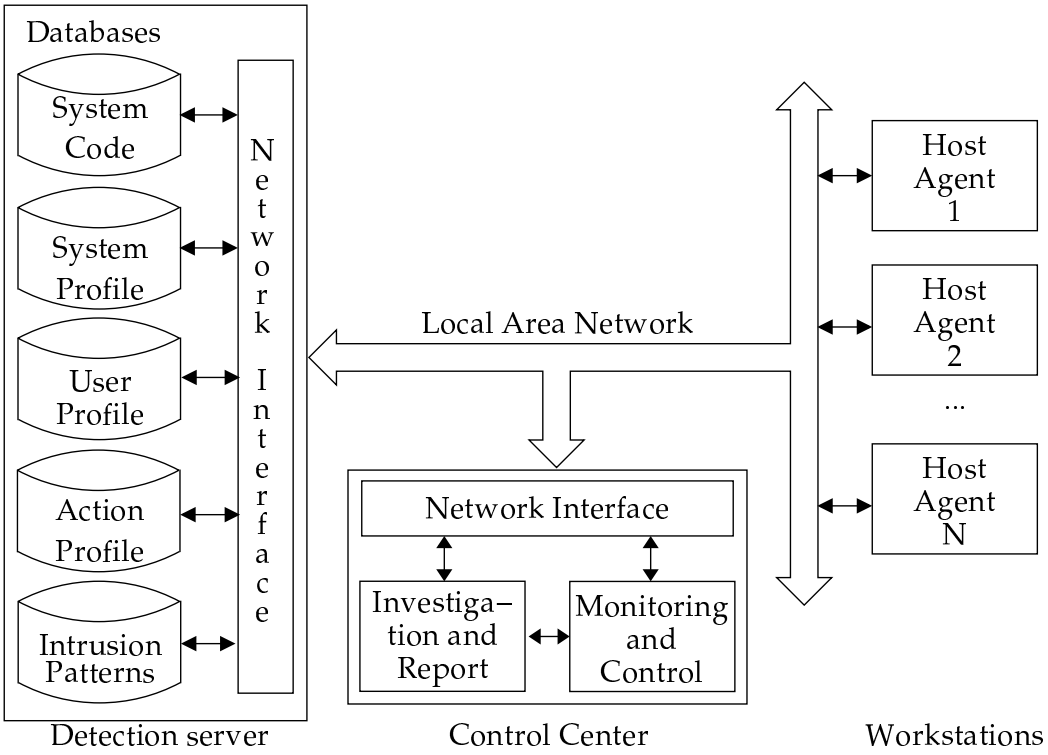


Figure 3.3 Networked HIDSUR components

When an operating system on a network workstation is being started the *initiator* program is being loaded from the detection server (system code database) and executed on a local workstation. It is a very small program aimed to create an environment for downloading the *host agent* for this workstation from the system code database of the detection server. Initially, *the initiator* determines the type of operating system and checks the system's environment. After this it establishes an encrypted connection with the *detection server* through the *network interface* and reports the gathered information. According to the received information the *detection server* uploads the *host agent's* precompiled code for the right type of operating system and reports to the *control center* that a new *host agent* has been created. The *monitoring and control* part of the control center is waiting for the *host agent* to finish its initialization. If confirmation of initialization is not received from the created *host agent*, then the *monitoring and control* component reports the agent's failure for further investigation.

After being uploaded the *host agent* installs itself into the operating system's memory: it creates sensors for gathering the required information, initializes the local OS activity monitor and establishes control over OS software. The latter is required when an intrusion is detected, to enable the *control center* to manipulate the workstation (close connection, lock it, etc.) by sending commands to the *host agent* and thus, separate the intrusion. After successfully completing all initialization tasks the *host agent* establishes a command channel with the *con-*

*trol center* and data channel with the *detection server*. The latter is used for the transfer of user activity information between the agent and the server. Through the former, the *command center* may poll agents to see whether they are alive. If the agent does not respond the *control center* becomes suspicious and initiates an investigation. Also through the command channel the *control center* receives information about the local OS load. Finally, the *host agent* establishes an event cache in case an essential part of HIDSUR is under denial-of-service attack or had some failures and needs some time to recover. The cache does not allow the information to be lost, while the recovering process takes place.

The *detection server* processes information about the user's actions, classifies it and reports the results to the *control center*. Getting information about resource usage, the *control center* makes decisions about tasks distribution over the network. If the *detection server* is busy processing all the incoming information, the *control center* checks the workstations' resource load. Then it issues the command and the corresponded *host agent* begins to perform the initial preprocessing and misuse pattern matching (as shown in Figure 3.2). Patterns and required system routines are provided by the *system code database*, through the network interface. Therefore, one important task of the *control center* is dynamic reallocation of the network's resources.

## 3.3 Summary

In this chapter we presented a hybrid architecture for intrusion detection. The architecture combines the two approaches, the anomaly and misuse intrusion detection, trying to take benefit of their advantages through their cooperation. It also employs the network agents (Zamboni *et al.*, 1998) to distribute detection tasks over the network. The anomaly detector creates a profile to recognize a certain user in the future. The misuse detector offers fast recognition of misuse actions that were exposed in the past.

The presented system introduces features of artificial intelligence (i.e. online learning) in intrusion detection and creates the possibility to benefit from the advantages of the anomaly and misuse intrusion detection. It:

- combines anomaly and misuse detection in a way that anomaly detection provides detection of unknown attacks by deep analysis of user actions and misuse detection minimizes the detection time for known attacks,

- survives denial-of-service attacks by keeping information in local cache in case of connection loss and being able to transmit it over a network later when the connection restored,

- allows task distribution over the local network to avoid overloads that affect the detection speed.

The following chapters we devoted to the different HIDSUR components describing their functionality in more detail, outline different problems and provide solutions. Therefore, below we:

- provide formal definitions and describe the way the HIDSUR manages user information/profiles (Chapter 4),
- explore different methods for user profiling employed in the architecture (Chapter 5 and 6),
- discuss a HIDSUR implementation issues and performance of the prototype (Chapter 8 and 9) and, finally
- show how the HIDSUR employs a misuse detection (Appendix 1).

# 4 TEMPORAL RELATION BETWEEN EVENTS

Anomaly intrusion detection systems are based on the detection of anomalous behavior or the abnormal use of the computer resources (Kumar, 1995). In most works the classification is based on the sequence of events in time (i.e. in which order they follow each other), and time relations between events are not taken into account at all, or only very little attention is given to them (for example in Lane and Brodley (1999b) , Lee *at al.* (1999b) , Lunt *at al.* (1992) , and Teng *at al.* (1990)). Sometimes time relations play a crucial role in attempting to classify events, i.e. determining whether an event is part of anomalous or normal behavior. For instance, if an account has been compromised by an IP spoof attack (Heberlein and Bishop, 1998), it is easier to recognize it relying on the time relations between events, since the misuse activity appears as a continuation of the normal behavior within a single session. Another aspect that has not received enough attention is the natural change of the users' normal behavior, when the user takes new applications in his/her use or when his/her topic of interest changes. Therefore, some of the old data does not accurately reflect the user's behavior any more and should be removed from the training set to handle this *concept drift* (Schlimmer, 1987).

This chapter establishes a foundation for our research. It provides all necessary definitions and describes a way how the user information is handled in the HIDSUR. Here we develop a representation of knowledge about the user behavior as temporal-probabilistic networks, which are later used to analyze the behavior patterns and make decisions with them. This representation uses Allen's temporal interval algebra (Allen, 1983) to describe the knowledge of temporal relations between events. We define the notion of *layer* and *event representation* on each layer in Section 4.1, *distance consistency* between events in Section 4.2, and *coefficient of reliability* in Section 4.3. We discuss how to use our representation in order to help in handling the concept drift and in reducing the set of training samples by removing old data which is not used for classification any more.

## 4.1 Basic Concepts

Traditionally, in anomaly detection, user profiles have been built using different characteristics, such as consumed resources, command count, typing rate, command sequences, etc. In these cases information analysis has been made using system log files, command traces, and audit trails (Lane and Brodley, 1999), (Lee *et al.*, 1999b). In our approach we store and analyze information taking its context as a basis. We present information as temporal intervals and apply Allen's algebra (Allen, 1983) to discover temporal relationships between temporal intervals (Chen, 1988) and to store them for further classification.

The main goal of this chapter is to establish a basement for the further work by providing a universal description for the representation of user behavior and its regularities. There are two main objectives that we are trying to reach by developing the representation. The first one is to create a source (operating system and auditing facility) independent description of a user behavioral model. The second objective is to cover as fully as possible different aspects of user behavior. Here we define all necessary primitives that later will be used for these purposes.

The term *event* is widely used within the temporal database area giving it different meanings. For an operating system's point of view it is a single line in the audit trail; for example it can be a *request for a connection establishment* between the server and the user workstation. On the other hand, from the user point of view it is a much more complicated procedure - we call it *action*. For example - *checking mail*, *text editing*, etc. For the operating system each of these actions includes several events. *Checking mail*, for instance, includes *establishing a connection, user authentication, commands execution, data transfer,* and *closing the connection*.

We define the event *as a single indivisible occurrence on the time axis*. As can be seen from this definition the type of a possible occurrence is not defined. Therefore, we may conclude that the event meaning may depend on the source of the incoming information. In other words, applying the concept of event on different types of source information we are going to have different results. For example, for a GUI log an event may be represented by a GUI message, for network packets it may be a header of a single packet, etc. Applying our definition to the examples we can conclude that in these cases a single record in a log file represents an event.

In order to describe the information abstraction, we define a notion of a layer that reflects different levels of the information generalization. The higher the layer the more general and more descriptive the notions describing the user behavior are. Thus, on the highest layer we describe the user behavior using the most general way, not depending on the source, where the information comes from.

We use an underlying assumption that a person's interaction with a computer consists of different *activities* that he performs in order to achieve his goals. These activities consist of *actions*. An action is a sequence of operations the user performs during the activity. Each action causes a series of *events* in the operating system. Each user performs similar activities which are expressed by repeating sets of actions and which differ on a per-user basis. This gives the possibility to differentiate an intruder from a valid user.

*Layer* is a concept that actually is a generalization level of relationships between occurrences, each of which represents a single event on a different level of generalization. At the lowest *instant* layer, all occurrences are represented as time points (instants) on an underlying time axis. A single occurrence on this layer is called an *event*. It is the equivalent to a single line in the audit trail.

A single instant, relatively to a particular user, describes the event. Information on this layer is source-dependent (for example, it depends on the operating system and/or logging facility used to collect it). Thus, the same occurrences may be defined differently on this layer. We add to this description the name of the event with an integer index that defines the number of the event relative to the user and also a place from which the user caused this event. Also, the last field is reserved for the event's particular information, which differs according to the type of event: it may be the name of the user, the name of the computer, an error code, etc. In other words, it is auxiliary information. Thus the whole description of the event is as follows:

$$Event : << Name >, < Index >, < Place >, < Absolute\ time >, \\ < other\ information >> \tag{4.1}$$

One example can be a mail check using POP3 protocol:

$$<< Popper >, < 2364 >, < computer\_name >, < Jan\ 15\ 16:54:54 >, \\ < user\_name, 0\ 0\ 88\ 90232 >> . \tag{4.2}$$

On the interval or action layer the happenings (i.e. actions) with their relations are described. The action is considered as a temporal interval, as for example: $LOGIN - LOGOUT$.

A *relation* ($Erel$) defines a temporal relation between two events as one of Allen's point temporal relations (Allen, 1983). Considering the following model of time: time is linear, time points can be identified with the rationales under the usual ordering $<$. The difference of any two-time points is likewise a rational number (Kautz and Ladkin, 1991). There are three basic relations that are used to represent relations between events (temporal points): "$<$" $-$ "$less$" relation, "$=$" $-$ "$equal$" relation, and "$>$" $-$ "$greater$" relation.

$$Action : << Event_i >, < Erel >, < Event_j >, < Beginning >, \\ < Length >>, where\ Erel \in \{<, =, >\} \tag{4.3}$$

The most complicated is the *activity* layer, which is represented by actions (temporal intervals or moments) and relations between them, because the actions are extended in time, different actions may overlap in time and interact. One $LOGIN - LOGOUT$ temporal interval, for instance, includes dozens of mail check intervals. A single occurrence on this layer we call an *activity*. Relation between two actions $Arel$ (temporal intervals) is defined as one of Allen's interval temporal relations (Allen, 1983).

$$\begin{array}{lllll} Activity: & << Action_i >, & < Arel_1 > & < Action_j >>, \\ & << Action_k >, & < Arel_2 > & < Action_l >>, \\ & & ... & \\ & << Action_s >, & < Arel_N >, & < Action_t >>. \end{array} \tag{4.4}$$

where: $Arel_n \in \{before,\ after,\ meets,\ met-by,\ during,\ includes,\ overlap,\ overlapped-by,\ starts,\ started-by,\ finishes,\ finished-by,\ equals\}$, and $i = 1, ..., N, N \geq 1$.

According to Allen and Ferguson (1994):

1. given any interval, there exists another interval related to it by each of the thirteen relationships;

2. the relationships are mutually exclusive;

3. the relations have a transitive behavior, e.g. if $A$ is "$before$" $B$, and $B$ "$meets$" $C$ then $A$ is "$before$" $C$.

To visualize the layer structure we provide a simple example user session (Figure 4.1). This example session includes several mail sessions and file editions. On the activity layer (lowest part of the picture) it is possible to see what is happening from the user point of view. He/she has some tasks to perform: to edit some files and check for new mail.

To accomplish these tasks he/she does the following sequence of steps:

- Logs in into the computer.

- Runs mail client program. When initialized, the program establishes the mail session with the server and downloads the user's email messages. After this, the program automatically checks for new mail after a certain time interval and downloads it if new mail has arrived. Mail session length depends on the size of the incoming mail.

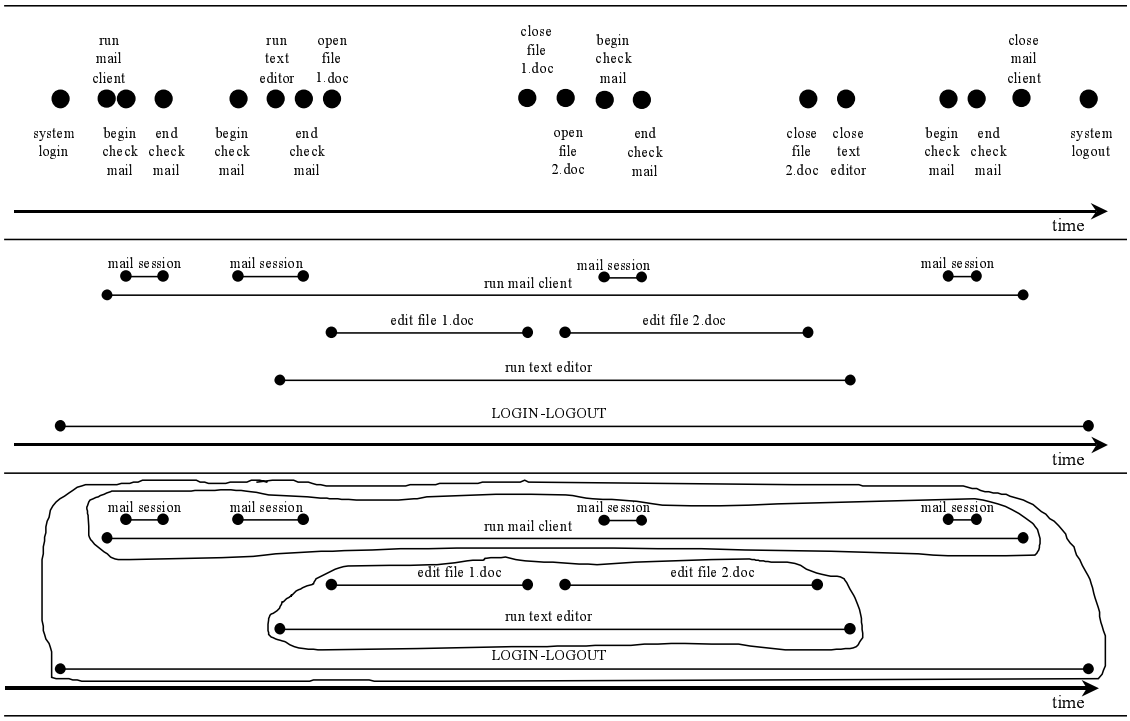


Figure 4.1 Layer structure of a simple user session: the upper layer is the event layer, in the middle is the action layer, and the lower is the activity layer

- Runs text editor and edits two files.

These steps are seen on the action layer (in the middle of Figure 4.1). On the event layer (upper layer of Figure 4.1) we have shown how the operating system sees and reacts to the user actions. All events come as a stream:

- To allow the user to login, the operating system has to spawn several processes (such as login, shell, etc.) and allocate the required resources for these processes.
- To run a mail client, the user requests the operating system to execute a certain program.
- To begin the mail session, the operating system has to accomplish a three-step ”handshake” with the server and send several commands to it.
- To close the mail session, the operating system has to close the connection with server.
- To run a text editor the user requests the operating system to execute some program stored on a disk.
- To open a file the user requests access to an object. To allow this, the operating system has to check user privileges.

- To close the file, the user modifies the object on the disk. The operating system has to check whether he has sufficient privileges to do this.

- To close the program or log out, the operating system closes the user applications and related system processes and frees any unneeded resources.

To demonstrate how to use the presented in this section notations we give an example how a user session, outlined by Figure 4.1, would be described[1]. The event layer is presented below. Each numbered part started by an integer number which corresponds to a single event. For example, the first line describes a login event. Each event contains fields as described at the beginning of this section (Equation 4.1). The first field shows the type of event. The second one is an index number of a sequence. Events that are related to the same sequence are identified by the same index number. After this it is possible to see when the login happened and who was logged in. The rest of the events are described in the same manner. We assume that a POP3 protocol is used for a mail exchange.

$\mathbf{1}, << login >, < 123 >, < workstation\_name >, < Jan\ 15\ 08:05:04 >,$
$< user\_name >>$
$\mathbf{2}, << run\ mail\ client >, <>, < 124 >, < Jan\ 15\ 08:10:32 >, < user\_name >>$
$\mathbf{3}, << pop3\ begin >, < 125 >, < server\_name >, < Jan\ 15\ 08:10:40 >,$
$< user\_name, 0\ 0\ 38\ 90232 >>$
$\mathbf{4}, << pop3\ end >, < 125 >, <>, < Jan\ 15\ 08:11:25 >>$
$\mathbf{5}, << pop3\ begin >, < 126 >, < server\_name >, < Jan\ 15\ 08:20:40 >,$
$< user\_name, 0\ 0\ 26\ 110902 >>$
$\mathbf{6}, << run\ text\ editor >, < 127 >, <>, < Jan\ 15\ 08:21:22 >, < user\_name >>$
$\mathbf{7}, << pop3\ end >, < 126 >, <>, < Jan\ 15\ 08:21:55 >>$
$\mathbf{8}, << open\ file >, < 128 >, <>, < Jan\ 15\ 08:28:46 >,$
$< user\_name, "1.doc" >>$
$\mathbf{9}, << close\ file >, < 128 >, <>, < Jan\ 15\ 09:27:25 >>$
$\mathbf{10}, << open\ file >, < 129 >, <>, < Jan\ 15\ 09:33:28 >,$
$< user\_name, "2.doc" >>$
$\mathbf{11}, << pop3\ begin >, < 130 >, < server\_name >, < Jan\ 15\ 09:36:41 >,$
$< user\_name, 0\ 0\ 3\ 8462 >>$
$\mathbf{12}, << pop3\ end >, < 130 >, <>, < Jan\ 15\ 09:37:03 >>$
$\mathbf{13}, << close\ file >, < 129 >, <>, < Jan\ 15\ 10:09:25 >>$
$\mathbf{14}, << close\ text\ editor >, < 127 >, <>, < Jan\ 15\ 10:09:43 >>$
$\mathbf{15}, << pop3\ begin >, < 131 >, < server\_name >, < Jan\ 15\ 10:36:11 >,$
$< user\_name, 0\ 0\ 2\ 7826 >$
$\mathbf{16}, << pop3\ end >, < 131 >, <>, < Jan\ 15\ 10:36:23 >>$

[1] This is a symbolical example. It is provided to demonstrate the idea presented in this section. Some events, such as user authorization, data transfer, etc. for mail session, are omitted for clarity and compactness purposes.

**17**, $<< close\ mail\ client >, < 124 >, <>, < Jan\ 15\ 10:59:34 >>$
**18**, $<< logout >, < 123 >, <>, < Jan\ 15\ 11:07:23 >>$

After obtaining the events (which belong to the event layer) from a local agent, a data converter of a host agent goes through the log file and constructs temporal intervals. By doing this it generalizes the information about user behavior up to the action layer. Below we continue the example and show the action layer described by the relational notation[2]. We would like to note that the audit log file that describes the event layer contains information related to many users and processes, but the information on the action layer is related to a single user (the system keeps a separate action log for each user), therefore we do not need to keep user identification any more.

Equation 4.3 defines the fields of each action. Each line started by an action name (they are marked with bold font) describes an action. The description contains the starting and finishing events, the relation that identifies which of them is the starting, time when the starting event occurs, and a temporal length between the events.

**login session**, $1 :<< login >, < " < " >, < logout >, < 0 >, < 10939000 >>$
**mail client**, $2 :<< run\ mail\ client >, < " < " >, < close\ mail\ client >,$
$< 328000 >, < 10142000 >>$
**mail check**, $3 :<< pop3\ begin >, < " < " >, < pop3\ end >, < 336000 >,$
$< 55000 >>$
**mail check**, $4 :<< pop3\ begin >, < " < " >, < pop3\ end >, < 936000 >,$
$< 75000 >>$
**text editor**, $5 :<< run\ text\ editor >, < " < " >, < close\ text\ editor >,$
$< 978000 >, < 6503000 >$
**access object**, $6 :<< open\ file >, < " < " >, < close\ file >, < 1422000 >,$
$< 3519000 >>$
**access object**, $7 :<< open\ file >, < " < " >, < close\ file >, < 5304000 >,$
$< 2157000 >>$
**mail check**, $8 :<< pop3\ begin >, < " < " >, < pop3\ end >, < 5497000 >,$
$< 22000 >>$
**mail check**, $9 :<< pop3\ begin >, < " < " >, < pop3\ end >, < 9067000 >,$
$< 12000 >>$

After the action layer is constructed information is independent from the place of its collection and the tool used for this. However, user activities still differ based on different program usage for the same goal. For example, if a user wants to check emails (activity "checking email") he can use the mail client, which downloads them to a user workstation from a mail server using, for example, the

[2] In our prototype, described in Chapter 8, time was measured in milliseconds starting from the first event in the current log file.

POP3 protocol. The mail client may use IMAP protocol, which slightly changes the sequence of actions in the activity. User may also establish a connection to the mail server and read mail without downloading it. In this case the sequence of actions would look completely different. Thus, the third layer is used to connect the used programs with user goals. In other words, it is used for abstraction from the "what is the user actually doing" to the "what goals he/she is trying to achieve". Being able to abstract the information about the user behavior) it is possible to discover patterns in the user behavior by looking at "how the user is normally achieving his/her goals". We are going to concentrate on the pattern discovery and their usage in Chapters 5 and 6.

If we consider our example we can notice (Figure 4.1) that the whole user session consists of three activities. One of them is denoted by $login - logout$ interval, which includes all the other user actions:

**user session**: $<< login\ session, 1 >, < includes >, < mail\ client, 2 >>$
$<< login\ session, 1 >, < includes >, < mail\ check, 3, 4, 8, 9 >>$
$<< login\ session, 1 >, < includes >, < text\ editor, 5 >>$
$<< login\ session, 1 >, < includes >, < access\ object, 6, 7 >>$

Nowadays it often happens that the user uses his computer without logging out for many days. Therefore, the *user session* activity has a number of subactivities, which are separated by inactivity intervals, and they represent actual sessions (for example one per day). Therefore, sometimes it may not be reasonable to consider a user session as a separate activity. If a system is implemented in a way that a session is not strictly denoted by login and logout actions a classifier of such system is able to automatically discover and use patterns in these subactivities. Below there are two other activities from the Figure 4.1.

**cheking email**: $<< mail\ client, 2 >, < includes >, < mail\ check, 3, 4, 8, 9 >>$
$<< mail\ check, 3 >, < before >, < mail\ check, 4 >>$
$<< mail\ check, 4 >, < before >, < mail\ check, 8 >>$
$<< mail\ check, 8 >, < before >, < mail\ check, 9 >>$
**editing text**: $<< text\ editor, 5 >, < includes >, < access\ object, 6, 7 >>$
$<< access\ object, 6 >, < before >, < access\ object, 7 >>$

These three layers are the way by which systems classify certain patterns of change. No one is more correct than the other, although some may be more informative for certain circumstances (Allen and Ferguson, 1994). They are aimed to manage incoming information from multiple sources. For example, if the system detects that a WWW browser is active and it exchanges information using HTTP protocol then it may conclude that the user is browsing WWW pages on the Internet. If the system observes network packet headers it may come to the same conclusion when it detects connection establishment between the user workstation and some server on port 80, followed by an information exchange.

It may come to the same conclusion when observing some sequence of system messages, between different modules of an operating system. Therefore, our point is - by monitoring different sources (sequences of events) it is possible to come to the same conclusions or, in other words, build a string of user activities, which are source and platform independent, and layer structure is aimed to make this transformation from the event to the activity layer, creating the possibility to combine multiple strings from different sources. Moreover, wide usage of I/O caching introduced some uncertainty in determining exact time points for certain events. System *read* and *write* operations may be delayed and appear later in system logs than the user actually performed them. Getting confirmation from different sources the system is able to reason about the actual time the user has requested a certain operation[3].

## 4.2 Consistency of Relations

One group of attacks takes advantage of the ability to forge (or "spoof") an IP address, which is sent along with every IP packet. This makes it possible for a user to pretend that he/she is his neighbor when sending packets to a server. While he/she will not see any transferred data of his neighbor, he is still able to take advantage of the situation. For example, he can pretend to be his neighbor on a command channel and ask the server to open a second channel to his own address. He still does not see any data of the first channel, but the second channel is wide open for him to exploit.

The situation above creates an inconsistency in the log files, which can be checked without expending many resources. There exists at least two different kinds of consistency:

1. *information consistency* which requires that every action has to begin with some event and end with a corresponding one, and

2. *distance consistency* defines the temporal conditions for distances between actions taking into account spatial places (time zones) these actions were issued from. It is used to make sure that, when we consider actions as temporal intervals, there is a temporal distance $t$ for every pair of actions and it has to be consistent with the temporal distance between spatial places these actions were issued from.

Checking for the information consistency is a trivial task and so here we concentrate on the distance consistency. We base the distance consistency evaluation between two actions on the IP addresses (i.e. approximate spatial places) of the source packets of the actions.

We introduce three notions: *temporal distance between actions*, *temporal distance between places*, and *coefficient of mobility*.

[3] Since we are using a user-oriented approach we are interesting in time when a user has requested a certain action not when it actually has been performed by an operating system.

*Relation* $Arel$ is a relation (one of thirteen presented by Equation 4.4) between two actions $Action_i$ and $Action_j$. It is characterized by a temporal distance between these actions. Thus, the relation itself is a qualitative parameter that describes what kind of relation it is, and the temporal distance is a quantitative parameter that shows how strong the relation is or how much of it is possible to find between two current actions.

What is a temporal distance in context of user behavior expressed by a sequence of actions? Below we consider all the possible relations between actions and define a notion of temporal distance for them. There are thirteen possible relationships between two actions (Allen, 1983). In our approach we are not using all of them. Since we have qualitative temporal characteristics we may define several *basic relations*, with which different temporal parameters produce all possible relations.

We define two relations as *basic* ones: ”before” and ”during”. Figure 4.2 shows them in temporal axis.

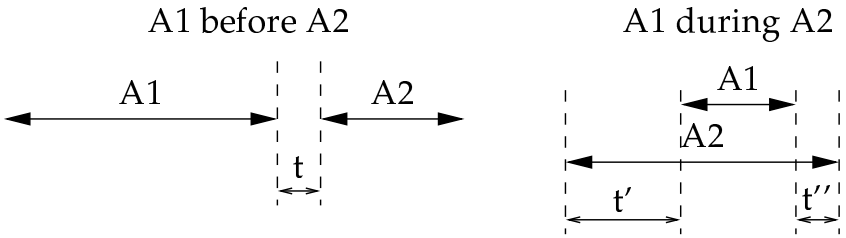


Figure 4.2 Basic relations ”before” and ”during”

Below we define temporal distance $t$ for basic relations. Let there exist two actions $A1$ and $A2$ described as:
$A1 :<< A1_{begin} >, < \text{”} < \text{”} >, < A1_{end} >, < A1_{start} >, < A1_{length} >>$ and
$A2 :<< A2_{begin} >, < \text{”} < \text{”} >, < A2_{end} >, < A2_{start} >, < A2_{length} >>$.
Then the temporal distance for each basic relation is calculated as shown by Equations 4.5 and 4.6.

$$\forall\ A1\ before\ A2,\ t = td(A1\ before\ A2) = A2_{begin} - A1_{end} \tag{4.5}$$

If $td(A1\ before\ A2) = 0$ then we have the case when $A1\ meets\ A2$. If $td(A1\ before\ A2) < 0$ in terms of Allen's temporal relations we may say that $A1\ overlaps\ A2$.

$$\forall\ A1\ during\ A2,\ t = td(A1\ during\ A2) = A2_{begin} - A1_{begin} \tag{4.6}$$

As it is possible to see from Figure 4.2 a position of the interval $A1$ relatively to $A2$ is defined by $t'$ and $t''$. If $t' = 0$ then this basic relation forms $A1\ starts\ A2$ relation. In case when $t'' = 0$ we have $A1\ finishes\ A2$. When both $t' = 0$ and $t'' = 0$ then $A1\ equals\ A2$. For our purposes we do not need to calculate and store $t''$. Since we have a relation $Arel$, $t'$ and $A1$ length we are always able to

reconstruct $t''$ when we need it. That is why we defined the temporal distance for a "during" relation as $t'$.

As we can see from Table 4.1 our two basic relations cover all seven direct Allen's relations[4]. It is always possible to revert from our basic relations to the 13 Allen's relations knowing the temporal lengths of the actions and a basic relationship that connects them. The basic relations were introduced to simplify the representation of relations between the actions and to avoid redundancy in the implementation.

Table 4.1 Correspondence between Allen's and our basic relations

| **Basic relation** | **Value of $t$** | **Allen's relation** |
|---|---|---|
| A1 *before* A2 | $> 0$ | A1 *before* A2 |
| A1 *before* A2 | $= 0$ | A1 *meets* A2 |
| A1 *before* A2 | $< 0$ | A1 *overlaps* A2 |
| A1 *during* A2 | $> 0$ | A1 *finishes* A2 or A1 *includes* A2 * |
| A1 *during* A2 | $= 0$ | A1 *starts* A2 or A1 *equals* A2 * |

* Depends on $t''$, which may be always calculated knowing $t'$ and temporal lengths of the A1 and A2.

*Temporal distance between two places* $td(place_i,\ place_j)$ is the minimal time needed by any user to change place from the first place to the second one. For simplicity we include a description of a user to each profile, a matrix, which defines these minimal times between different *time zones*. In this thesis we define them as: department, university, city, country, and world. The choice of time zone depends on the home network topology and a security policy. The implementation of this mechanism is relatively simple. It is necessary to identify a range of IP addresses for the home network. Then the system will automatically analyze the IP address during every new login. Making the DNS address lookup it will retrieve all available information including the name and address of the ISP or the organization that owns the IP block. According to the address it is possible to identify a time zone of a new login.

The diagonal elements of the matrix (Table 4.2) - $td(time\_zone_i,\ time\_zone_i)$ define the minimum time needed to log from different IP address within the same time zone. This time depends on:

- the local physical network structure which sets constraints on every user of the network and
- the user related characteristics of the IP address change, as for example how the user moves from one IP address to anther one.

[4] Six reverse relations we do not take into account since it is possible to change the order of actions.

Table 4.2 Minimal time required to move between time zones

| | **Time zone 1** | **...** | **Time zone** $n$ |
|---|---|---|---|
| **Time zone 1** | $td(time\_zone_1,\ time\_zone_1)$ | ... | $td(time\_zone_1,\ time\_zone_n)$ |
| ... | ... | ... | ... |
| **Time zone** $n$ | $td(time\_zone_n,\ time\_zone_1)$ | ... | $td(time\_zone_n,\ time\_zone_n)$ |

The *coefficient of mobility* shows the probability that the next session (i.e. login) of a user will occur from a different IP address than the previous one. The profile of each user contains a matrix (Table 4.3) of coefficients $p(time\_zone_i,\ time\_zone_j)$, which are statistically created and constantly updated for every user. The diagonal elements of the coefficient matrix $p(time\_zone_i,\ time\_zone_i)$ define the probabilities of changing IP addresses within the same time zone and the non-diagonal elements $p(time\_zone_i,\ time\_zone_j)$ define the probabilities of changing IP addresses between two time zones.

Table 4.3 Probabilities of moving between time zones

| | **Time zone 1** | **...** | **Time zone** $n$ |
|---|---|---|---|
| **Time zone 1** | $p(time\_zone_1,\ time\_zone_1)$ | ... | $p(time\_zone_1,\ time\_zone_n)$ |
| ... | ... | ... | ... |
| **Time zone** $n$ | $p(time\_zone_n,\ time\_zone_1)$ | ... | $p(time\_zone_n,\ time\_zone_n)$ |

The coefficient mobility matrix is mostly defined by each particular user behavior. Consider, for instance, a simple university example:

- A university researcher participates in conferences around the world and he uses his/her account occasionally from different countries and continents.
- A system engineer does not travel so much around the world, but he maintains the network and the computers of the department staff using his account from many different computers in the department.
- A department secretary maintains a secure student's credit unit database using his computer mostly from one location - his/her office.

Let the *temporal distance* between two events be:

$$td(event_i,\ event_j) = |event_i - event_j| \tag{4.7}$$

The relation between two events $i$ and $j$ is distance consistent if:

$$\forall(event_i \neq event_j),\ \exists td(event_i,\ event_j) \geq td(time\_zone_i, time\_zone_j) \tag{4.8}$$

Let the temporal distance between two actions be:

$$td(action_i,\ action_j)\ =\ |action\_begin_j\ -\ action\_begin_i| \tag{4.9}$$

The relation between two certain kinds of actions $i$ and $j$ is *distance consistent* if:

$$\forall(action_i \neq action_j),\ \exists td(action_i,\ action_j) \geq td(time\_zone_i, time\_zone_j) \tag{4.10}$$

The above distance consistency definition of actions requires that the two actions be of a certain kind. As can be seen in Figure 4.1 the actions are generally active during some temporal intervals, which may overlap. Thus the application of the distance consistency between actions requires considerations of actions and the local arrangements. Some of these situations are more obvious than the other ones. They are introduced by a system administrator to give him additional control over user profiles. For example, if two users are logged in from different places during the same time interval under the same login name, then we can suspect that one of them is an intruder[5]. If we take an example of $login\ session, 1$ activity from the Figure 4.1 and assume that there is another activity $login\ session, 2$, it is possible to write that if we take two different user sessions - $A1:\ login\ session, 1$ and $A2:\ login\ session, 2$:

$$\forall(A1\ before\ A2),\ td(A1, A2) \geq td(time\_zone_{A1}, time\_zone_{A2}) \tag{4.11}$$

This means that between a logout and a next login a certain amount of time has to pass. This time is taken from the Table 4.2 depending on time zones from where the sessions were initiated. This also implies that the sessions must not overlap each other. This was a simple example of a consistency requirement and its usage. Some other requirements in real life may be more complicated. They should be defined during the learning or system operating processes.

To conclude this section we would like to note that in different time zones the accuracy of the spatial location of the IP address identification is different. For example, inside a department a network topology is known and therefore, precise locations of the IP addresses are known. If a user travels to another country from which he logs in, the locations of the IP address may be sometimes identified with possible error in several hundreds kilometers. However, we are not interested in a very precise location. When a new login happens it is necessary to know only from which time zone it comes (is it from the same time zone as a previous login, or if it comes from a new location, in which time zone this location is). This knowledge allows us to introduce statistical measures that show probabilities of movement, and time usually required by the user to move inside each time zone as well as between them. These measures help to cover additional aspects of user behavior in a user profile.

[5] Of course, this conclusion requires that the user never forgets to logout the connection or that there exists some kind of automatic logout system.

## 4.3 Coefficient of Reliability and Concept Drift

One of the main objectives of abnormal behavior detection systems is not only to expose the abnormal behavior of users, but also to sometimes choose between two contradictory cases. For example, if two audit records show that there exist two sessions from different IP addresses at the same time, under the same login name, the task of the system is not only to detect this, but also to determine which login is abnormal (perhaps they both are).

The *coefficient of reliability* $r$ is a value that shows how much current events (that were claimed to be issued by user "X") correspond to the ordinary events of a real user "X". The coefficient of reliability is a real number from the closed interval from 0 to 1 and it shows, at any certain point of time, the probability $r$ that there is really the user "X" who was logged in as "X" up to the current time point. We assume that these coefficients are initialized for each user and then being changed, taking into account the validity of each action. The more doubt a system has about an action's nature the more "punishment" value it assigns to the coefficient and vice versa. The value of a coefficient change for a new action is calculated taking into account the previous value of the coefficient, the user's mobility, and the temporal distances of time zones and actions.

In the following chapters we suggest an implementation of the concept drift detection and a way of updating profiles in real time. We suppose that every action in the training set has a usage frequency value. If an action is often used for a new case classification, then the frequency value is increased and vice versa. When the coefficient of an action in a training set becomes smaller than a certain predefined threshold value then the action is replaced by another action that reflects the user's behavior much better. Thus, the threshold value has a direct influence to the speed of the training set updating.

## 4.4 Incorporation of Layer Structure into Profiling Component

In this section we suggest layer structure implementation for information generalization while creating or using profiles. We defined a notion of layer, related to the level of information abstraction, and the concept of happenings on each layer in order to clarify our usage of the event concept. In Figure 4.3 the structure of the profiling system is shown. It consists of four main blocks, which are briefly described below.

The main purpose of the *information preprocessing block* (IPB) is to prepare information for its later use. It takes different event streams as an input. These streams can be any discrete streams of nominal values, for example, system calls, keystrokes, GUI events, etc. Then an IPB separates the events of the stream on a per-user basis and stores them in separate FIFO queues created by IPB for every active user. A particular queue covers different time intervals, which depends on the user's rate of activity.

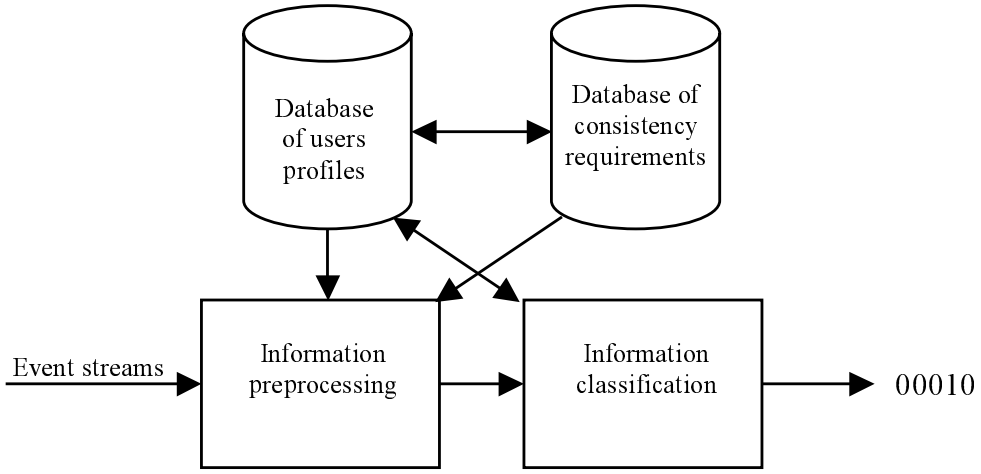


Figure 4.3 A structure of the profiling component

After an IPB has started to receive information about a particular user it begins to scan the user's queue of events. While scanning the queue the IPB creates the information of the higher layers (first -actions, and then, after seeking for temporal relationships between actions, - activities) and substitutes the original queue of the instant layer by highest one - activity. Finally, the IPB takes from the database of consistency requirements a list of the consistency constraints presented using temporal relations. From the database of users profiles IPB takes the values of the parameter, which are different for each user. These values of the parameters are updated while the system learns the users behavior.

The *database of user profiles* contains information about all previous users. Every user has its own profile in the database. A profile contains information about the user behavior, for example matrixes of temporal distances, matrixes of probabilities, and the temporal-probabilistic tree. It may also contain different characteristics of the user, as typing rate, error rate, consuming resources, etc.

The *database of consistency requirements* - part that contains the temporal consistency criteria. All incoming events must satisfy these criteria adapted using the parameters included in the profile of a user. This database also includes the default values of the parameters, which are used, for example, in the case of a new user.

The *information classification* block takes the user profile for the new action classification, in order to determine to which class it belongs: normal or abnormal. A detailed description of this part will be provided in Chapter 6.

## 4.5 Summary

In this chapter we have proposed an information representation method that is used by our intrusion detection system that detects anomalous behavior. Here we provided all formalisms necessary to establish and handle relationships be-

tween actions. Also, the approach makes the information about the user behavior source and operating system independent providing opportunity to process at the same time all information coming from different environments and operating systems. The methods and techniques presented in the following chapters are based on the representation described above.

The approach suggests that in cases where the audit trail information is inconsistent it is possible to expose it using a temporal interval representation applying temporal algebra. Thus, some attacks (such as "spoof" attack (Heberlein and Bishop, 1998), etc.) may be detected without significant effort and, therefore, detection may be performed in real-time. Also, the primary disadvantage of anomaly detection - the gradual training - may be overcome using temporal algebra to determine important relations between events by grouping together actions based on the relations of their endpoints and testing for a trend among them. We are going to discuss this problem in detail and suggest one of the possible solutions later in Chapter 7.

Being able to construct a user's activities from events reported by the operating system gives us an understanding of user behavior, that minimizes the volume of the information being processed, and simplifies the task of differentiating an intruder from a valid user. The simplification reduces the probability of possible mistakes.

# 5 USING RELATIONAL MATRIX TO DETECT ANOMALIES

While performing an anomaly detection a typical intrusion detection system compares current events with expected or predicted ones and decides whether they are normal or anomalous. Thus, the problem of anomaly detection may be transformed into a typical decision problem (Krishnan, 1995) that consists of three scenarios: *what* event is going to happen in the future, *when* it is going to happen, and *how much* danger it may cause.

*What* is happening - this is the problem of identification of different aspects of user behavior. Each user performs similar activities which are expressed by repeated sets of actions and which differ on a per-user basis. Thus, the answer to the question *"what?"* would be a recognition of a certain user's activity.

*"When?"* - in order to answer this question we expose and monitor temporal aspects of the user behavior (when the activity is happening and how long) by analyzing and tracing it in its temporal context using Allen's temporal algebra (Allen and Ferguson, 1994) to describe relationships between temporal intervals or actions.

To answer the question *"how much?"*, would help us to determine the possible danger. The possible loss may be estimated, according to what objects are manipulated in an abnormal sequence of events. However, it is an approximate estimation because it is very difficult to reason about potential losses automatically. For example, a buffer overflow attack may lead to a user or root account compromise. Possible losses may be different in these cases, and it is not feasible to anticipate them without knowledge of this particular attack.

In this chapter we develop algorithms that cope with these scenarios. They are used as a main technique for online user verification in the HIDSUR. They are based on probabilistic networks represented in the form of trees adapted to the problem of anomaly detection. In particular, they are aimed at automatically finding the normal behavior patterns from audit data, encoding and matching them against current event streams in order to find deviations from normal

behavior; and then decide whether it is an intrusion or normal user behavior changes. The information derived from these patterns could be used to detect the abnormal behavior and to train the intrusion detection system.

## 5.1 User Profile

Traditionally, in anomaly detection, user profiles have been built by calculating statistics for different characteristics, such as consumed resources, command count, typing rate, command sequences, etc. (Lane and Brodley, 1999). In our class approach we construct a user profile by defining *classes* or *cases*, which are used as bricks in constructing a model of user behavior, and analyze information inside classes based on its temporal characteristics. We present incoming information as a set of temporal intervals that gives us the possibility to apply Allen's algebra (Allen, 1983) to discover relationships between temporal intervals and to store them for further classification.

In this section we provide necessary definitions and describe the basic concepts of the class approach. Here we consider a problem of user profile building, as bringing to conformity events, provided by the operating system log facilities, with notions used to build a user behavioral model. Thus, the system may possibly have several streams of discrete events such as GUI events, system call traces, network packets, and keystrokes. It needs to automatically build a profile for every user in order to recognize him in the future, fitting current behavior into the behavioral model described in his profile. During this process, the system should use as little as possible manual and ad-hoc elements. In other words, our aim is not only to develop an approach for behavioral model description, but also to automate all processes used for creation and manipulation of the user profiles, as much as possible.

In Chapter 4 we have described a possible structure of information obtained from the operating system. This structure is used for abstracting information included in audit log files to a more general - source or platform independent form, giving it more meaning in a context of person-computer interaction. How to store and process this information? Below we present a way to construct user profiles using information obtained from different sources.

We gave definitions related to the structure of user information going from specific to more general levels of information (operating system's events - used applications - user goals). In contrast to this, for a user profile description we are going to move in a reverse direction. Hence, the structure of user profile definitions resembles an inheritance mechanism in object-oriented programming languages. Thus, we go from general to more specific, where a more specific description is formed by inheriting all the features of the "parent" and adding some additional ones.

In Figure 5.1 it is possible to see a structure that describes a part of a user profile presented in this section.

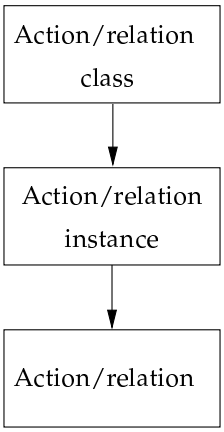


Figure 5.1 Structure of information stored in user profiles

As a general concept we define an *action class* which describes one of the possible kinds of actions. It provides a formal description of a single action without providing any specific details. An action class contains descriptions of events that start and end that action, and possible events between the start and the end. Continuing our previous example, consider a hypothetic action class "Web browsing". It may be defined by the Web browser activity interval. This interval may include other events, such as multiple connection establishments, data transfer, printing, and access to local objects. Those events are caused by child processes of the Web browser, therefore we group all these events into a single action, which is represented by the "Web browsing" class.

Also, if we monitor network packets we may define the same action class by a longer sequence of events. For example, an action class "Web browsing" may include events: a request for connection establishment between some server and a user workstation port 80 (handshake protocol) followed by some information exchange and closing connection.

This is a usage example of the information representation in Chapter 4 to build a source independent behavioral model. The classes provide translations from different information forms (network packets, GUI messages, etc.) to a unique representation. In the example the GUI messages, taken from an operating system's kernel, and network packets, taken from the LAN, lead to the same conclusion that the user is browsing Web pages. As a matter of fact, a number of all possible actions is limited by the operating system tools and additionally installed programs, therefore, a number of action classes is finite and known beforehand.

Each action class is composed of one or more *action class instances*. Each *action class instance* describes an individual group of actions of the same action class having similar temporal characteristics. These characteristics are the mean and standard deviation of the time lengths of the included actions. We assume that the lengths of the actions grouped into the same instance form a normal

distribution with these parameters. Therefore, the distances must be distributed normally in order to be grouped into the same instance.

In Figure 5.2.a it is possible to see an example of an action class that contains three instances N1, N2, and N3 (Figure 5.2.b will be discussed in Section 5.3). The figure shows how temporal lengths of the actions are distributed inside the action class. The distribution is shown by a histogram identifying the amount of actions (number of cases) with the same temporal length at each time point. In our example in Figure 5.2 it is possible to clearly identify three clusters: N1, N2 , and N3 with different parameters of $\mu$ - mean and $\sigma$ - standard deviation.

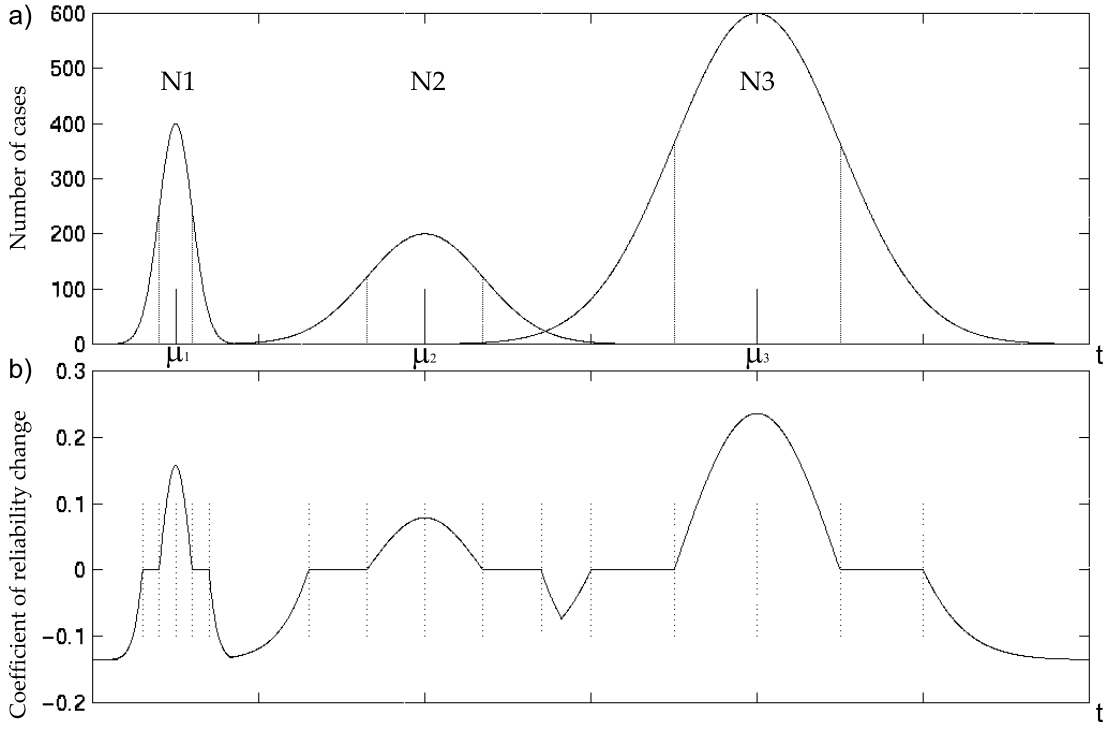


Figure 5.2 Action class "X": a) cases distribution depending on their temporal lengths; b) distribution of the coefficient of reliability changes inside this action class

Suppose that the example in Figure 5.2.a shows a single action class of a user - "Web browsing". Consider him using the Internet during a whole working day. Assume the user starts every working day reading the news from the Internet and it takes in average of 20 minutes. In the figure this regularity is represented by a cluster N1 with parameters $\mu_1$ and $\sigma_1$. After this he handles his paper work till lunch. During lunch he likes to read funny stories from the Internet (average time 50 minutes) - represented by N2 with parameters $\mu_2$ and $\sigma_2$. In the afternoon he works with clients and has to connect to some database on the Internet (average time 120 minutes) - represented by N3 with $\mu_3$ and $\sigma_3$. In this case despite intersections it is possible to clearly identify three clusters. Each

of those represents one of the discussed cases. They are the instances of the "Web browsing" action class. These instances represent the same action, but they differ by their temporal parameters. The reason for this is that the user has different patterns of the Internet usage. Sometimes he does it occasionally (look for a word translation in an online dictionary) without any regularity and sometimes he spends time there accomplishing his everyday tasks, which are indissolubly synchronized with his everyday rhythm.

An instance is *stable* if cases, grouped in it, form a normal distribution with the parameters $\mu$ and $\sigma$. The instances in the above example are described by three normal distributions each of which has its own parameters $\mu$ and $\sigma$. Finally, the amount of actions included in the instance is described by a parameter $n$. It is used for the calculation of usage frequencies of instances inside a single action class.

In this section we presented primitives used to represent information about user actions with their temporal parameters. These primitives will be used in this and the following chapters to describes user events by forming actions, classifying them, and splitting them into instances of the same action class.

## 5.2 Relations between Action Classes

Looking at previous work we may summarize that traditionally, in anomaly detection, user profiles have been built based on different characteristics, such as consumed resources, command count, typing rate, command sequences, etc. In these cases information analysis has been made using system log files, command traces, and audit trails.

In most cases the classification has been performed based on the sequence of events in time. However, the sequential data is not the only information that is possible to get from a stream of discrete events. It also contains some hidden information that is usually neglected: time relations between events are not taken into account at all or only very little attention is given to them. For example, instance-based learning algorithms developed by Lane and Brodley (1997b and 1999) use keystroke sequences that do not contain any temporal information. Systems that use rule-based techniques, such as MIDAS (Sebring *et al.*, 1988), NIDES (Javitz *et al.*, 1993), and ASAX (Mounji, 1997), encode expert opinions in a form of $if - then$ rules, which keep only sequential information. A system that uses a state-based approach (USTAT (Ilgun *et al.*, 1995) - state transition analysis IDS, IDIOT based on Colored Petri Nets (Kumar, 1995)) make their decisions based on tokens moving through states sequentially representing landmarks of attacks. The other systems (for example, NADIR (Hochberg *et al.*, 1993), Haystack (Smaha, 1988), DIDS (Snapp *et al.*, 1992), and EMERALD (Porras and Neumann, 1997)) do not directly employ temporal information in building misuse/normal behavior models for later detection of anomalies or attacks.

Sometimes time relations play a crucial role in attempting to classify events, i.e. determining whether an event is a part of anomalous or normal behavior. To be able to discover and later use relations between actions, it is necessary to define additional concepts. In the previous section we introduced notions related to actions and their representation. Here we define the concepts related to the temporal relationships between actions, their discovery and usage. The notions provided here are similar to those in the previous section.

A *Relation class* is a notion that describes one of all possible relations between two actions: $Action_i$ and $Action_j$. It is defined as one of the basic relational classes. The relation class describes one of the possible relations between two actions without providing its temporal details. The number of possible relation classes is the same as the number of basic relations - two.

A *Relation instance* describes some set of relations that have similar temporal characteristics and each of them belongs to the same relation class. In other words, a relation instance describes some distribution of temporal parameters of relations grouped by this instance.

A *Relation* is one of the basic relations (one of the two presented in Chapter 4) between any two actions $Action_i$ and $Action_j$. It is characterized by a temporal distance $t$ between these actions. Thus, the relation type is a qualitative parameter that describes what kind of relation it is, and the temporal distance is a quantitative parameter that describes the temporal characteristics of the relation.

Consider the web browsing example in the previous section. Assume that when the user works with clients (third instance N3 in the action class "Web browsing") he has to answer by email to some of their requests. In this case it is possible to note that the action "Web browsing" includes numerous "E-mail checking" actions and these checks are connected to each other by a relation class *before*. If, for example, the answer to a single user request takes 20 minutes on average, then it is possible to note that "E-mail checking" actions inside the "Web browsing" action are connected by a relation instance of the class *before* with a parameter $\mu = 20$.

To represent relationships between actions we use a square matrix, $N \times N$ - *relational matrix*, where $N$ is the number of the action classes. In every cell $\{i, j\}$, the matrix holds relational classes that are allowed between every action classes $i$ and $j$. Thus, cells $i, j \ when \ i > j$ contain direct relations for $Ai$ and $Aj$, and cells $i, j \ when \ j > i$ contain the reverse ones.

Continuing the above example, Table 5.1 shows the relational matrix for the case described above. There are two instances present: "Web browsing" and "E-mail checking". There is no relation discovered between the different "Web browsing" instances, thus the table reflects this by an empty cell (showed by "-"). Then it is possible to see that there is a relationship $during$ that connects the instances "E-mail checking" and "Web browsing". The "E-mail checking" is connected with another "E-mail checking" by a relationship $before$. A cell that

contains ”o” represents a reverse relation to that already described in the cell (2,1).

Table 5.1 Relational matrix for two instances

| | **Web browsing** | **E-mail checking** |
|---|---|---|
| **Web browsing** | - | o |
| **E-mail checking** | $during(\mu, \sigma)$ | $before(\mu, \sigma)$ |

Using the relational matrix we may check whether there is a certain relation which exists between any of two classes and compare the parameters of the current relation with the ones stored in the matrix.

## 5.3 Detecting Abnormal Behavior

In this section we are going to discuss how to use the action and relation classes to discover abnormal behavior by monitoring deviations between the current user behavior and a user behavior model stored in the profile as $N$ action classes and a relational matrix. Every action class of the model has one or more instances that represent some user action characterized by the statistic parameters of the corresponding time distribution - mean and standard deviation. All possible relationships between instances are described in the relational matrix, every cell in which contains relation instances with their temporal parameters.

To build a behavioral model it is necessary to train a classifier, which creates a profile for each user during this process. It is performed by submitting actions issued by users to the classifier. To build a profile for a certain user the classifier takes each action issued by the user and puts it into an action class where this action belongs to (the name of the action determines the correspondent action class). Then it determines the relationships between the current action and the previous actions and calculates the temporal distances between them. For the next step the classifier updates the relational matrix by putting the obtained relations into the corresponding cells in it.

After some time the classifier is ready to be used. Empty action classes show that the user does not perform such actions at all. Empty relation classes show that such relations are not present between certain pairs of actions. Classes containing homogeneous data, in which it is not possible to determine any distribution, are not used for classifications. They show that there are no strong patterns in usage of some programs in the case of action classes or there are no relationships between pairs of action classes in the case of relation classes. The rest is used to monitor the user behavior and detect anomalies. Figure 5.3 shows the algorithm of the decision making of the classifier. Below we are going to discuss the classification mechanism in Figure 5.3 step by step.

Deviations from the values stored in a user profile are considered as a consequence of the abnormal behavior. To estimate the value of deviation we

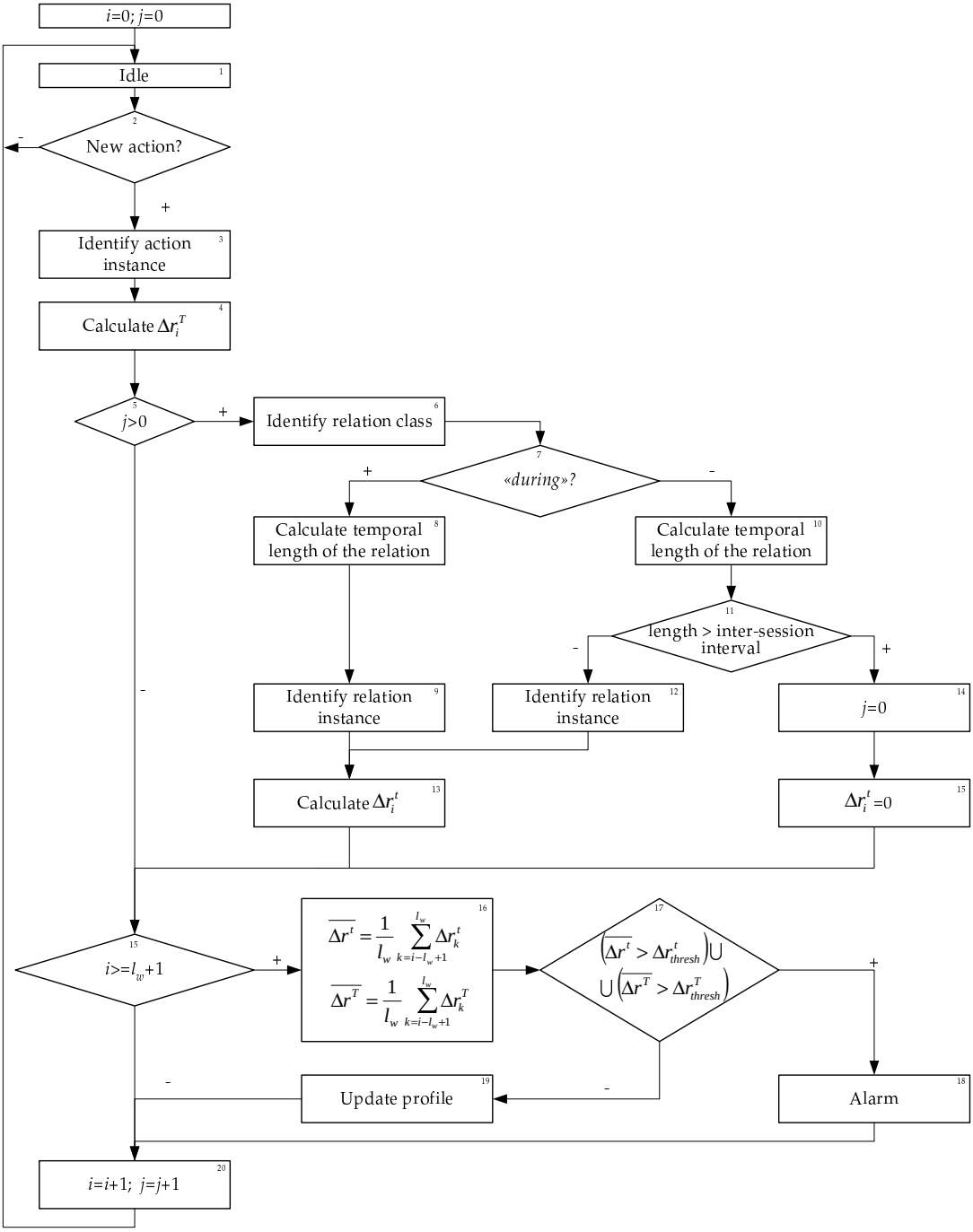


Figure 5.3 Classification algorithm of the classifier that uses a relational matrix to describe a user behavioral model

use a *coefficient of reliability*: $r = [0, 1]$. It is assigned to every active user and shows probability that the user is the one who he/she claims to be. During classification the system monitors deviations between the expected or predicted user behavior and the current one. According to these deviations it calculates some penalty (negative) or encouragement (positive) value, which reduces or increases the coefficient $r$ by the value of $\Delta r_i$.

At the beginning of the classification process the coefficient of reliability is assigned to be $0.5$ since there is not yet any evidence, neither of distrust nor of trust. Therefore, the coefficient of reliability has an ability to grow as well as to diminish. If it crosses a certain threshold it means that there is a sequence of actions where the parameters of each action are not in admissible intervals. It is considered as a case of abnormal behavior and thus, an alarm should fire.

The coefficient change value $\Delta r_i$ shows the amount of change of the coefficient of reliability and it is calculated at each step of classification. It consists of two parts: $\Delta r_i^T$ - change is due to deviations between the current and the expected value of the action's temporal length, and $\Delta r_i^t$ - change is due to deviations between the current and the expected values of the relation's temporal length.

$$\Delta r_i = \Delta r_i^t + \Delta r_i^T \tag{5.1}$$

The classification process is idle (1)[1] until a new action comes. Each time the system receives a new action (2) for classification it needs to determine the correct action class and instance. Finding the correct class is relatively easy. Knowledge of an action's name points to a certain class. After this it is necessary to find a correct instance inside the class.

Below we present our way to automatically determine the action class instance, where the current action belongs (3), among several ones inside the class:

1. determine a normal distribution where a new action belongs: $T \in [\mu - 2\sigma, \mu + 2\sigma]$;

2. if there is more than one interval found in step one, then find $min\ |\mu - T|$ among them;

3. if there is more than one instance with the same temporal distances between them and the current action, choose the one with the smallest standard deviation: $min\ [\sigma]$.

In Gaussian distribution 95%, of all cases are lying in the interval $[\mu - 2\sigma, \mu + 2\sigma]$. During the first step we use this to determine where a new case belongs. If there are some action class instances that are overlapping each other and the new action is in the overlapping area, then we apply steps two and three to introduce the additional restrictions to find the right instance for the new action.

After the classifier identifies the correct instance it has to check how well the new action fits into this instance (4). In other words, it calculates the action part ($\Delta r_i^T$) of the coefficient of reliability change for the current action.

How do we use these temporal characteristics of action and relation instances to detect abnormal behavior? Well, if we take one action class instance it

[1] This identifies the state in the classification algorithm we currently discuss. It refers to the number of the box in Figure 5.3.

is described by normal distribution parameters $\mu$ and $\sigma$. We have separated all the area outlined by the curve presenting this distribution into three areas:

- Area limited by one $\sigma$, i.e. 68% of the whole area. In the case when a new action is in this area we consider this as a strong match that supports this instance. Therefore, the coefficient of reliability should be "encouraged".
- Area between one and two $\sigma$, i.e. 27% of the whole area. When a new action appears in this area it is a weak match. In other words, we have proof that the new action belongs to this instance, but it does not support it as much as in the previous case. Thus, the coefficient of reliability should not be changed.
- Area beyond two $\sigma$. In this situation the penalty for the coefficient of reliability should be applied.

Figure 5.2.b visualizes the idea. Below we present a formula for the calculations of the coefficient of reliability changes:

$$\Delta r_i^T = \begin{cases} \nu_i \times exp\left(\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) - exp(-2)\right), & 0 < t \leq \mu - 2\sigma \\ 0, & \mu - 2\sigma < t \leq \mu - \sigma \\ \frac{n}{c} \times \left(exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) - exp(-0.5)\right), & \mu - \sigma < t < \mu + \sigma \\ 0, & \mu + \sigma \leq t < \mu + 2\sigma \\ \nu_i \times \left(exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) - exp(-2)\right), & \mu + 2\sigma \leq t < \infty \end{cases} \tag{5.2}$$

where $c$ is the coefficient that limits $n$ and determines the system's sensitivity to deviations and, $\nu_i$ is the coefficient of security significance of the action (in other words, how much danger of improper usage of this action may cause); it is defined by the system administrator for each action.

The calculations of the $\Delta r_i^T$ part for the coefficient of reliability change are based on the following aspects:

- how big the difference is between the predicted and the current action lengths;
- the security significance of the current action;
- how big the standard deviation of the current action class instance is.

After this the classifier checks whether the current action is the first action during the current session (5). If it is, then the classifier returns to the idle mode (1) and waits for another action from the same user. In the case it is not the first action in this session the classifier checks the transition between the current and the previous actions. It checks for the type of basic relation (6). If the relation

type is *during* the classifier calculates the temporal length of the relation (8) as it shown in Figure 4.2 of Chapter 4. This calculation is followed by the identification of the correct relation instance. It is done in the same way as for the action class instance.

If the relation class is *before*, the classifier calculates the length of the relation (10) (Figure 4.2). The obtained length is compared with a certain threshold (11). This threshold is established to define an inter-session interval in case the user does not log out after each session. The threshold shows that if there is a certain amount of time passed the new action may be considered as a start of a new session. In this case (14, 15) the classifier initializes its parameters and classifies the action as the first in this session.

If the current action is considered within the same session then the classifier looks for the correct relation instance (12) for this action. On the next step, despite of the relation class identified, the classifier calculates the relational part ($\Delta r_i^t$) of the coefficient of reliability change (13). The same equation 5.2 is applied to calculate $\Delta r_i^t$. These calculations are based on the following aspects:

- how big the difference is between the predicted and current transition lengths (for $\Delta r_i^t$);
- the security significance of the current action;
- how big the standard deviation of the current relation transition is (for $\Delta r_i^t$).

In order to detect sudden changes of the user behavior we use a sliding window to analyze the behavior inside it (15). The width of the window will be determined and discussed in Chapter 9. If there are not enough actions in a queue to fill the window then the classifier goes to the idle mode (1). Otherwise, it calculates the average values for the parameters $\Delta r_i^t$ and $\Delta r_i^T$ inside the sliding window (16).

When the average values are calculated they are compared with established thresholds (17). If one of the average values ($\overline{\Delta r^t}$ and $\overline{\Delta r^T}$) is bigger than it should be then the system administrator is notified and has to investigate the alarm (18). If both of the average values lie inside the defined safe interval then the user profile has to be updated taking the new action into account (19).

Below we consider the update of the temporal-probabilistic characteristics for an action class instance. In order to save computer resources a system does not need to keep a history of events, it needs only to have two parameters that describe the distribution inside this instance. Therefore, if a new action supports a current instance we update its temporal parameters. For every node $i$ its mean $\mu_i$ and standard deviation $\sigma_i$ are calculated each time it is being used for classification:

$$\mu_i = \frac{\mu_{i-1} \times (n-1) + T}{n} \tag{5.3}$$

$$\sigma_i = \sqrt{\frac{\sigma_{i-1}^2 \times (n-1) + (\mu_i - T)^2}{n}} \tag{5.4}$$

where $n$ is the number of cases in this action class instance and $T$ is the temporal length of the action being classified.

The same formulas may be applied to calculate the temporal parameters of relation the instances. In this case $T$ is substituted by $t$ - the temporal length of the transition being classified.

At the end of the classification process the coefficient of reliability of the user $r$ is summed with the $\Delta r_i$. If the coefficient of reliability $r$ is lower than a certain threshold, an alarm is fired. In Figure 5.2 it is possible to see an example of some class "X" and a graphical representation of the distribution of the coefficient of reliability changes inside this class.

## 5.4 Summary

Generally the anomaly detection approaches classify the incoming events based on the distance to the members of different classes or on the probability of falling within known patterns. Usually the learning algorithms used for anomaly detection are reduced to learning on spaces with nominal-valued attributes (decision trees, data mining, etc.), that requires an assumption to be made that the attributes are independent. During the classification the search is performed through these structures to examine each feature independently from all others. As a result internal relationships between them are ignored and a complete model of normal behavior can not be created. In our classification approach we were able to circumvent this problem by transforming the data to a universal temporal representation that explicitly encodes such relationships and allows the use of them in the classification.

Another problem is the automatic handling of the concept drift. In anomaly detection each classifier during its normal operations (after initialization) has to decide whether to add an event into the profile or not. This decision is especially critical in the presence of the concept drift when an unknown behavior may be resulted by normal user behavior changes or an intruder. The techniques used in machine learning (such as inclusion misclassified instances into the profile) to handle these situations are not suitable for anomaly detection since the anomaly classifier is not aware of misclassifications. In our classifier we introduced a feedback that allows the updating of a profile each time the new instance is classified as normal. It leads to an automatic update of the action and relation instances. They may appear, disappear, merge, and split with time reflecting normal changes in the user behavior and preserving a precise and up-to-date behavioral model for each user.

The main assumption behind our relational matrix approach is that the behavior of each user follows regularities that may be discovered and presented

using a limited number of action and relation classes. The profiles are created by forming a vector of $N$ action classes each of which contains several instances. In other words, it contains several temporal patterns of some action. Every profile also contains a matrix of relation classes (they are similar to action classes but describe relations). This matrix allows us to check whether a relation is valid between every pair of action classes. Using the described profiles a monitoring system evaluates every user action according to its length and relations with the previous and the next actions. During classifications a coefficient of reliability is changed. Based on it, a decision is made whether the current behavior is normal or anomalous.

The presented relational matrix approach has some advantages. It is relatively simple and easy to visualize. It is quite fast and it does not require much computational power since it does not require many calculations. The user profile is represented by action and relation classes, which are described by the same temporal parameters, and thus, it is possible to use the same formulas to calculate them. At every time point every class contains several instances and therefore, may be described by some curve (as in Figure 5.2a). To eliminate some calculations it is possible not to describe every instance as a distribution of temporal parameters, but as a distribution of a coefficient of reliability change value (as in Figure 5.2b). It would not require calculations of this value at every step and thus, increases the classification speed.

The presented in this chapter approach does not provide an in-depth analysis of user behavior[2]. This approach is aimed to catch short patterns - activities (two actions connected by a relation). It performs quite well (the performance discussion is in Chapter 9) in different categories of users without losses in precision in certain categories of "casual" users, that use their accounts from time to time, and their behavior does not contain deep hidden regularities that may result in long patterns. Other anomaly detection approaches are not performing well in monitoring this category of users. The anomaly detection approaches require a longer training period and a wider sliding window width (Anderson *et al.*, 1995), (Lane and Brodley, 1999), (Lee *et al.*, 1999a), and (Lunt *et al.*, 1992). In most cases the anomaly detection approaches require thousands of events/tokens to initialize a classifier and a sliding window in hundreds events of width. It often happens that the length of the sliding window is longer than the average session length of such a user, which results in the impossibility to make even a single classification during a single session, which leads to the impossibility to classify at all or an extremely long time-to-alarm. This approach only requires hundreds of such event/tokens for learning and the length of the sliding window is normally tens of them (exact numbers and discussion are provided in Chapter 9).

[2] We are going to extend it in the next chapter in order to build a comprehensive model of user behavior.

# 6 DETECTING ANOMALIES IN USER BEHAVIOR USING TEMPORAL-PROBABILISTIC TREES

It has been shown that human interaction with computer systems shows regularities in almost everything they do (Davison and Hirsh, 1998). This assumption was successfully used to build systems that when applied to sequences of user actions learn over a period of time to an individual's pattern of use (Lane and Brodley, 1999). Moreover, the behavioral regularities of a certain user may be closely linked with the users' plans and goals (Albrecht *et al.*, 1997). Much work to date has resulted in ad-hoc approaches such as simply capturing user preferences (usually only sequential) at a shallow level ignoring the more difficult problem of capturing the user intentions (Lee *et al.*, 2000). Here we provide one of the possible solutions to the problem. We develop an approach that covers many aspects of user behavior by discovering not only the sequential user preferences, but also the temporal ones. The approach connects discovered behavioral patterns with the user's intentions and goals by generalizing the information about the user behavior (from events to activities). As a result a user profile is represented by a set of user goals from one side and the actions usually involved in achieving these goals from the other side.

In this work there is also the underlying assumption that every user performs similar sequences of actions to achieve a certain goal and they are supposed to have similar temporal characteristics of actions as well as relationships between them. We call these sequences *patterns*. The main goal of user profiling is to model the characteristic aspects of user behavior. A pattern characterizes a kind of user activity that he performs to achieve his/her goal. Thus, the pattern is a sequence of instances connected by transitions and the simplest pattern is a single instance.

In this chapter we present another approach to detect anomalies in user behavior. It may be considered as an extension of the approach described in the previous chapter. The main idea of the approach is to obtain information about

normal user behavior, encode it in a set of patterns that represents common sequences in user behavior, and finally use these patterns to detect anomalies. The set of patterns forms a general model of user behavior that is presented by a tree, where the patterns form branches of this tree. Using sequential information extracted from multiple strings of events we are able to construct the probabilistic network (presented by a tree). However, the sequential data is not the only information that is possible to get from a stream of discrete events. It also contains some additional information in the form of relations between events, which often play a crucial role in the classification of events, i.e. determining whether an event is part of anomalous or normal behavior. Therefore, our approach is also aimed to extract and encode the temporal and sequential patterns and build the tree based on them.

## 6.1 Temporal-Probabilistic Tree Definition

Recognition of user behavior is a problem of interpreting data which comes from a dynamically changing environment; therefore it is a problem of intelligent data analysis, where probabilistic graphical models have been recognized as one of the key paradigms (Bellazzi *et al.*, 1998). These models allow a compact description of complex stochastic relationships; thus, we use the probabilistic trees to represent the user's behavior and its changes. The temporal information will be added to every node and edge of the probabilistic tree and thus, we are going to call it a *temporal-probabilistic tree*.

In order to build a temporal-probabilistic tree, sets of actions with similar temporal characteristics - instances should be formed. Then it is necessary to find sequences, using created sets, that in each sequence the corresponded pairs of adjacent instances will be connected by temporal relations that have similar temporal characteristics.

In this section we define a temporal-probabilistic tree aimed to represent peculiarities of user behavior. Our main goal is to be able to automatically discover and use sequential patterns of user behavior as well as temporal. We assume that a user's typical behavior corresponds to the individual temporal-probabilistic tree $S(G, E)$.

On the learning stage we suppose that the system is closed, in other words it uses only "good" examples for learning. Thus, on the initial stages all actions in user behavior are considered as normal.

The temporal-probabilistic tree that describes user behavior is defined as $S(G, E)$ where $G$ is a set of nodes $\{G_i^k\}$ and $E$ is a set of edges $\{E_i^k\}$. Every node $G_i^k \in \{G\}$ on level $k$ represents a certain action instance $I_i^A$, which is a set of actions $\{O\}$ with similar temporal-probabilistic characteristics (level $k$ is defined as a shortest distance from this node to the root).

Each action instance $I_i^A$ inside an action class $A$ is represented as $I_i^A$($A$, $n$, $\mu$,$\sigma$), where parameters $\mu$ and $\sigma$ define the time distribution of actions' lengths

$T_i$ and they are calculated over the corresponding set of actions $\{O\}$. In order to know how often a certain node is used, $n$ is stored for each node. It is the number of transitions when the process of the user's behavior classification have gone through the node $G_i^k$. The probability that this node on level $k$ will be chosen next time is $p_i^k$ and it is calculated as:

$$p_i^k = n_i^k / \sum_{\forall G_i^k \in \{G^k\}} n_i^k \tag{6.1}$$

A set of actions $\{O_1,\ ...,\ O_n\}$ of the same action class forms an action class instance $I_i^A$ if it is possible to describe a distribution of the actions' temporal lengths $\{T_1,\ ...,\ T_n\}$ by the following parameters:

$$\mu = \sum_{i=1}^{n} \frac{T_i}{n} \tag{6.2}$$

$$\sigma = \sqrt{\sum_{i=1}^{n} \frac{(T_i - \mu)^2}{n}} \tag{6.3}$$

This instance is valid and may be used for the classification if the following conditions are met:

$$n \geq n_{min}. \tag{6.4}$$

$$\sigma \leq \sigma_{Tmax}, \tag{6.5}$$

Here the parameter $n_{min}$ defines a minimal amount of actions that may form an instance of an action class - it has to be statistically large enough to minimize possible mistakes. This variable affects the pattern extracting process, i.e. regulates its sensitivity. The smaller the value $n_{min}$ is the more instances (and hence patterns) of user behavior the process may find. Therefore, it is possible to automatically limit the size of the tree by changing $n_{min}$: if a number of the patterns in the tree ($n_{ptrn}$) is too big, the process makes $n_{min}$ bigger and after that constructs the tree again - as a result a smaller tree with less patterns appears. Similarly, if for any node $G_i^k$ a usage frequency $n$ becomes too small, it shows that the tree contains branches that does not reflect user behavior any more. To prune such branches automatically, the parameter $n_{min}$ should be increased.

Variable $\sigma_{Tmax}$ limits a deviation between the action's temporal length and the parameters of the current instance that is allowed for this action to be included in the instance of the action class. This variable is chosen experimentally: it impacts the algorithm's selectivity, meaning that the smaller the value $\sigma_{Tmax}$ is the more precise description of user behavior the algorithm creates (patterns are more detailed). Too big and too small values of $\sigma_{Tmax}$ lead to false negatives and false positives correspondingly. Thus, by carefully choosing $\sigma_{Tmax}$ it is possible to minimize the amount of false negatives and false positives.

A single edge $E_i^k(Arel, n, \mu, \sigma)$ represents a certain transition between two connected nodes from consecutive levels. It connects a node $G_i^{k-1}$ on the level $k-1$ with $G_j^k$ on the level $k$ by a relation $Arel$. By analogy with the node definition $\mu$ and $\sigma$ are the parameters of the distribution for transition lengths. Since the edge represents a relationship between the nodes $G_i^{k-1}$ and $G_j^k$ that it connects, $\mu$ and $\sigma$ describe this relationship. They are calculated using the time history of $t_i$, where $t_i$ is the time of transition between the two nodes: on levels $k-1$ and $k$.

Temporal parameters of each relation between two actions may be calculated in a similar way. As we have already mentioned that there are two time points describing each action: where it begins ($T_{beg}$) and ends ($T_{end}$). Let $e$ represent a point in time, which corresponds to either a starting or a finishing point of an action $e \in \{T_{beg}, T_{end}\}$. Allow $td(e_l, e_m)$ to represent the length of the temporal interval between time point $e_l$ of the action $O_l$ and time point $e_m$ of the action $O_m$; and action $O_l$ is "before" $O_m$. Allow at the same time the actions $O_l$ and $O_m$ belong to instances $I_j$ and $I_{j+1}$ correspondingly. They would form a pair, if they were both within the same sequence; for example, if they both occur during the same session or between two longtime breaks in user activity. Let $k$ be the number of such pairs of actions. Then for the transition between $I_j$ and $I_{j+1}$, the temporal-probabilistic parameters would be:

$$\mu = \sum_{i=1}^{k-1} \frac{td_i(e_l, e_m)}{k} \tag{6.6}$$

$$\sigma = \min_{e_l, e_m \in \{T_{beg}, T_{end}\}} \sqrt{\sum_{i=1}^{k-1} \frac{(td_i(e_l, e_m) - \mu_T)^2}{k}} \tag{6.7}$$

Similarly to the definition of the $\sigma_{Tmax}$ a $\sigma_{tmax}$ is defined. It is an upper limit for the deviation of the transition time length. The influence of this parameter is similar to $\sigma_{Tmax}$. It is necessary to note, that the temporal distance between a pair of actions, that are connected by a stable relation, varies in significantly smaller range than the action lengths. That is why the value of $\sigma_{Tmax}$ should be bigger than $\sigma_{tmax}$.

If the value of $\sigma_t \leq \sigma_{tmax}$ and the amount of such action pairs $n \geq n_{min}$ we can say that there is a *stable* relation between the two sets of actions. If such a relation between two sets of actions is discovered, it means that these sets of actions and the relation between them can be used to describe user behavior and thus, it corresponds to the transition $E_{j+1}$ described by $\mu$ and $\sigma$ between the nodes that represent instances $I_j^A$ and $I_{j+1}^A$.

After defining the temporal-probabilistic tree it is possible to see how the tree describes the layers structure. If a single node $G^k$ represents the action and consists of its starting and a finishing events with the relation that connects them, then the two nodes connected by the edge represent an activity.

Figure 6.1 demonstrates a simple example of an activity represented by a temporal-probabilistic tree. As it is possible to see the activity includes two actions and a relationship between them. Consider a case when the user comes to his work place in the morning and after logging in he/she reads emails that have come during his absence (for example 17.00-8.00). The number of mail messages collected by the server determines the length of the mail session (see upper curve in Figure 6.1). Assume that the number of messages arrived during the night is close to some certain value then the mail session time length would be in the interval $[\mu - \sigma; \mu + \sigma]$ with 95% probability.

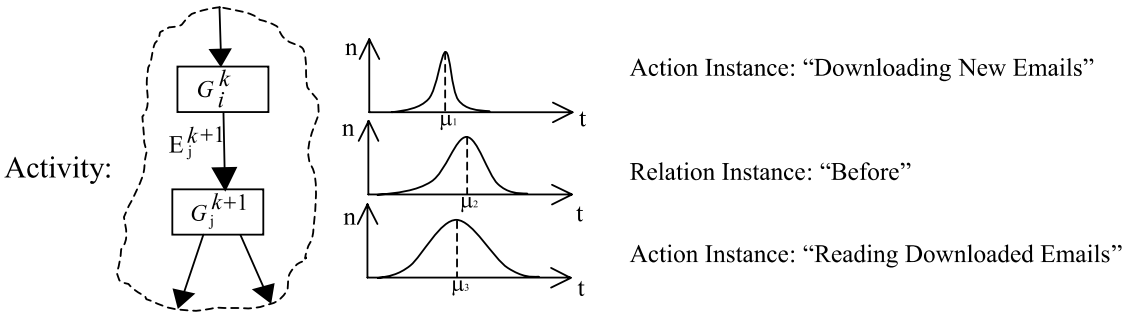


Figure 6.1 An example of activity

After downloading email messages, the user reads them. Some of them he/she reads carefully, some he gives a brief look, others he looks only at "subjects" fields without actually opening them. Taking into account our assumption about the number of emails we may say that the same dependency with the length of this action applies here (see lowest curve in Figure 6.1).

How is it possible to describe relationships between these actions? The user may read messages after finishing downloading the emails or he can do it simultaneously: begin to read messages after first ones have been downloaded and the mail client still receives messages from the server. The edge that connects these actions contains distribution parameters of temporal distances between them (see Equation 4.9). In Figure 6.1 it is presented by the middle curve. Thus, knowing the length of both actions and the temporal distance between them, it is possible to define the relationship between the actions.

In Figure 6.2, an example of a simple pattern is shown. Actions that belong to the class "Edit file", with similar temporal lengths are grouped into a corresponding instance. Similarly, the actions of the class "Send E-mail" with similar lengths form another instance. These instances may have some kind of temporal relationship between them. Thus, two instances with relationship between them form a pattern.

As it is possible to see from the example above, each branch of the tree or its part may represent a certain pattern of user behavior. Therefore, a pattern can be described as $M_i(G', E') \in M$, where $M$ is a set of user patterns, $G' \in \{G\}$

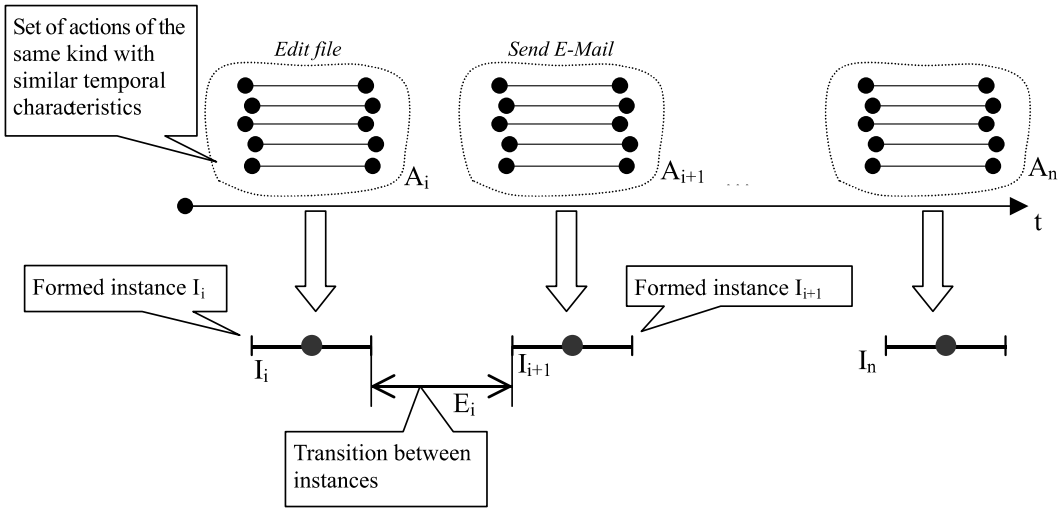


Figure 6.2 An example of a simple pattern

and $E' \in \{E\}$. Hence, it is also possible to define and consider a temporal-probabilistic tree as a set of $n_{ptrn}$ patterns $S(G, E) \equiv \{M\}$.

In Figure 6.3 a visualized example of a user profile is shown[1]. It shows how a model of behavior of a user may be described as a temporal-probabilistic tree. It includes actions in $L$ levels. The type of each action is defined by its class $\{A\}$ and for simplicity a unique number to every action class is assigned. Big numbers identify action classes and the smaller ones define instances of the classes.

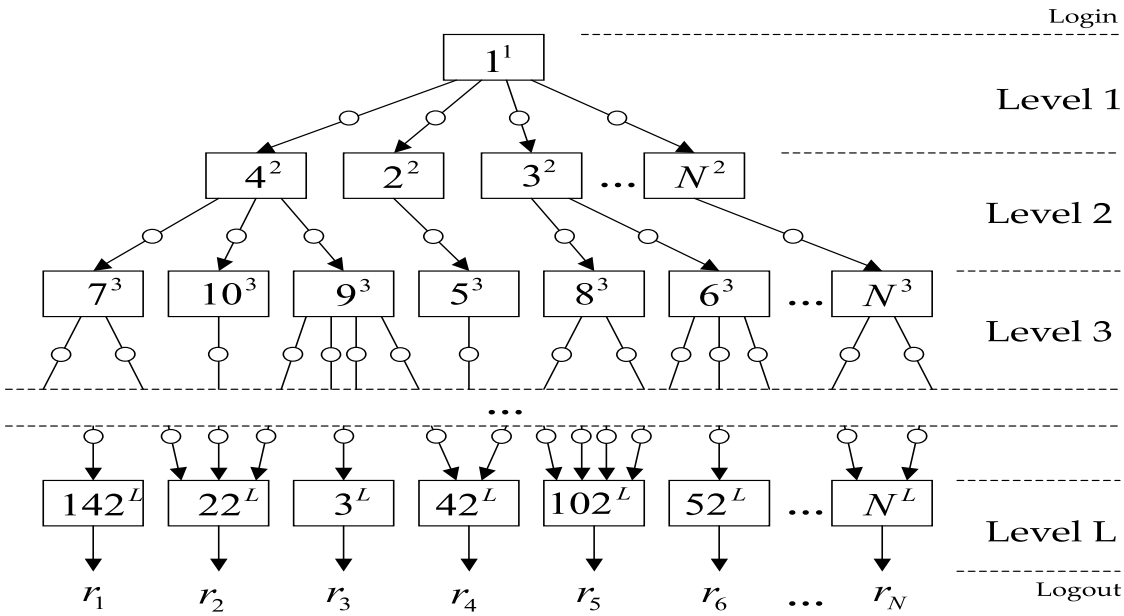


Figure 6.3 A visualized example of user profile

[1] It is not necessary in a real case that all branches are finished on the same level.

User activity begins with a certain root event ($1^1$) and the system has full trust to him/her (it has not yet any reason to distrust). Then the user behavior follows some branch until the session has finished. The system making the profile comparison with the current behavior decreases the user's coefficient of reliability by punishing it when deviations occur. At the end, the final coefficient of reliability ($r_i$) is calculated.

To finish this section we would like to point out, that it is possible that several nodes may represent the same instance. In other words, a single instance may be included into different patterns of user behavior. In the next section we will show how to create a temporal-probabilistic tree using instances of action classes and relation classes.

## 6.2 Training the Temporal-Probabilistic Tree

In this section we are going to focus on pattern extraction from a stream of actions. In other words, how to automatically discover repeating sequences of user behavior and how to create a temporal-probabilistic tree using these sequences. It pursues two objectives: to maximize the user recognition probability or distinction probability and to minimize the size of the tree. Figures 6.4 and 6.5 depict the process of the temporal-probabilistic tree initialization and in this section we follow this process step by step in the description.

The temporal-probabilistic tree is implemented as a part of the anomaly intrusion detection model presented in Chapter 3. The anomaly detector of the intrusion detection system takes, as input, a stream of events (for example GUI events) and stores them into the event log. Using a database where all valuable actions (from a security point of view) are predefined, a data converter translates the source stream to a stream of actions (or temporal intervals) [2] and stores them in the action log. The reason that actions are needed to be predefined is to avoid an infinite number of branches in the tree by minimizing the number of nodes, it also allows updating or changing of the system definitions without rebuilding all profiles (each tree will adapt itself to the new behavior automatically).

The predefined actions' templates are initialized with some values - the security significance of each action $\nu_A$. These values provide information about how important a certain action may be for the operating system security. The more critical the action is, the higher value it has. For example, actions like *su* or *chmod* in Unix would have the highest values.

Initially the tree has only one (root) node, which determines a certain user. It is assumed that the anomaly system is closed on the stage of learning (i.e. it uses only "good" data as input). During the initialization of the classifier takes a training set of actions from the action log and determines the action and relation classes and instances. Whole algorithm may be separated into three parts each

[2] This process is described in Chapter 3.

of which is a logical step in each iteration of learning: initialize the tree and optimize it.

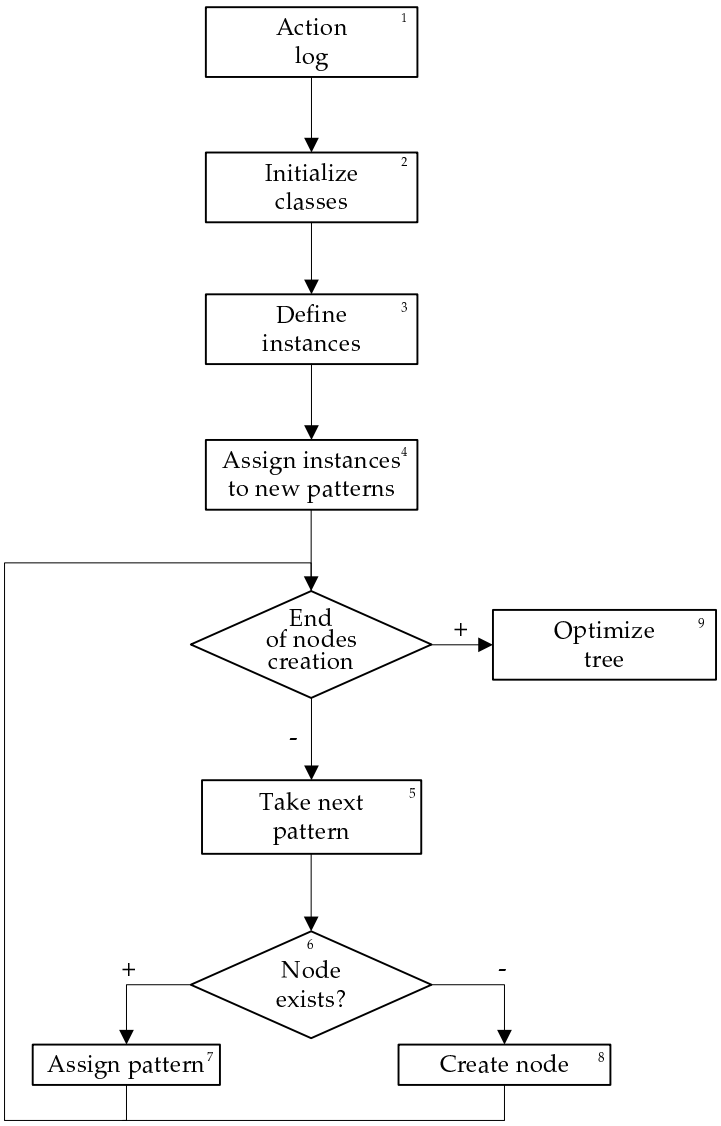


Figure 6.4 Creating set of nodes in the temporal-probabilistic tree

### 6.2.1 Tree Initialization

Figure 6.4 shows the tree initialization phase. At this stage it is not necessary to form complete patterns since they will be automatically extended and revised during each iteration of the tree update. This part of the algorithm is aimed at creating an initial set of nodes in the tree, thus the process is called - initialization. The algorithm goes through the whole training set (in the action log) creating action classes and defining instances inside these classes. When the classifier (Figure 3.2) finishes analyzing the training set and the set of action instances is obtained it creates a set of nodes for the tree correspondent to the set of instances. Below we consider this process in detail.

At the beginning, the initialization of the classes is performed by the classifier (2). Each action from the training set (1) is taken and dispatched to the corresponded action class (according to their names). At the same time relation classes are initialized by putting the relations inside them. In other words, the cases are separated according to their names.

When all actions in the training set are processed the definition of instances is performed by the classifier (3). It takes each action and relation class and performs cluster analysis inside it. After this number of cases in each cluster is compared with $n_{min}$ - minimal number of cases required to form a single instance. For each cluster that has the number of cases greater then $n_{min}$ the classifier is able to form an instance. It takes all the cases inside each of such instances and calculates the parameters of distribution. Later the cases inside of these instances are disregarded and therefore, only the calculated parameters are kept (to save space).

If the number of cases in the class is less or equal than $n_{min}$ it means that the classifier is not able to define an instance. In other words, there is not enough cases to form a statistical distribution. The classifier keeps such sets of cases, they might be completed (and therefore, transformed into the instances) later - during normal classifier operations.

The instances definition process is performed similarly for the action and relation classes. It is continual. A set of instances is updated when new actions come (meaning that actions of this class and with these temporal parameters have not appeared before) or when the value of deviation for some instance overpasses the limit value $\sigma_{Tmax}$.

It is necessary to note, that as was discussed in the previous chapter, patterns are created automatically (this is one of the strengths of the approach). They appear during statistical processing of the actions from the training set. This automatic process is controlled by two variables - $n_{min}$ and $\sigma_{Tmax}$. The former one makes sure that a possible statistical error is on an acceptable level. The latter one defines the precision of the pattern. For example, after defining instances there are possibly two action instances: ”text editing” and ”printing”, and there is a relation instance connecting them. This is considered as a pattern. It may be extended by further observations. For such patterns, $n_{min}$ ensures that there are statistically enough cases to support this pattern. When $\sigma_{Tmax}$ is smaller the discovered patterns are more precise (the description of the behavioral model is more precise). However, for bigger values of $\sigma_{Tmax}$ it is easier to discover the behavioral patterns and a size of the resulted tree is smaller (due to smaller amount of instances).

After the patterns have been discovered, the classifier starts an iterative process of nodes creation. On each iteration an instance is taken. Each newly created instance $I_i^A$ is assigned to the new pattern $M_i$ (4). At the same time the usage frequency $n$ of this pattern is assigned to an amount of actions, which belongs to this instance. After that the system checks whether the tree already

has a branch beginning with the node $G_i^k$ (6), which represents the instance $I_i^A$ (for a branch the beginning node is the node on level 1). If it has, then this new pattern is assigned (7) to be represented by this node (the node $G_i^k$ becomes a common node for several patterns). If not - a new node is created (8) that represents the instance $I_i^A$ of the new pattern. This new node forms a new branch of the tree that at the moment contains the only node and represents the only pattern.

This operation is performed over the set of newly created instances. If there is no more pattern left it means that the initial set of nodes is created. Thus, a set of new patterns appears that contains only instances (simplest cases of patterns). After this it is necessary to connect the nodes by edges (9).

### 6.2.2 Optimization

After the temporal-probabilistic tree is initialized it can not be used for user verification. The process of initialization provides only a starting point for assembling the different patterns into a single entity representing a behavioral model. Figure 6.5 depicts the basic steps of each iteration of the optimization process.

The tree initialization process establishes a basis for further behavioral model building. After the initialization the temporal-probabilistic tree has a number of quite short branches that represent initially discovered short patterns. There also exists some redundancy in the tree.

The main goal of the optimization is to complete building the behavioral model. The patterns (tree branches) are arranged in a way that later allows the classifier to follow from one state of the tree to another. Also the redundancy present in the tree is eliminated.

The optimization is performed by looking for concatenation possibilities for each pattern in the pattern set. This allows connecting patterns (branches) or their states (nodes) by edges. During the concatenation for each pattern $M_i$ an attempt is made to find another pattern to concatenate with - $M_j$, when the sequence of user actions, represented with the $M_j$, is the most likely continuation of the sequence of user actions represented by the pattern $M_i$. Here we consider in detail how the system can automatically perform it.

Completion of this process ensures the optimal size of the tree and the correct connections between patterns. It also results in the appearance of longer patterns (branches) since several shorter ones may be connected during the optimization. The longer branches reflect a more strict user behavior peculiarities, and hence lead to a smaller amount of errors in user behavior evaluation.

The operation of concatenation is repeated until all patterns presented in the tree have been tried to concatenate with each other. When it is not possible to find a pattern to concatenate, the creation process of the temporal-probabilistic tree is over. This tree is stored in the *user profile database* and it is used for online user behavior classification.

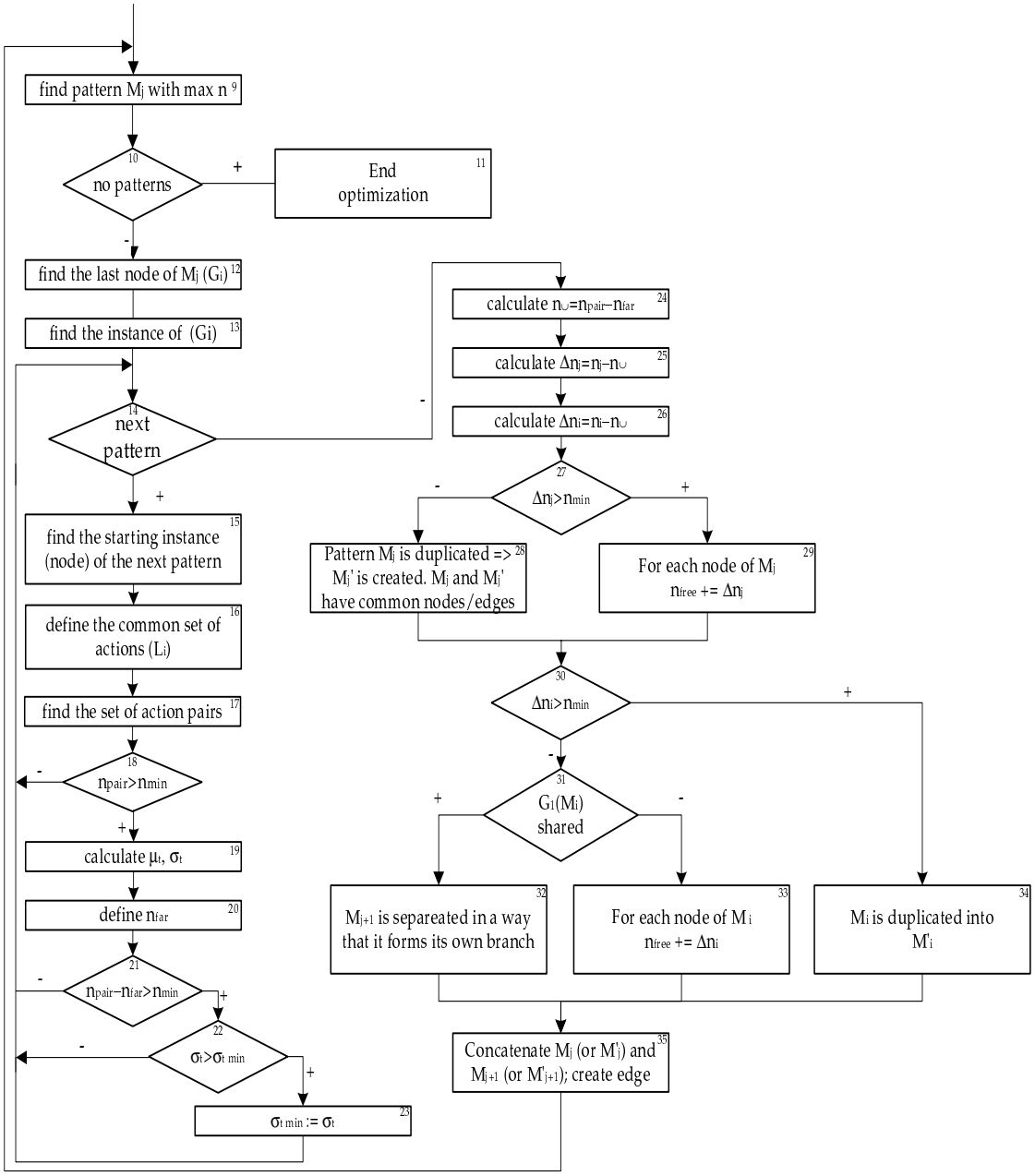


Figure 6.5 Optimizing the patterns

The optimization process is started by sorting all patterns by field $n$ (usage frequency). After this the classifier takes the pattern with the biggest $n$ (9) and looks for concatenation possibilities, then with smaller $n$ and so on, until the last pattern with smallest $n$ is processed (10, 11). In other words, the classifier starts processing the most supported (by cases) patterns and continues towards the less supported. This leads to the creation of the more correct and precise model since the most supported patterns establish the foundation of the model, which is extended by less supported patterns providing additional details and ensuring the completeness of the model.

For each pattern, the system looks for another one to concatenate. The main goal is to find and establish relations (connect nodes by edges) between patterns with the smallest $\sigma_t$ to assure the precision of the behavioral model. For the current pattern $M_j$ it is performed by the following steps:

1. The last node $G_i$ of the pattern $M_j$ is selected (12) and the instance $I_i^A$ is identified that is represented by the node $G_i$ (13).

2. For each pattern the set $L_i$ of actions is created that contains only the actions that belong at the same time to both: node $G_i$ and to the pattern $M_j$ (15, 16).

3. Then for the node $G_i$ a vector of edges is created. The $j$th element of this vector corresponds to an edge, connecting the set of actions $L_i$ of the node $G_i$ with the set of actions $L_j$ of the first node $G_j$ of the pattern $M_j$. For example, let the node $G_i$ be the common node of the patterns $M_i, M_{i+1}, M_{i+2}, \ldots, M_k$ (but it is the last node only of the pattern $M_i$). Let the node $G_j$ be the common first node of the patterns $M_j, M_{j+1}, M_{j+2}, \ldots, M_l$. Then the edge that connects nodes $G_i$ (pattern $M_i$) and $G_j$ (pattern $M_j$) is formed only by the pairs of actions that are taken form the sets $L_i$ and $L_j$ (17).

4. If the number of the obtained pairs is greater than $n_{min}$ (18) it means that the classifier is able to establish a relation and connect the nodes by an edge. If the classifier is not able to connect the states, then it takes a next pattern and goes to step 1 (14).

5. For each element of this vector parameters $\mu, \sigma, e_{prev}, e_{next}$ are calculated (19, 20) using formulas 6.6-6.7.

6. To limit the interval inside which the analysis is performed the $n_{far}$ is introduced (21). It is a number of relations, the temporal lengths of which is not included into an interval, formed by the majority of relations that form a current distribution, i.e. for such relations the following condition holds:
$$\forall |td(e_{prev}, e_{next})| \notin |\mu \pm c_{trans} \times 3\sigma|. \tag{6.8}$$
The constant $c_{trans}$ affects the length of the interval inside which all relations are grouped. It is introduced for flexibility purposes. It allows to affect from outside patterns' concatenation process during systems operations. This constant is set experimentally and must be $c_{trans} \geq 1$. The limitation of the interval allows not to search for stable relationships between events that are situated far from each other (for example, in the morning and in the evening).

7. If $n_{pair} - n_{far} < n_{min}$ it means that it is not possible to establish a stable relation and the next pattern is chosen.

8. If a stable relation can be established a standard deviation of this relation $\sigma$ is compared with the obtained before from the previously analyzed patterns (22).

9. When the element with the smallest value of $\sigma$ is found, the pattern $M_j$ is selected relatively to this element. After that, the system concatenates two patterns $M_i$ and $M_j$.

The search for concatenation possibilities for the current pattern on this iteration is finished (14). Now, it is necessary to perform the concatenation itself.

Before concatenation the amount of actions that will form a new pattern is calculated (24). Let $n$ be the amount of relations forming an edge $E_j$ between $G_i$ (pattern $M_i$) and $G_j$ (pattern $M_j$). Then the usage frequency of the concatenated pattern would be $n_\cup$ and it is calculated as:

$$n_\cup = n_{pair} - n_{far}, \tag{6.9}$$

These $n_{far}$ relations between pairs of actions most likely do not belong to the pattern produced by concatenation and therefore should be excluded. The process of patterns concatenation depends on the following conditions:

- How big is the amount of actions of the pattern $M_i$ that are not included into the concatenated pattern (25):

$$\Delta n_i = n_i - n_\cup. \tag{6.10}$$

- How big is the amount of actions of the pattern $M_j$ that are not included into the concatenated pattern (26):

$$\Delta n_j = n_j - n_\cup. \tag{6.11}$$

- Whether the node $G_j$ is common node for several patterns.

Below we consider all possible cases in details:

1. If for pattern $M_i$ the following condition is met (27):

$$\Delta n_i < n_{min} \tag{6.12}$$

   Then in the pattern $M_i$ all the nodes are selected for concatenation. Knowing the nodes we know the instances represented by these nodes. For each of these instances, the amount of "free" actions is increased by $\Delta n_i$ (29).

2. If the condition 6.12 is not met then the pattern $M_i$ is duplicated (29). While duplicating a new pattern $M_i'$ is created. Both $M_i'$ and $M_i$ patterns

are represented with the common set of nodes and edges (as the pattern $M_i$ had before duplicating).

3. If for the pattern $M_j$ the condition is met (27):

$$\Delta n_j < n_{min} \tag{6.13}$$

then there are two possibilities:

- *Node $G_j$ is shared between the patterns (31).* Then the pattern $M_j$ is detached in a separate branch (32). It means that for the pattern $M_j$ the own branch in the tree is created so that all nodes of this branch would not be shared with any other pattern. Then for the pattern $M_j$ all the nodes are selected. Knowing the nodes, the instances represented by these nodes, are found. For each of these instances the amount of "free" actions is increased by $\Delta n_j$.
- *Node $G_j$ is not shared between the patterns.* Then for the pattern $M_j$ all the nodes are selected (33). Knowing the nodes, for each corresponding instance represented by these nodes, the amount of "free" actions is increased by $\Delta n_j$.

4. If for the pattern $M_j$ the condition 6.13 is not met, then it is duplicated (34). While duplicating a new pattern $M_j'$ is created. For the pattern $M_j'$ its own set of nodes and edges (i.e. own branch of the tree) is created. The nodes and the edges of the new pattern $M_j'$ are identical to those of the pattern $M_j$.

In a case, when it is necessary to duplicate $M_i$ or $M_j$, then only a new pattern $M_i'$ or $M_j'$ takes part in the concatenation.

Finally patterns $M_i'$ ($M_i$) and $M_j'$ ($M_j$) are concatenated (35). It means that a set of $l$ nodes of the pattern $M_j'$ ($M_j$) (with the edges connecting them) becomes the $l$ last nodes of the pattern $M_i'$ ($M_i$). After that each instance $I^A$, for which the amount of "free" actions ($n_{act}$) has been increased during the concatenation process, are checked whether the condition is satisfied:

$$n_{act} \geq n_{min}. \tag{6.14}$$

If it does, a new pattern $M_i$ is created, representing this instance, as if it was a newly created instance. It means, that the system checks whether the tree has already the branch beginning with node $G_i$, which represents the instance $I^A$. If it does, then this new pattern is assigned to be represented by this node. If not then a new node is created that represents the instance $I^A$ of the new pattern. This new node forms a new branch of the tree that at the moment contains the only node and represents the only pattern.

The operation of concatenation is repeated until all patterns presented in the tree are tried to concatenate with each other. When it is impossible to find a pattern to concatenate, then the creation process of the temporal-probabilistic tree is over. This tree is stored in the *user profile database* and it is used for online user behavior classification.

As can be noticed from this section the learning process (the optimization in particular) is very time and resource consuming (compared to a relational matrix approach described in Chapter 5). However, it is performed only once at the initial stages of system operations - during the profiles creation. It is updated automatically and thus, never required to rebuild the behavioral models later due to a concept drift. Also the HIDSUR architecture was designed to ensure the safety of the knowledge base and transmitted information assuring the correct recovery form of different failures (such as LAN failure).

In this section the process of creation of a temporal-probabilistic behavior tree, that represents a model of normal behavior of a user, was described. The process of pattern concatenation produces a tree with longer branches. The longer branches reflect more strictly user behavior peculiarities, and hence leads to a smaller amount of errors in user behavior evaluation.

In the following section we describe how to detect abnormal behavior by matching the current user actions against the user's profile stored in a form of the temporal-probabilistic tree.

## 6.3 Detecting Abnormal Behavior

While performing every day tasks a user repeats the same activities over similar sets of data, while spending approximately the same amount of time. Revealing sequential and temporal patterns of user behavior, it is possible to detect when someone else is pretending to be him/her. As a measure that shows authenticity of a user a *coefficient of reliability* is used.

### 6.3.1 Defining the "When"

The process of online user behavior verification is presented in Figure 6.6. It is similar to a classification process that involves the relational matrix approach. Briefly describing the process, it is possible to say that the classifier goes from state (node) to state of the temporal-probabilistic tree changing the coefficient of reliability according to discovered deviations. Initially, it checks whether the current user action belongs to the node that the classifier was expecting as a next state - probabilistic fraction of the coefficient of reliability change. After that action's and relation's (transition that led to the current state) lengths are compared with their average values from the profile - temporal fraction of the coefficient of reliability change. Based on the comparison results the classifier decides to which direction and how much it is necessary to change the coefficient of reliability.

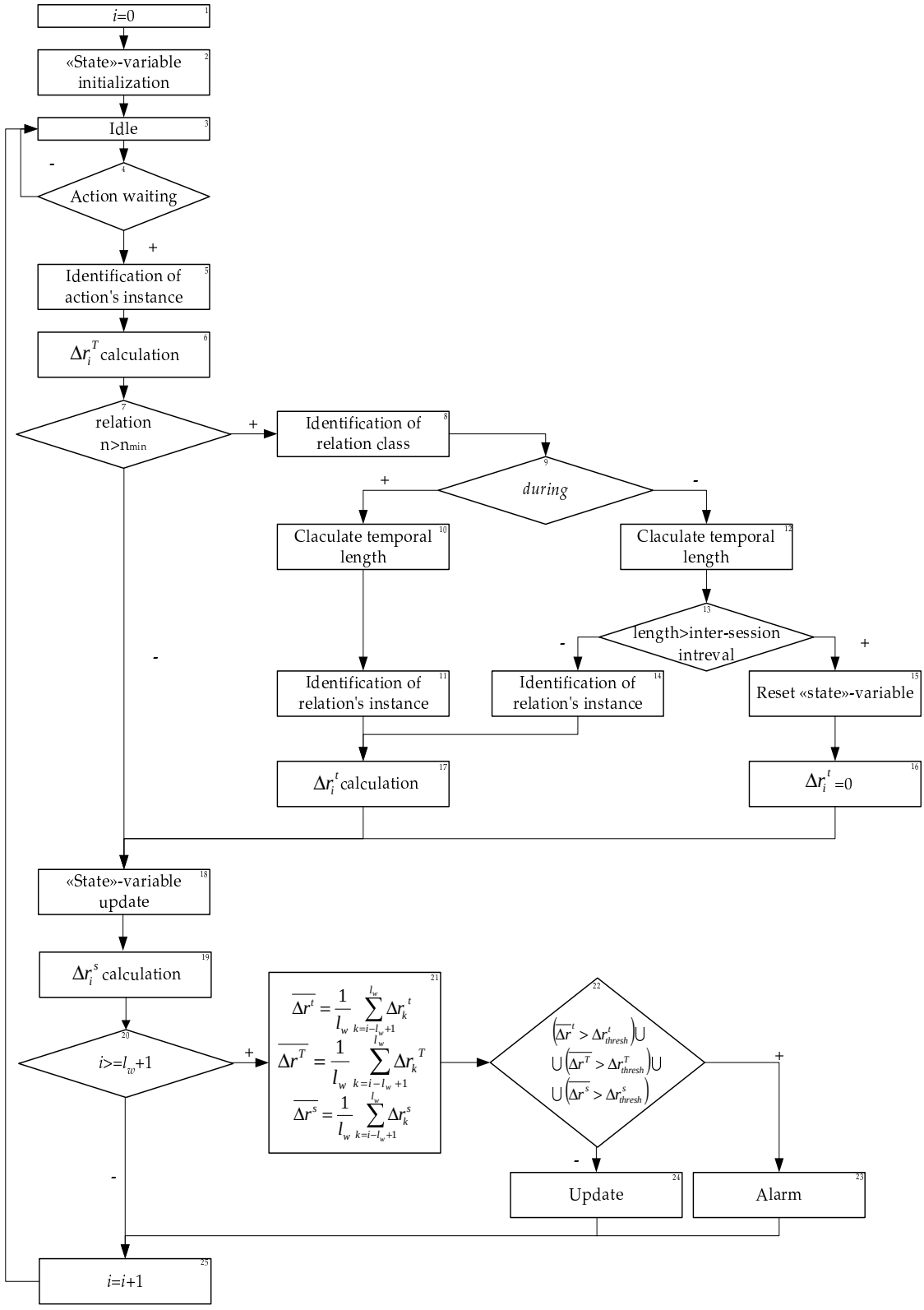


Figure 6.6 Classification algorithm implemented in the classifier that uses temporal-probabilistic trees to describe a user behavioral model

As it is possible to see the coefficient of reliability change on each step of the classification consists of three forming components - the number of features involved in user behavior verification:

$$\Delta r_i = \Delta r_i^s + \Delta r_i^T + \Delta r_i^t, \tag{6.15}$$

where $\Delta r_i^s$ is a probabilistic part showing how much it was expected (according to the user behavioral model in the profile) that the user performs this action. $\Delta r_i^T$ and $\Delta r_i^t$ shows a difference between the action's and the relation's temporal lengths and their average values at this state.

To avoid potential false alarms due to changes in user behavior it is necessary to design the classifier in a way that it would not react as soon as it detects a single abnormal action. There should be some sequence (but not very long) of abnormal event present in the audit log to "convince" the classifier to fire an alarm. Thus, we make an assumption that in the case of normal behavior changes the average value of the coefficients of reliability changes should aspire to zero $\sum_i \Delta r_i \rightarrow 0$. However, in case of repeated occurrences of observations with low probabilities the values of estimative coefficients drop dramatically.

At the beginning of the pattern matching process an intrusion detection system does not yet have any evidence of either trust or distrust, therefore it assigns an average coefficient of reliability to a user (0.5). During the matching process the system updates the coefficient of reliability according to deviations found during the matching of the current behavior with the profile (Equation 6.15) according to the following rules:

- Each time a node is chosen in behavior classification its $n$ is incremented and the parameters are recalculated taking into account a new time $T_i$.

- Each time an edge is used its parameters $n,\ \mu$ and $\sigma$ are recalculated by the analogy with the node.

- When the transition probability of the node diminishes to a certain limit $l$ (since probability calculations based on quantities of transitions this means that this node is not used for classification for the new ones) the system deletes this node, as it does not reflect the user's behavior any more. Changing the limit $l$ we can vary the speed of tree renewal. The higher the level $l$ is, the faster the unused nodes will be deleted and new nodes added instead and vice versa.

- There are two possibilities when a new node(s) should be added:

  - When a new event sequence is detected and recognized as normal, then the system adds new node(s) in order to cover the discovered pattern.

– If a sophisticated pattern is encoded in the tree then it is possible to split it into several simple patterns in order to increase detection precision. Pattern splitting allows calculating coefficients of reliability to be much more accurate, but at the same time it makes the tree grow.

Consider the matching process in details. In our intrusion detection architecture there is a classification module that takes an action stream as an input and calculates a coefficient of reliability change for every action. At the beginning, when the first action $O_1$ comes into the classifier, it starts the matching process by creating a new record ("state" variable) related to this action and its possible continuation. This record includes:

- information about an instance $I_i^A$ where a current action (at the moment - the first action) belongs to,

- the node $G_i^k$, that represents this instance,

- the set of possible states on the next level, that have edges connecting them with $G_i^k$,

- the starting and ending time of the action,

- the time $t_{lim}$ which is a temporal interval during which the next action related to $O_i$ is supposed to appear. The value $t_{lim}$ is calculated as $\mu_t + 3\sigma_t$, where $\mu_t$ and $\sigma_t$ are the parameters of the longest transition connecting $G_i^k$ with any node on level $k+1$ (that is; this edge has the biggest value $\mu_t$ among all the edges coming out of $G_i^k$).

If a certain node on level one in the tree, that represents the instance $I_i^A$, is not present then this record cannot be created. In this case, the amount of "free" actions of the instance $I_i^A$ is incremented by '1'.

The new matching process is launched (and correspondingly, its record is also created (2)) if a new action comes for which its instance does not match any anticipated actions of all matching processes. Similarly, when the new action arrives and there are no matching processes active, then a new matching process is initiated.

When a new action appears in a stream of actions (4) a classifier should determine what instance this action belongs to (5). The action's name determines a correspondent action class. After this the action instance is identified. The identification process is the same as for the relational matrix approach and it was described in detail in the previous chapter, therefore we are not going to concentrate on it here.

After the correct instance is found the classifier looks to what node of the tree this action may correspond. In order to accomplish this task the system searches among possible nodes on the next level, stored in a record of each

matching process. During the search for the next node a transition is defined between the current node and the preceding one. If there are several nodes for which the transition may be defined, then the node closest to $\mu_t$ is chosen. When the next node is found, then the record of the matching process is updated. In the case that there is no match found, the classifier looks for the possibility to launch a new matching process.

At this stage of the action's classification one of its features is identified - correct position in the temporal-probabilistic tree (node). The classifier compares the length of the action with the distribution described by parameters $\mu$ and $\sigma$ taken from the instance represented by the node found. As a result of the comparison $\Delta r_i^T$ (6) is calculated (the way how it is calculated and discussion will be presented later in this chapter).

After the classifier determines the current node it has to evaluate a transition between this and the previous nodes. The edge connecting these states may represent a relation. During online learning new edges may appear. Each edge represents a relational instance. If a number of cases of this instance is not statistically big enough it can not be used for quantitative classification (7). However, the presence of such edges is enough for the classifier not to fire an alarm, therefore they are used for qualitative classifications showing possible transitions. In case there is no a stable relationship between the compared nodes the classification process goes to the next phase. If a relationship is found its relation class is identified according to the relation's name (8).

If the relation class is *during* (9) then the temporal length of the classified relation is calculated (10) and the correct instance inside the relation class (11) is identified (the same way as for actions). After this it is possible to compare how well the current relation fits into the distribution that describes the found relational instance. As a result of the comparison the $\Delta r_i^t$ is obtained (17).

If the relation class is *before* (14) then there are some differences in the calculations of the $\Delta r_i^t$. The temporal length is measured differently (see Chapter 4 for details). Also it is necessary to compare time passed from the last action with inter-session interval (13). If a new session is started the classifier is not evaluating the time interval between the last action of the previous session (16) and the first one of the current session. Also the classifier resets the "state" variable (15).

During the search for the next node each matching process is constantly checking the time elapsed after the current action $O_i$ of this process. If this time interval is longer than $t_{lim}$ then it is necessary to check if there is a pattern with the last node $G_i$ that contains the action $O_i$. For this pattern, a value of usage frequency $n$ is incremented by '1'. It may probably happen that such a pattern does not exist. In this case, all the nodes are selected that represent this pattern. For instances from selected nodes the amount of "free" actions is incremented by '1'. Regardless of the result of the search this matching process for the current session is over.

When the action and the transition were evaluated the "state" variable is updated reflecting the new state of the classification process (18). The probabilistic part of the coefficient of reliability change is calculated $\Delta r_i^s$ (19). There were a number of possible transitions (edges outgoing form the previous node) that lead from the previous state to one of states on the current level. Each transition had a probability that it would be chosen. This possibility depends on how many cases support it - $n_i^k$. Therefore, it may be calculated using Equation 6.1.

In order to detect sudden changes of the user behavior we also use a sliding window (with $l_w$ length) to analyze the behavior inside it (as for relational matrix approach). If there are not enough actions in a queue to fill the window (20) then the classifier goes to the idle mode (3). Otherwise, it calculates the average values for the parameters $\Delta r_i^t$, $\Delta r_i^s$ and $\Delta r_i^T$ inside the sliding window (21).

When the average values are calculated they are compared with established thresholds. If one of the the average values ($\overline{\Delta r^t}$, $\overline{\Delta r^s}$ and $\overline{\Delta r^T}$) is bigger than it should be (22) then the system administrator is notified and has to investigate the alarm (23).

If all of the average values lie inside the defined safe interval then the user profile has to be updated taking into account the new action (24). For every node $G_i$ its mean and standard deviation are recalculated each time it is being used for classification according to:

$$\mu_{T_i} = \frac{\mu_{T_{i-1}} \times (n_i - 1) + T_i}{n_i}, \tag{6.16}$$

$$\sigma_{T_i} = \sqrt{\frac{\sigma^2_{T_{i-1}} \times (n_i - 1) + (\mu_i - T_i)^2}{n_i^2}}. \tag{6.17}$$

The mean and standard deviation of a transaction may be calculated by analogy:

$$\mu_{t_i} = \frac{\mu_{t_{i-1}} \times (n_i - 1) + t_i}{n_i}, \tag{6.18}$$

$$\sigma_{t_i} = \sqrt{\frac{\sigma^2_{t_{i-1}} \times (n_i - 1) + (\mu_i - t_i)^2}{n_i^2}}. \tag{6.19}$$

where $T$ is the time length of the action being classified; $t$ is the temporal distance between the actions[3].

### 6.3.2 Predicting the "How Much"

In order to avoid false alarms, the system needs to distinguish between natural behavior change and malicious training attempts. To accomplish this the coefficients of reliability are used. While classifying the user's behavior, the process of

[3] The issue of concept drift is discussed in the next chapter.

classification searches the tree. It goes from one action to another action on the next level through some edge on the tree. During this process the coefficient of reliability of the user decreases/increases according to the amount of deviations between his actions and the expected or encoded ones.

The more the actual time differs from the expected time, the more punishment the coefficient of reliability receives. There are two possibilities of quantitative differentiation, therefore there are two cases of how the coefficient of reliability may be decreased: $\{T\}$ -actions' time lengths (decreasing occurs when the classification process leaves certain node) and $\{t\}$ - edges' time lengths (decreasing occurs when the classification process went through an edge and enters a node).

On each step of classification a coefficient of reliability $r$ is updated accordingly:

$$r_i = r_{i-1} + c\nu_A \Delta r_i, \tag{6.20}$$

where $r_{i-1}$ is the value of the coefficient on the previous step; $\Delta r_i$ is the coefficient of reliability change on this step; $c$ is the coefficient that determines a system's sensitivity to deviations (chosen by a system administrator); and $\nu_A$ is the coefficient of security significance of this action class, defined by the system administrator for each action class.

Calculations of the coefficient of reliability change are based on the following assumptions:

- how big is the deviation between an average action length and a current action length;
- how big is the deviation between an average transition length and a current transition length;
- how big is the standard deviation of action instance that the predicted action is included in;
- how big is the standard deviation of relation instance that the predicted transition is included in; and
- the security significance of the current action class.

Thus, the value of coefficient of reliability change should depend on the temporal-probabilistic characteristics of an instance and a transition that connects the previous node with the current one. Besides, for the normal user behavior, the average value of this coefficient should aspire to be zero.

The coefficient of reliability change may be calculated as:

$$\begin{aligned} \Delta r_i &= \Delta r_i^T + \Delta r_i^t, \\ \Delta r_i^T &= f(T) - f(\mu_T + 0.67\sigma_T), \\ \Delta r_i^t &= f(t) - f(\mu_t + 0.67\sigma_t), \end{aligned} \tag{6.21}$$

where $f(x) = \sqrt{2\pi\sigma^2} \times \exp\left((x-\mu)^2/(2\sigma^2)\right)$ is the probability density function of the instance time distribution. Dependence of $\Delta r_i^T$ ($\Delta r_i^t$) on time is shown in Figure 6.7.

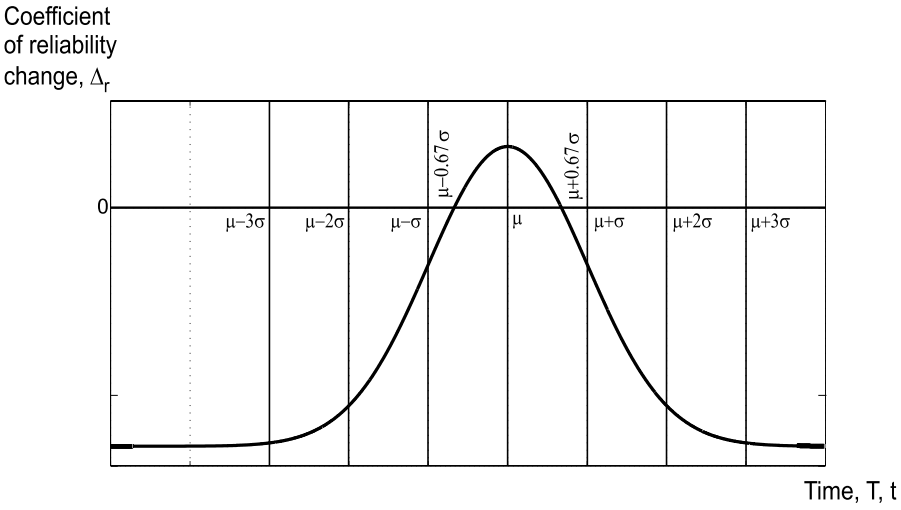


Figure 6.7 Coefficient of reliability change

Normal user behavior should lead to no average change. To satisfy this requirement a shifted probability density function is chosen. For normal user behavior, the action (transition) times are within the interval $[\mu - 0.67\sigma, \mu + 0.67\sigma]$ with probability 0,5 (Borden, 2002). This interval splits horizontally the normal distribution into two equal parts. Therefore, probability density function is shifted to the value of $f(\mu + 0.67\sigma)$, in other words the horizontal axis is shifted to this position (Figure 6.7) and it separates the part where the coefficient of reliability change is positive (above it) and negative (below it).

As can be seen from the formula 6.22 the values of $\Delta_r^T$ and $\Delta_r^t$ can be positive as well as negative. The more the action (transition) time differs from the mean of its instance the smaller the value of the coefficient of reliability change is. If the deviation of the current action (transition) time differs from the mean more than $0.67\sigma$, the user behavior is considered as abnormal rather than normal. Therefore, the coefficient of reliability is "punished" by a negative value of $\Delta_r^T$ ($\Delta_r^t$) and the value of the coefficient of reliability $r_i$ is reduced. If the current action (transition) time differs from the mean less than $0.67\sigma$, the user behavior is considered as normal. Therefore, the coefficient of reliability is "encouraged" by a positive value of $\Delta_r^T$ ($\Delta_r^t$) and thus, the value of the coefficient of reliability $r_i$ is increased. At the end of the matching process the coefficient of reliability $r$ is analyzed and if it is lower than a certain threshold the alarm is fired.

There are two cases when it is impossible to calculate the coefficient of reliability punishment by the method described above. These cases are:

- a new action cannot be assigned to any of the possible nodes as well as to any of the first states in a pattern set; and

- after the appearance of the current action in a stream of actions, more than $T_{lim}$ time has passed; and there is no existing pattern that has $G_i^k$ as a final state.

It is obvious, that in these cases the temporal-probabilistic tree does not contain such a relation. In these cases, the coefficient of reliability punishment is calculated as:

$$\Delta r_i = f(\mu_T + 3\sigma_T) - f(\mu_T + 0.67\sigma_T) \tag{6.22}$$

As can be seen from above, in spite of the probability that this action may belong to the interval $[\mu_T - 0.67\sigma_T, \mu_T + 0.67\sigma_T]$, the value of $\Delta r_i$ is negative. Hence, the coefficient of reliability is decreased showing that the current action does not match the user profile.

The shortcoming of this function is its relatively large computational complexity. To overcome this it is possible to approximate it by a linear function. Such linear functions of change for the coefficient of reliability may be calculated as:

$$\begin{aligned} \Delta r_i &= \Delta r_i^T + \Delta r_i^t, \\ \Delta r_i^T &= -(f(\mu_T) - f(\mu_T + 0.67\sigma_T))\frac{|T - \mu_T| - 0.67\sigma}{0.67\sigma_T}, \\ \Delta r_i^t &= -(f(\mu_t) - f(\mu_t + 0.67\sigma_t))\frac{|T - \mu_T| - 0.67\sigma}{0.67\sigma_T}. \end{aligned} \tag{6.23}$$

The first part of the formulas for $\Delta_r^T$ ($\Delta_r^t$) changes in time very slowly and therefore may be recalculated once per-hour or per-day. Thus, calculations of the functions 6.23 is computationally easier, and therefore it makes the classification process to perform faster.

Analyzing the dynamic of the coefficient of reliability changes, it is possible to derive the answer to the question *how much*. Since the coefficient punishments depend on the security priority of the action, therefore it gives a clue to a possible damage degree. So far the detection system has detected an intrusion and it is even possible to estimate a possible damage degree and separate the intrusion. What can the intrusion detection system do in order to expose a real source of abnormality?

If an abnormal event stream is detected it means that the events of the stream do not belong to the person they are claimed to. In case of a masquerader who performs an intrusion the system may attempt to automatically detect the source of the intrusion. The intrusion detection system tries to match the abnormal event stream against the other profiles of active users and analyzes coefficients of reliability. If at the end it finds a coefficient that is higher than the current one, then the system has suspicions that this insider abuses someone else's network account. In the case that the system has detected an intrusion, it

is reasonable to automatically encode the abnormal event stream to pattern for its usage by a misuse intrusion detection system.

## 6.4 Summary

In this chapter we have proposed a new approach for anomaly detection. It is based on the same theoretical background as a relational matrix approach and aimed to be used for different user categories and under different circumstances. However, a temporal-probabilistic tree may be converted into a relational matrix at any time, therefore, having only a temporal-probabilistic tree it is possible to employ both these approaches.

The temporal-probabilistic approach is aimed at discovering and usage of user behavioral patterns. It stores them in a form of temporal-probabilistic trees, which are adapted to catch temporal aspects of a users' behavior. The main assumption behind this approach is that the behavior of users follows regularities that may be discovered and presented using the temporal-probabilistic trees for the recognition of the user. The approach described here is used as main classification techniques in the HIDSUR and implemented in a *classifier* (see Figure 3.2), which is the most important part of the system since it has to track dynamically a users behavior and differentiate normal behavior changes from abnormal behavior.

In the next chapter we are going to consider different issues related to dealing with behavioral changes. They may be caused by different reasons, such as normal behavior change or attempts of abnormal learning, and should be properly addressed in order to avoid potential false alarms.

# 7 DEALING WITH ANOMALIES IN USER BEHAVIOR

An anomaly in a system may be caused by three different reasons. One of the possibilities is an intrusion. Another two are: concept drift and malicious attempts to train the system's profile(s). The first case was discussed in the previous chapter. Here we concentrate on the other two: what are they, how to differentiate between them, and how to handle them.

If a user (process or network) behavior is to remain static there would not be a concept drift problem. However, in computer systems everything is changing and with time changes occur more often (lifetime of software products is constantly shrinking, networks are becoming faster, etc.). The larger the network the more likely change is occurring somewhere on the network. If anomaly detection is to remain accurate and complete it must be able to learn continuously, and it is this learning process that is the Achilles heel of anomaly detection.

A small data set is incomplete and therefore, can not be used for learning. A large data set will be inaccurate once change occurs. An anomaly intrusion detection system that continues to learn based upon new data exposes itself to learning that attacks are not anomalous.

This problem was researched before in the artificial intelligence area. There are some ad-hoc methods suggested in a supervised learning for dynamic update of a knowledge base (Mitchel, 1997). However, there is no efficient method for unsupervised learning to handle the concept drift (Nilsson, 1998). The reason for this is the difficulty in establishing a feedback to a classifier providing the information about the correctness of provided classifications.

In the anomaly intrusion detection area many works have discussed this problem, however, only suggestions to the possible solutions were provided, for example, (Carpenter *et al.*, 1992), (Lane and Brodley, 1999), (Lee *et al.*, 2000), and (Zhang *et al.*, 2001). Sometimes, a question of how a system's parameters may affect such updates in a knowledge base is raised (Eskin, 2000), (Tan and Maxion, 2002). However, there is no generic method suggested directly for

anomaly intrusion detection to handle the concept drift efficiently differentiating it from malicious learning attempts. Therefore, we feel that this issue has to be addressed since it is the weakest point of the anomaly intrusion detection preventing further development and widespread use of anomaly intrusion detection systems (especially ones that use behavioral models).

A very important problem in information systems security is to prevent undesirable or malicious learning of systems that use online learning to detect the presence of an intruder masquerading as a valid user, or abusive actions of a legitimate user. It not possible to create a universal approach to differentiate the concept drift from the intellectual attack since systems that employ machine learning are unable to automatically establish a strict border between them (Schlimmer, 1987). Therefore, to handle these situations it is usually required to devise an ad-hoc element for the machine learning approach. It varies depending on the learning algorithm involved and the type of data processed since for different algorithms and types of information the line dividing concept drift from intellectual attacks must be established differently. It is mainly defined by the way the information is processed and information context that can not be translated and understood automatically by the learning system.

In this chapter we discuss the above identified problems of the anomaly intrusion detection. We start from discussing how we handle the concept drift and continue to the detection of abnormal learning. On the implementation stage of system development some methods are engaged to detect and, therefore, prevent such undesirable learning. In the HIDSUR system there is a component that is designed to detect such undesirable (malicious) learning - *profile analyzer* (see Figure 3.2). A brief description of its functionality was provided in Chapter 3. In this chapter we discuss methods to detect attempts to train the system and prevent the system from learning such undesirable actions.

Additionally, in this chapter we introduce profile evaluation criteria that help to expose the already trained profiles (i.e. profiles that have been trained in a way to represent a much more general model of behavior). Intellectual attack detection methods and the profile evaluation criteria are based on the trustworthiness technique described in this section.

## 7.1 Monitoring Natural Behavior Changes

User behavior during his interaction with a computer is not constant. He may gradually change his behavior weekly or monthly due to many reasons. He/she might learn new features of the software; he/she might begin to use other software (or another version of the same software); the goals that the user tries to achieve might be changing, etc. Therefore, the time that the user spends on performing his/her tasks may also change. These gradual behavior changes are referred to as *concept drift*. The tree should be periodically updated in order to take into account the natural behavior changes; otherwise a rate of false alarms

would be dramatically increased. Therefore, the tree upgrade includes the addition of new patterns and, therefore, branches in the tree, and the deletion the outdated patterns, that the user does not follow any more. To reach these goals the constant recalculation of the following parameters is required:

- the value $n_{act}$ of the instances and the value $n$ for the patterns;
- the mean $\mu_T$ and deviation $\sigma_T$ of the action class instances; and
- the mean $\mu_t$ and deviation $\sigma_t$ of the relation class instances.

Consider the cases when certain changes in the tree structure are needed:

1. The amount of "free" actions $n_{act}$ of an instance becomes bigger than $n_{min}$.
2. The usage frequency $n$ of a pattern becomes less than $n_{min}$.
3. The deviation $\sigma_T$ of an action class instance becomes bigger than the threshold value $\sigma_{T_{max}}$.
4. The deviation $\sigma_t$ of a relation class instance (current transition) becomes bigger than the threshold value $\sigma_{t_{max}}$.
5. The probability of a node usage becomes smaller than the threshold level $p_{min}$.

Each of these cases is processed separately. Below we consider every one of them.

### 7.1.1 Learning User Normal Behavior Changes

One of the very important abilities is to learn online user behavior changes in order to avoid false alarms. This kind of learning implies online addition of new patterns and the removal of old ones that do not accurately describe user behavior any more. To perform this, certain criterion should be established. By criteria we imply the setting of certain limits for parameters $n_{act}$ (amount of "free actions") of instances, $n$ (usage frequency) of patterns and $p$ (usage probability) of nodes should be checked. The parameters $n_{act}$ of instances and $n$ of patterns are dynamically updated as follows.

During every classification step, a new action issued by a user is assumed, firstly, to an action class and, secondly, to a certain instance of this class. Knowing the owner instance, the IDS finds a corresponding node for this action. If the action does not match any node it means that the instance that includes this action does not belong to the expected node, nor to any of the first nodes of other patterns. In this case the value $n_{act}$ of the instance, which this action belongs to, is increased by "1".

Outdated actions from the input stream are deleted on a daily (or weekly) basis. For current data (day or week), all actions that occurred earlier than a certain time point are considered as outdated and are not taken into account any more. The usage frequency $n$ of the patterns, that include such actions, decreases, as well as the amount of "free" actions. The value of $l_{node}$ is chosen experimentally and should be big enough so that the length of the history would be long enough to be able to statistically find sequential and temporal patterns in it.

If during classification the usage frequency $n$ of a pattern $M_i$ becomes less than $n_{min}$, this pattern is considered as outdated. The nodes of the tree, that represent this pattern, are deleted. For the instances of pattern nodes the value $n_{act}$ of "free" actions is incremented by $n$.

If during an update step the value $n_{act}$ of an instance appears bigger than $n_{min}$, then a new pattern is created and represented by the newly created node. After that, the system attempts to concatenate the new pattern with the older ones as described in the previous section.

Each time the classification process has passed a node, the parameters $\mu$ and $\sigma$, for the action instance of this node and for its incoming transition, are recalculated. If for an instance $I^A$ $\sigma_T > \sigma_{Tmax}$ it means that the current instance $I^A$ should be checked if it contains more than one instance. The check is performed by an attempt to split the instance into two instances $I'$ and $I''$ as shown in Figure 7.1. A method used to search for instances is applied on the current instance to determine the possibilities of splitting it.

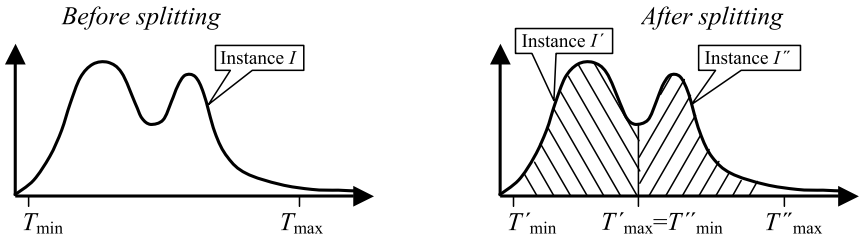


Figure 7.1 Simple example of instance splitting

If it is not possible to split the instance $I_i^A$ it is considered as useless for classification and the intrusion detection system does not use it any more.

In the case of deviation $\sigma_t$ of the edge between $G_i^k$ and the node $G_i^{k+1}$, becomes bigger than the threshold level $\sigma_{tmax}$ ($\sigma_t > \sigma_{tmax}$), this edge should be pruned.

If either of the conditions above is met, the tree is rebuilt. After conversion a new tree appears in which for all instances the parameters $\sigma_T$ and $\sigma_t$ are lower than their possible maximum values.

### 7.1.2 Dealing with Concept Drift

As was already mentioned, in real life user behavior is not constant during the time he/she performs his/her activities. For example, when a user begins to use a program, he/she may not know its options and abilities. He/she usually makes many mistakes. After some time, with experience, the user's mastery is growing, the level of mistakes becomes lower, and program abilities are used in a more optimal way. Thus, temporal characteristics are constantly changing. Methods that do not consider such smooth behavior changes may produce many false alarms. Below we suggest two ways to deal with these possible problems.

The first way is to limit the action lengths' history based on which the temporal-probabilistic characteristics of nodes and edges are calculated. The user behavior may not be constant during long periods of time, due to many reasons. Correspondingly, the model of normal behavior is also changing. Outdated temporal data, still being used for classification, would disfigure the result. Therefore, it is necessary to limit $n$ for node and edge by some value - $l_{node}$ and $l_{edge}$ correspondingly. The value of $l_{node}$ ($l_{edge}$) depends on how fast the $n$ is growing. In other words how often a certain node (edge) is used for behavior classification. The faster $n$ is growing the bigger the value $l_{node}$ ($l_{edge}$) should be. This should be taken into account in the calculation process. This can be done, for example, as follows:

$$\mu_{T_i} = \left( \sum_{j=i-l_{node}+1}^{i} T_j \right) / l_{node} \tag{7.1}$$

where $i$ is $i^{th}$ hit of the node; $l_{node}$ is the volume of time length pool; and $T_j$ is the time of action when this node was hit $j^{th}$ time.

$$\sigma_{T_i} = \sqrt{\left( \sum_{j=i-l_{node}+1}^{i} (\mu_{T_i} - T_j) \right)^2 / l_{node}} \tag{7.2}$$

Each time a node or edge is used for classification its temporal-probabilistic characteristics may be recalculated as:

$$\mu_{T_i} = \frac{\mu_{T_{i-1}} \times l_{node} - T_{i-l_{node}} + T_i}{l_{node}} \tag{7.3}$$

$$\sigma_{T_i} = \sqrt{\frac{\sigma^2_{T_{i-1}} \times l_{node} - (\mu_{i-l_{node}} - T_{i-l_{node}})^2 + (\mu_i - T_i)^2}{l_{node}}} \tag{7.4}$$

It is necessary to note that to calculate the standard deviation, the system uses the mean, that is being calculated, using the previous history of mean values. Introducing the limit for the length of this history gives the system the opportunity to estimate and control the usage of computer resources that are involved in storing and processing this history for every node and edge.

By analogy with calculation of temporal parameters for action instances contained by nodes (Equations 7.3 and 7.4), the parameters of relation instances (in edges) are calculated:

$$\mu_{t_i} = \frac{\mu_{t_{i-1}} \times l_{edge} - t_{i-l_{edge}} + t_i}{l_{edge}} \tag{7.5}$$

$$\sigma_{t_i} = \sqrt{\frac{(\sigma^2_{t_{i-1}} \times l_{edge} - (\mu_{i-l_{edge}} - t_{i-l_{edge}})^2 + (\mu_i - t_i)^2}{l_{edge}}} \tag{7.6}$$

The second possibility of how to manage normal behavior changes is to control smooth changes of the mean, when calculating a change value for the coefficient of reliability.

In Figure 7.2 it is possible to see a curve that outlines the dynamic of the mean changes during the concept drift. Since in point three, for recalculation of the mean instead of all the previous history only the information about $l_{node}$ last hits is used, the action length that is considered as normal would be different from the mean calculated in this point.

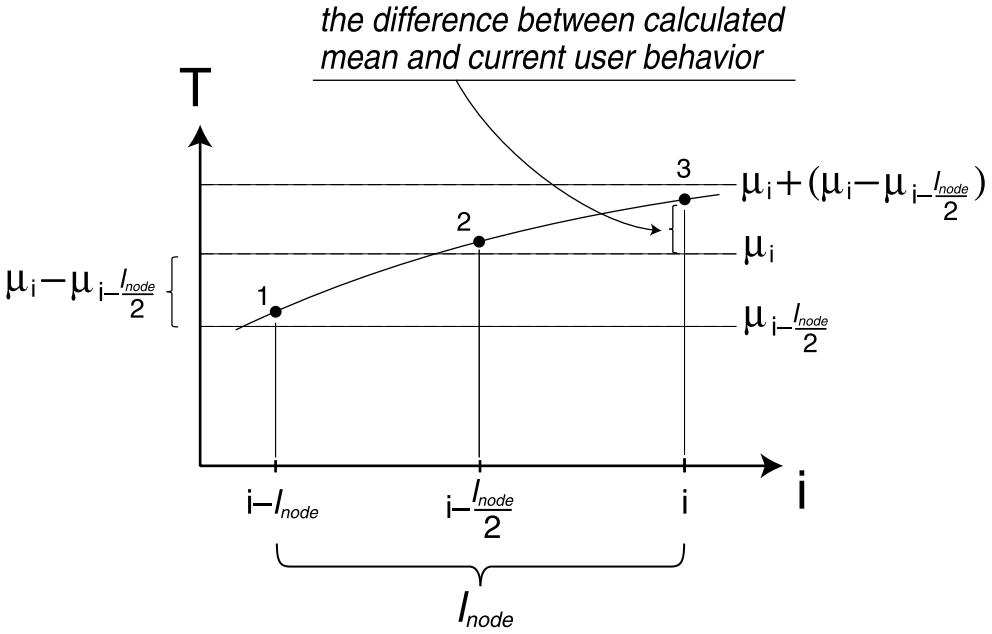


Figure 7.2 Dynamic of mean change

To lower this error we suggest the use of a *corrected* mean. Coefficients $w_i = (0, 1)$ allow to limit the mean correction for the actions with high coefficients of security significance:

$$w_i = 1 - \frac{\nu_{A_i}}{\nu_{max}} \tag{7.7}$$

When $w_i \to 0$ then $\mu' = \mu$ and correction is not performed. Then the mean is calculated as:

$$\mu'_{T_i} = \mu_{T_i} + w_i \times \left[ \mu_{T_i} - \mu_{T_{\frac{i-l_{node}}{2}}} \right] \tag{7.8}$$

$$\mu'_{t_i} = \mu_{t_i} + \left[ \mu_{t_i} - \mu_{t_{\frac{i-l_{edge}}{2}}} \right] = 2 \times \mu_t - \mu_{t_{\frac{i-l_{edge}}{2}}} \tag{7.9}$$

By analogy, it is possible to monitor smooth changes of a standard deviation. In this case the history of standard deviations used is limited by $l_{node}$ ($l_{edge}$). In this case the corrected standard deviations are:

$$\sigma'_{T_i} = \sigma_{T_i} + w_i \times \left[ \sigma_{T_i} - \sigma_{T_{\frac{i-l_{node}}{2}}} \right] \tag{7.10}$$

$$\sigma'_{t_i} = \sigma_{t_i} + \left[ \sigma_{t_i} - \sigma_{t_{\frac{i-l_{edge}}{2}}} \right] = 2 \times \sigma_t - \sigma_{t_{\frac{i-l_{edge}}{2}}} \tag{7.11}$$

Therefore, calculating the coefficient of reliability change, previously corrected mean and standard deviation are used. The usage of corrected means and standard deviations lets the system escape "inertia" while recalculating the temporal parameters of distributions and the coefficient of reliability change. Prediction of values of the means and standard deviations provides this possibility. This property of prediction makes the change of the coefficient of reliability to grow, if user behavior change inverts its direction (for example, the average time length of the action used to grow, but now it decreases). At the same time during a certain time interval it is predicted the mean to grow, but it begins to diminish. This brings the system's attention to this point, strengthening its ability to detect intrusions.

Finally, at the end of this section we would like to say a few words about checking for information consistency. This question was discussed in Chapter 4 and here we show that the checking for information consistency is a part of the internal structure of the tree. If there is an action X in progress (system received a starting point and expects action's ending point) it means that there is a correspondent node in the tree on level $k$. There are a number of nodes on level k+1 that are connected with the current node and one of them may be chosen as the next node. If a transition to any of these nodes starts it means that action X is finished. Every transition is supposed to happen at a certain time. If this time has passed then this transition is not possible. If all times of all possible transitions, between the node corresponding the action X and the connected by them nodes at the level k+1, have passed it means that the system has not received the action's endpoint and, thus, there exists an inconsistency.

## 7.2 Detecting the Abnormal Learning

In this section we introduce the requirements and evaluation criteria for user profiles in an anomaly detection system and techniques for their evaluation in terms of these requirements. We base these techniques on the approach described in Chapter 6.

The goal in the intrusion detection task is to identify malicious occurrences while falsely flagging innocuous actions as rarely as possible. This task also includes the detection of intelligent attacks. For these purposes we have developed a special component - *profile analyzer*. In Figure 7.3 we can see the part of a hybrid intrusion detection system architecture where the profile analyzer is incorporated.

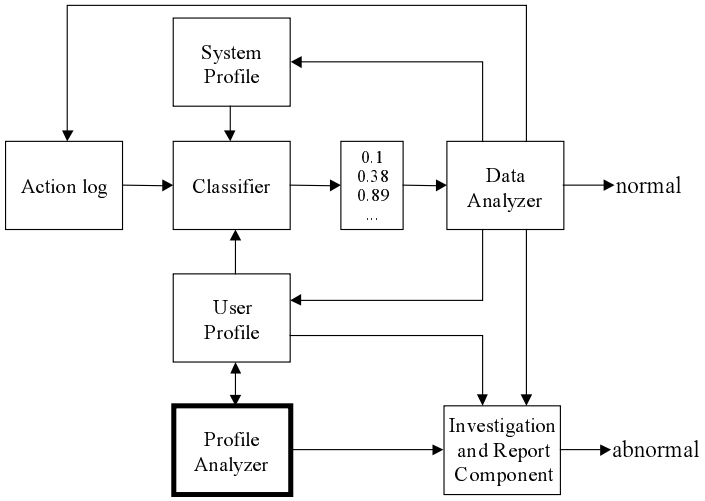


Figure 7.3 Part of HIDSUR with embedded profile analyzer to aimed at the detection of intelligent attacks

The main goal of the profile analyzer is to detect *intellectual attacks*, which are not possible to detect in real-time. It works asynchronously from other components of the intrusion detection system. When the system has spare CPU time (for example, nighttime) it is activated and it analyzes every user profile observing all changes in the profile over some period of time. This period should not be too short since during a short period of time there can not be many changes in the profiles. Therefore, frequent checks would be a waste of computer resources. On the other hand, the interval between checks should not be too long, because in this case time-to-detection would also be too long. Additionally, it would be impossible to detect when the attack happened. Taking all said before into account, it would be logical to set the length of this time interval to a length of the training period of a used classifier. During this period of time there are some natural changes in the user behavior as expected, but it is not too long (a detailed discussion about the length of the learning period is presented in Chapter 9).

Making tree traversal the profile analyzer calculates the coefficients of reliability for every node in the tree and then for the whole tree by putting mean values of temporal parameters into formulas. The change of the coefficient of reliability on each step has an inverse dependency on the value of the standard deviation. It means that if there is a trend between standard deviations to grow, the overall coefficient of reliability will decrease with time. It shows that there is a problem with some particular profile and that may be the result of an intellectual attack or incorrect choice of thresholds. Either case requires further investigation and some action to perform. In case of the intellectual attack, the analyzer reports to the investigation and report component, which reports further to a system administrator.

In case of an incorrect choice of parameters the analyzer performs manipulations on a tree in order to find the correct values of the parameters. It splits general action patterns into several more detailed ones, etc. It also may vary the parameters $l$, $c$ and $\nu_A$ to manage the tree.

### 7.2.1 Profile Evaluation Criteria

In this section we propose criteria for user profile evaluation. Also we describe algorithms for the detection of intellectual attacks. These algorithms are based on continuous calculations and the monitoring of the profile parameters, i.e. observing dynamics of parameters' changes.

As it is possible to see in Figure 7.3, our HIDSUR architecture has a component called *profile analyzer*. It is intended to analyze user profiles and maintain their integrity. The profile analyzer is activated after a certain time interval and(or) when it detects the availability of spare CPU time. Then it checks all the user profiles for satisfaction of the following criteria.

One of the profiles checking criteria is matching someone's profile against a stream of events created by another user or randomly generated. If the calculated coefficient of reliability begins to grow with time the system has a suspicion that something goes wrong and it requires an additional and more detailed check. Growth of the coefficient of reliability shows that the system tends to accept a more general model of behavior for a certain user, instead of recognizing personal features of the user's behavior and differentiating him from other network performers.

There are two reasons that may affect the coefficient of reliability in that way. The first one is that some of the coefficients $l$, $c$ and $\nu_A$ (parameters of the profile) have not been properly chosen. In this case the coefficient of reliability will increase for every profile in the system. Therefore, the system administrator should reconsider the constants and revise the profile of this user according to the new ones.

The second reason is an undergoing intellectual attack. If the coefficient of reliability only increases for a certain profile this may mean that this profile is under attack. Thus, it requires further detailed investigation. In order not to

produce a false alarm the system employs several levels of analysis. If there is suspicion of an intellectual attack the system has two choices. One of them is to search the whole particular tree and separate the new branch(s) that provoke the overall growth of the coefficient of reliability and then analyze the security significance coefficients of each action $\nu_A$ for the separated branch. Based on this, the profile analyzer is able to estimate the rate of usage of privileged commands in the analyzed stream. If the rate is high this means that the user trains the system to accept his malicious actions for future abuse. In case of a low rate, the system may manipulate the security significance coefficients. Since these coefficients are included in the coefficient of reliability calculation, this manipulation will allow the system to pay more attention to the usage of the analyzed actions in the future.

There are other possible scenarios of meta-attack detection. For these purposes the profile analyzer also controls tree growth, in order to detect sudden tree augmentation, and determine reasons. There are two possible cases:

- Concept drift (Schlimmer, 1987). A user has begun to use new programs. In this case it is possible to differentiate by appearance of new nodes with new types of events that have not been typical of this user before. After this it is expected that the tree diminishes due to the removal of actions and branches that do not reflect the user's behavior any more (when the user begins to use new versions or new programs in most cases he/she stops using some program or performing some tasks).

- The user has begun to use the applications he/she has been using before, but in a new and unpeculiar way for this user. Therefore, it requires detailed investigation (for example, the system may manipulate the coefficients of security significance as described above or, as described below, analyze the standard deviation of time distributions).

In contrast to tree growth the intellectual attack may also be detected by monitoring the changes of the standard deviations of the tree. If the standard deviations $\sigma_t$ and(or) $\sigma_T$ grow, it means that the system will classify, for this action, more and more relations as normal relatively to any other action. Thus, the profile analyzer, making the tree traversal, calculates the coefficient of reliability for the whole tree according to:

$$R = \sum_{\{G\}} \frac{1}{\nu_{A_i} \times \sigma_T^2} + \sum_{\{E\}} \frac{1}{\sigma_t^2} \tag{7.12}$$

It is possible to see from this formula, that the bigger the standard deviation is, the less we trust the action or relation, if the user behavior tracing process goes through this node or uses this relation. Also, it depends on action's security

significance coefficient. Frequent and unrelated, to a certain very strict sequence of actions, usage of privileged commands should flag a system's suspicions.

A certain threshold is established for the tree. If after calculations the coefficient of reliability is smaller than a predefined threshold value then the profile analyzer applies *pattern splitting* on the tree. It searches for complicated patterns with a wide deviation interval and splits it into several simple ones with narrow $\sigma$. An example of such splitting can be seen in Figure 7.4.

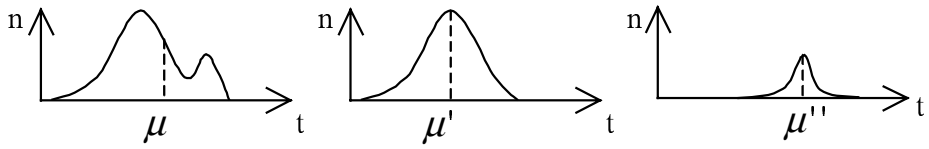


Figure 7.4 An example of splitting an action instance

In this section we have suggested methods for the intellectual attacks detection, which is based on the usage of artificial intelligence features (such as online learning) in the anomaly intrusion detection area. Our methods are based on the algorithm for user behavior verification described in Chapter 6.

## 7.3 Summary

An anomaly intrusion detection system that employs static profiles has to rebuild them often due to their short life cycle. It leads to frequent interruptions of the system's functionality, additional manual labour, etc. At the same time computer systems become faster and more powerful sending more and more information over a network continuously increasing the network's load. Therefore, intrusion detection systems with static profiles cannot be efficiently used in a modern network environment. On the other hand, a usage of the dynamic profiles introduces additional problems that need to be solved.

This chapter discusses potential solutions to problems caused by the usage of dynamic profiles in anomaly intrusion detection. These problems may be solved if a system may differentiate changes in a user's behavior from malicious attempts to train a system. We proposed solutions showing how to handle the concept drift and detect abnormal learning. There is a part in the HIDSUR architecture that is devoted to detection of such an undesirable learning. The profile evaluation criteria introduced here are aimed at exposing the already trained profiles (i.e. profiles that have been trained in a way to represent a much more general model of behavior). Intellectual attack detection methods and the profile evaluation criteria are based on the trustworthiness technique described in this chapter.

# 8 OVERVIEW OF THE IMPLEMENTATION ARCHITECTURE

This chapter discusses an architecture of a prototype we built basing it on the HIDSUR model described in Chapter 3. During the implementation of the prototype we have encountered numerous problems that we have had to solve. Below we identify some of them:

1. Every system has a unique configuration of software that makes it almost impossible to have a consistent test environment.

2. System behavior may depend on the software patches installed. Often that software generates intrusion alerts simply because it is poorly designed (e.g. writing logs into read-only storage).

3. There is no comprehensive definition of what a ”system load” is, therefore it is necessary to define it for our own case. Moreover, our understanding of what the ”system load” is may slightly differ from those found in other works, and thus, it makes it difficult to compare performance results obtained with other IDSs (Durst *et al.*, 1999).

4. One of the important requirements of an intrusion detection system is a minimum performance impact, but nowhere is defined what does the ”performance” mean. We observed that in relevant research the researchers sometimes take the CPU load or time-to-alarm as a main performance measure (Lane and Brodley, 1999), (Wee, 1998). It even happens in commercial intrusion detection systems (Weinberg, 2001).

   If we consider the minimal performance impact as a requirement to an intrusion detection system not to disturb the normal operation of the system, it is not possible to provide a general definition of a normal operation. Usually a security policy defines the requirements of acceptable usage of the computer resources and establishes the correct procedures

for their usage, i.e. normal operation of the system. However, a policy also defines security priorities, since different organizations have different security concerns and priorities, which their policies must properly reflect. These organizations would have different priorities, which in turn imply different definitions of a system's normal operation in these organizations.

5. Different people have a different measure of effectiveness: some want to know about new attacks launched against their systems, some want to know about a pre-configured set of attacks, others just want to see no attack warnings. Therefore, it is necessary to use different approaches to establish thresholds for classification.

6. Time delays in reporting alerts are often very dependent on the particular system configuration: if there are checks for race conditions, it is probably necessary to watch *stat()* family of system calls. If a user has many applications that do a lot of *stat* calls (like *top*) then the system will quickly fall behind processing the data.

These issues make it very difficult to evaluate an intrusion detection system (Puketza *et al.*, 1996). During our design we had to properly address them. We defined the "system load" and "performance" for our case in a way that would allow us to compare our results with the evaluation results of other intrusion detection systems. The tests described below are aimed at showing the possibility of usage and possible benefits (listed in Section 1.4) of discovering and employing temporal behavior regularities for user recognition. The prototype has been developed to serve as proof of concept of the implementation of the model.

## 8.1 Architecture of the Prototype

In this section we describe our design choices during a prototype implementation. There were following design problems that had to be solved during the implementation process:

1. The external representation of information that describes the dynamics of user behavior (event→action→activity): how to exchange this information between computers, and intrusion detection system components or anomaly detector's modules.

2. The representation of the behavior regularities (temporal-probabilistic tree): what is the best way to store and process them in user profiles.

3. The interface to the user information source. In our implementation we had to design a *host agent* in order to access user events on the operating system level.

4. Events' management: delivering the necessary events to a corresponding classification process. On this stage it is necessary to ensure that the classification processes obtain all necessary information without losses, in order to perform online user behavior tracking.

5. Parameters' selection: how to select the correct classification thresholds to minimize the amount of possible misclassifications.

In the following sections we are going to discuss these issues. In addition to dealing with those issues our implementation was designed taking into account the following:

- The ability to manage autonomous user classification processes by automatically creating and destroying them.
- The ability to distribute tasks over a controlled network to improve the overall performance.
- The ability to handle multiple event streams by the same classification process.
- The ability to locally filter unnecessary information without sending it over a network in order to minimize network resource usage.

Below we describe some important issues we were dealing with during the implementation and testing stage.

### 8.1.1 Application Architecture

The software we have implemented to support the classification process contains numerous classes, each providing a different functionality. Some part of it had to be implemented on an operating system level, another on an application level. Below we will concentrate on some important design details. Figure 8.1 depicts the main components of the implemented prototype.

As was discussed in Chapter 3 the architecture consists of three main components, which interact with each other: detection server, control center, and host agents.

The control center was implemented as a module on a system administrator's workstation. This module allows the control of all HIDSUR operations and the collection of statistics for experiments. The control center of the prototype is responsible for:

- starting and stopping host agents,
- polling host agents,
- enabling and disabling the detection server,

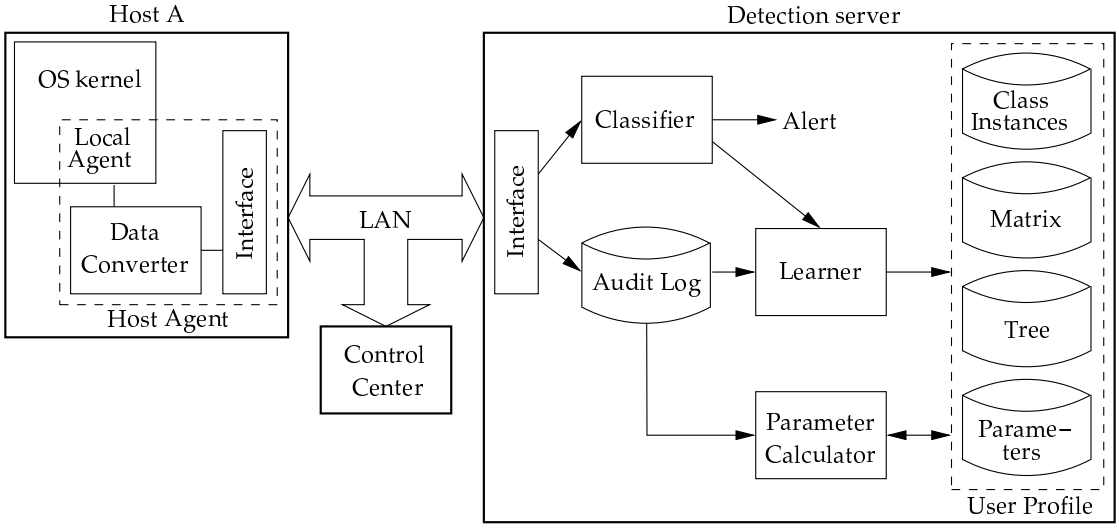


Figure 8.1 Prototype architecture

- synchronization of the host agents and the detection server,
- manipulation of the classification parameters (for experiments), and
- collection of statistics.

All these operations are performed by exchanging control signals between the control center and the different modules of the HIDSUR. Additionally, to collect the network statistics the control center passively observes network traffic between HIDSUR components by setting a network card into a promiscuous mode.

In our prototype we have used host agents compiled of several *nix type of operating systems. They were implemented as loadable kernel modules. When an operating system starts on a local workstation the host agent is loaded into the workstation's memory. Then it reports to the control center that it is loaded and goes into standby mode waiting for activation and synchronization from the control center. A detailed description of the algorithm and signals that the host agent exchange with an operating system's kernel may be found in Section 8.2.

There is a devoted server in our prototype assigned for anomaly detection. All necessary definitions and methods implemented in the prototype were provided in Chapter 5 and 6. The algorithms employed in the classifier (for both classification methods) can also be found there. The process of learning and classification are described in the following sections.

As we have mentioned at the beginning of this chapter, this prototype was implemented as a concept proof, therefore we are not going to consider the issues here that are not directly related to the acquisition of the performance results of our methods.

### 8.1.2 Client-Server Information Exchange

The external representation of actions (problem 1) is designed as a straightforward representation syntax that directly reflects their structure and provides user id, computer id, class name, and the start and end time of each action. These specifications are stored in a file and maintained as encrypted strings. Events are collected by a local agent, transformed into actions by a data converter, prepared for sending and encrypted by a network interface.

In Figure 8.2 an information flow in the prototype is depicted. As can be seen the information comes from different sources - workstations. A single user may be logged into a single workstation, or several remotely/locally-logged users may share it. All collected information is assembled into a single stream of events and comes to a detection server through an encrypted data channel, which protects sensitive information about user behavior from outside eavesdropping.

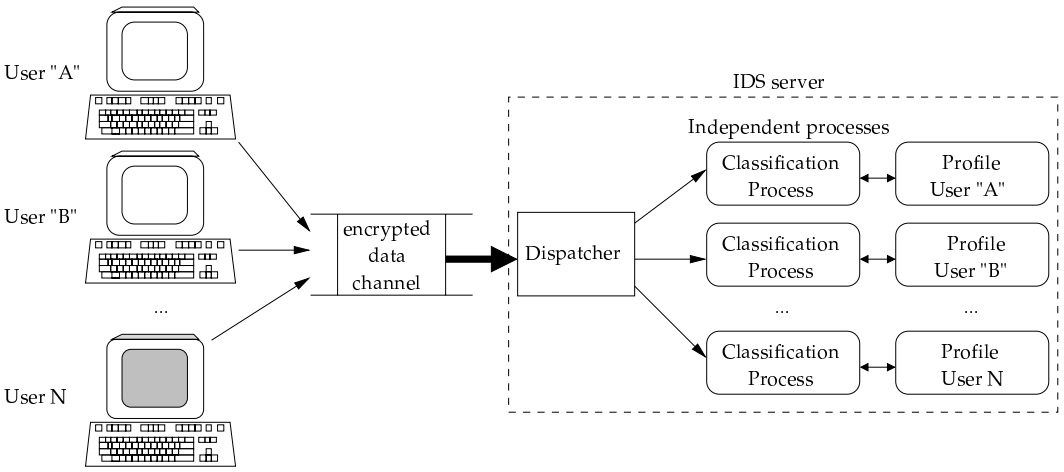


Figure 8.2 Data flow inside the prototype during online classification

The IDS server instantiates an independent application server (classification process) for every active user. Every process encapsulates the necessary information about the normal behavior of a single particular user (it is taken from a user profile). All information, received by the IDS server, is decrypted and dispatched to different application servers, each of which tracks and analyzes the behavior of a specified user as described in Chapter 6.

## 8.2 Host Agent

In this work we had to implement our own logging facility to collect the necessary information about user behavior. Therefore, in this section we concentrate on the operations of the host agent that was implemented for *nix type operating systems in our prototype. This is the only part of the HIDSUR architecture that contains an operating system dependent code. This section gives the idea

of how the host agent may be implemented for other types of operating systems (for example, Windows). The overall structure remains the same, only messages and functions that the host agent uses to interact with the kernel are different.

A description of a host agent's architecture and its interaction with *nix kernel is provided here. The aim of this small program is to collect detailed information about user behavior[1].

### 8.2.1 Host Agent Operations

The host agent collects information about active system processes, GUI messages, and monitors the modifications of a file system. To minimize resource consumption and to make it impossible to alter the host agent is implemented as a Unix daemon process with *suid* bit set.

Figure 8.3 demonstrates the main algorithm of the host agent.

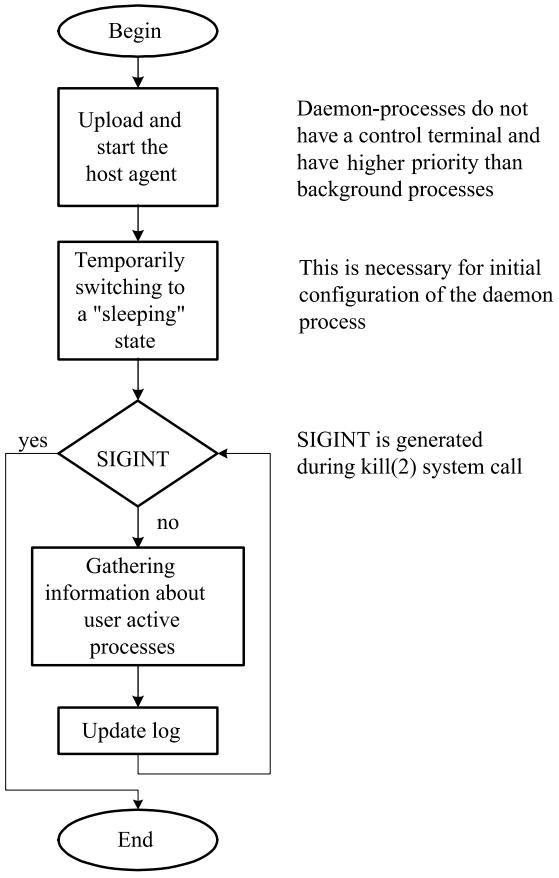


Figure 8.3 Main algorithm

When a workstation's operating system is being started the *initiator* program is executed for the workstation. This program creates an environment for downloading the host agent to this workstation. At the beginning the initiator determines the type of operating system and checks the system's environment. After

[1] See Chapter 3 for details.

this it establishes an encrypted connection with the detection server and uploads the host agent's code for the current type of operating system.

The host agent is initialized as a "daemon" process without a control terminal (it is controlled from a control center through the local area network). Additionally, the priority of this process is set higher than the system's background and user processes. The details of the host agent initialization are depicted in Figure 8.4.

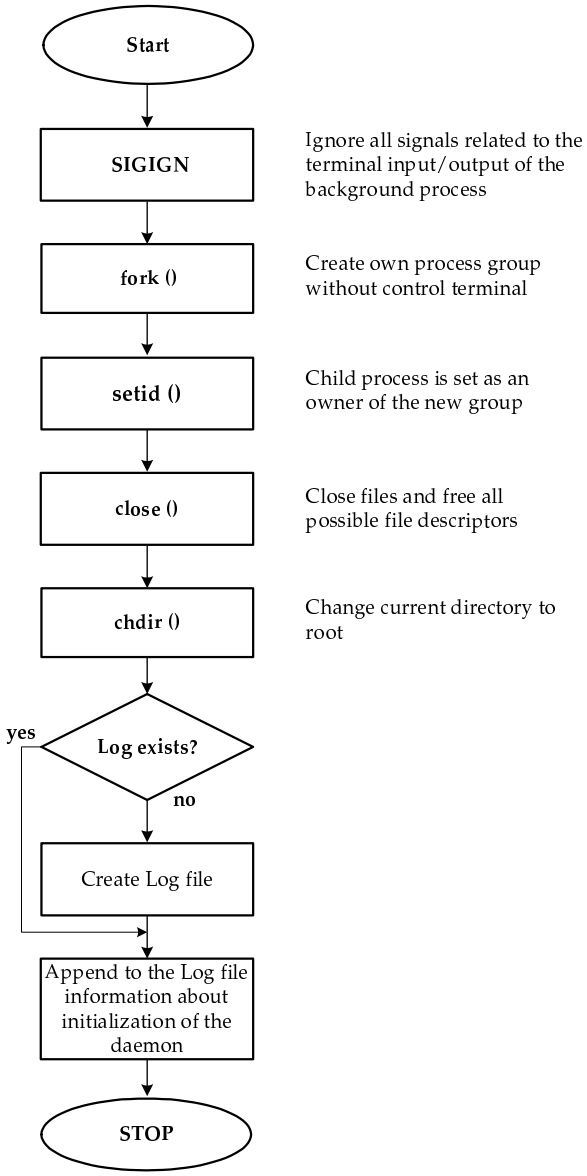


Figure 8.4 System daemon

At the beginning the host agent needs to filter all unnecessary signals. The filter is set up by a *SIGIGN* signal[2]. After this the kernel permanently stops dispatch-

[2] Signals and system calls used by the host agent are described in Appendix 2.

ing messages related to the terminal input/output of the background processes to the host agent.

After this the host agent creates its own process group without a control terminal (*fork()*) and sets itself as an owner of this process group (*setid()*). The workstation's resources that are not required for a further agent's operations are freed by *close()* command and the current directory is changed (*chdir()*) to a one where log file should be located.

If the log file exists at the given location it is opened, if not - the host agent creates an empty one. At the end of the initialization process the information about the successful (or unsuccessful) start of the host agent is appended to the log file.

After the initialization the host agent goes inactive - "sleep" mode (Figure 8.3). At the beginning it is done to allow the rest of system processes to be launched and activated (their amount and sequence of loading are determined by a system's configuration) and a user to log in.

The inactive or "sleep" mode (Figure 8.5) is achieved by setting a timer (*setitimer()*) into an infinite loop and waking up by an *alarm()* when a message related to user activities comes from the kernel.

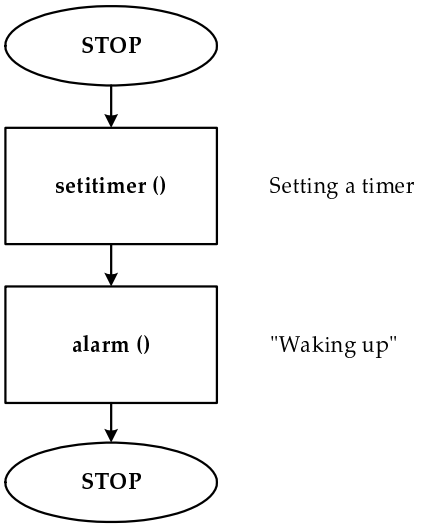


Figure 8.5 Inactive mode

If it is necessary to disable the host agent on this workstation the control center initiates a *kill(2)* signal. When the host agent receives this signal it aborts its operations and exits.

If the termination signal is not received the host agent performs its normal duties: collects information about user behavior and puts it in the log file. From the log file the information goes to the data converter, preprocessed (see Chapter 4) and transmitted over a network to the detection center.

### 8.2.2 Information Collection

Figure 8.6 provides a step by step description of the user data collection process. These steps of the process are performed each time the kernel sends a message to a host agent. Changes in the structure of the system processes are not possible without a message exchange between kernel's modules, therefore the host agent is notified each time something happens in the process structure. However, the host agent is not going to listen and process all kernel messages because it would require additional CPU time. Thus, as we have mentioned in the previous section the kernel is informed to only send to the host agent messages related to the changes in the structure of processes.

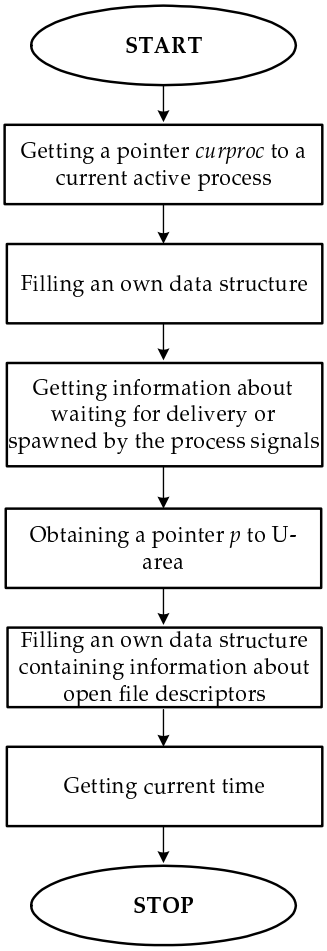


Figure 8.6 Gathering information about an active user process

The information collection process depicted in Figure 8.6 is performed each time a change happened in the structure of the processes: start process, end process, activate/suspend process, change priority of the process, etc. Initially the host agent obtains a pointer *curproc*, which points to the current user active process. The *proc* (Table 8.1) structure is a record of the system's process table. A record for a current running process is pointed by a system variable *curproc*.

Every process in a *nix operating system is described by two data structures - *proc* and *user*. They are described in header files of the system's kernel source <*sys/proc.h*> and <*sys/user.h*>.

Table 8.1 Structure of *proc*

| Name | Type | Description |
|---|---|---|
| *p_stat* | char | process state |
| *p_pri* | char | current priority of the process |
| *p_flag* | unsigned int | flags defining process's auxiliary information |
| *p_uid* | unsigned short | *UID* of the process |
| *p_suid* | unsigned short | *EUID* of the process |
| *p_sid* | int | session identifier |
| *p_pgrp* | short | identifier of a process group |
| *p_pid* | short | process identifier (PID) |
| *p_ppid* | short | parent process identifier (PPID) |
| *p_sig* | sigset_t | signals waiting for delivery |
| p_size | unsigned int | size of an address space in pages |
| *p_utime* | time_t | time of execution in a task mode |
| *p_stime* | time_t | time of execution in a kernel mode |
| *p_ldt* | caddr_t | pointer to a process's LDT |
| **p_region* | struct pregion | list of CPU memory areas |
| *p_xstat* | short | return code that is given to a parent process |
| *p_utbl()* | unsigned int | array of pages records |

Information about user processes is obtained from the Unix file system */proc* where each directory name represents a process number (PID). In Table 8.2 we identify files of the */proc* file system used for gathering information about user behavior.

Table 8.2 */proc* File system files used for obtaining user behavior statistics

| File name | Description |
|---|---|
| *cmdline* | name of executable file with arguments |
| *environ* | variables of the process's environment |
| *stat* | process execution statistic |
| *statm* | statistics of the virtual address space |
| *status* | important characteristics of the process |

After this, the host agent analyses and stores the obtained information into its own structures. When the initial information is obtained the host agent looks for and analyses messages to the current process or issued by this process.

After getting all necessary information about the current process the information about the current user is analyzed. There is the possibility that there are multiple users logged into the same workstation. Therefore, the information describing the current user is obtained from the system structure - *user*, also called

*u-area* (Table 8.3), which contains auxiliary process information related to a certain user and it is needed by a system kernel only during the process execution.

Table 8.3 A *user* structure

| Name | Type | Description |
|---|---|---|
| *u_comm[32]* | char | executable file header |
| *signal* | long int | signal disposition |
| *regs* | struct user_regs_struct | hardware context |
| *start_code* | unsigned long | beginning of the code segment |
| *start_stack* | unsigned long | beginning of the stack segment |

The last step, before the obtained information is appended to the log file, is the forming and adding of a time stamp to it. It is not a trivial task. The workstations around the local area network may have time differences with the detection server. Moreover, a malicious user may change workstation's time in order to fool the anomaly detection system. The system described in this work relies not only on an events' sequential information, but also on the temporal. That is the reason why knowledge of the exact time when an event occurred is crucial for the classifier. Otherwise, there would be inconsistencies in the user data and the classifier would not be able to accurately classify the events producing an overwhelming amount of errors.

In our implementation this problem is solved by synchronization with the control center during the initialization of the host agent. When the host agent creates a log file it obtains time from the control center. The log file creation time is set to the absolute current time of the control center. After this the host agent creates its own timer and all events that it puts into the log file have time stamps relative to the absolute time of the log file's creation. This mechanism of time stamps ensures the correctness of time intervals between events even if a user changes workstations time during his interactions with a workstation.

As it is possible to see from the description of the host agent we have implemented an efficient mechanism that is able to provide all necessary detailed enough information about user behavior.

## 8.3 Learning the Classifier

This section describes the implementation of the classifier in our prototype. It shows the details of realization and defines the fields of the temporal-probabilistic tree.

### 8.3.1 Data Model Used for our Approach

For each user his/her personal profile contains a temporal-probabilistic tree $S(G, E)$ that is described as a linked list of patterns and contains a coefficient of reliability for a user whose behavior the tree represents:

$$S :< M_0, n_{ptrn}, r >, \tag{8.1}$$

where $M_0$ is the first pattern of the pattern set - it defines the top level of the tree. $n_{ptrn}$ is the number of patterns that form this tree, and $r$ is the coefficient of reliability, which is used to monitor deviations of the current user's behavior from its usual (normal) behavior.

A single *pattern* $M_i$ is described by a set of nodes:

$$M_i :< G_0, G_{end}, l, n, M_{i+1} >, \tag{8.2}$$

where $G_0, G_{end}$ are the first and the last node of the pattern $M_i$; $l$ is the length of the pattern expressed by a number of nodes in it; $n$ is the number of times this pattern was used (number of cases that support this pattern); and $M_{i+1}$ is a pointer to the next pattern in the pattern set.

Each *node* $G_i^k$ represents a certain instance of an action class and it is described as:

$$G_i^k :< I^A, n, G_{prev}, E_0 >, \tag{8.3}$$

where $I^A$ is the instance of the action class represented by this node; $n$ is the number of cases this node contains (number of actions in the instance represented by this node); $G_{prev}$ is the previous node on the level $G_i^{k-1}$ connected to the current node; and $E_0$ is the first left transition, among the set of transitions, that come out of $G^k$.

Each action or relation class instance $I^A$ is defined as follows:

$$I^A :< A, T_{min}, T_{max}, n_{act}, \mu, \sigma >, \tag{8.4}$$

where $A$ identifies class where this instance belongs to; $[T_{min}, T_{max}]$ is a temporal interval, that is used to define, whether the action belongs to this instance or not. Hence, if for some action the following holds: $T \in [T_{min}, T_{max}]$, then it means that this action belongs to the current instance $I^A$. $n_{act}$ is the number of "free" actions forming this instance that do not belong to any pattern yet; and $\mu, \sigma$ are the parameters of temporal lengths' distribution.

Finally, actions are included in a set that forms an instance if:

- they represent a user behavior item (action) with the same name, for example, "Edit file", and
- they have similar temporal parameters $T$ (it means that the dispersion of temporal lengths $T$ in this set satisfy a condition $\sigma_T \leq \sigma_{T_{max}}$).

An edge $E_i^k$ that connects nodes $G^{k-1}$ and $G^k$ (or more strictly, the instances that are represented by these nodes) are described in a form:

$$E_i^k :< \mu, \sigma, G_{next}, Arel, E_{i+1} >, \tag{8.5}$$

where $\mu, \sigma$ are the distribution parameters of the edge between $G^{k-1}$ and $G^k$; $G_{next}$ points to the next node $G_k$; $Arel$ is a name of relation that connects $G^{k-1}$ and $G^k$; and $E_{i+1}$ points to the next edge that comes out of the node $G^{k-1}$. As it is possible to see, edges outgoing form each node are organized as a linked list. Each node contains a pointer to such a list.

Each *action* in a stream has the format:

$$O_i :< I^A, T, M_{own} >, \tag{8.6}$$

where $I^A$ points to the instance this action is included in; $T$ is the temporal length of the action; and $M_{own}$ is the pattern where this action belongs to. Each action can be included only in a single pattern.

All classes $A_i$ are described in *actions class database*, which is initially formed by the administrator and contains the following records:

$$A_i :< name, \nu_A, starting\_event, ending\_event, auxiliary\_events >, \tag{8.7}$$

where $name$ is the name of the class (for example, "Send E-mail"); $\nu_A$ is the security importance of that kind of action from the point of view of an operating system policy; and $starting\_event$, $ending\_event$, and $auxiliary\_events$ are the structural descriptions of the action class. It defines an event that starts this action, an event that finishes the action, and possible events between them. An example of different descriptions of a single action was provided in Section 5.1.

### 8.3.2 Learning Process

Each user action from the stream comes to a detection server where it is converted to an instance of an action class (see Chapter 6 for details)[3]. The action class encapsulates all attributes common for some set of events. An instance of the action class is used for specifying more specialized types of actions (in our case they group the same kind of actions with the same kind of temporal characteristics).

In order to use the system it has to be trained first. The training period consists of two phases: training itself (approximately 80% of the training time) and setting thresholds (20%). During the first phase the system has to create profiles for all users. These profiles have to be detailed enough in order to allow the system to perform a classification with acceptable accuracy. In other words, the first phase has to be long enough to allow the system to learn a number of patterns in user behavior (otherwise later it will make numerous classification errors). Also this phase cannot be too long. It is difficult to submit only "good" cases during long periods of time, the longer the time interval, the higher the probability that there will be "bad" cases among a training set that may increase a false positives rate. Additionally, the long training phase may cause some over-training: it is

[3] In our prototype we have used "k-means" clustering algorithm to separate instances.

possible that the system learns too many unnecessary behavioral details rather than patterns, which will result in the growth of a false negative rate.

On the second phase, the system calculates individual thresholds for every profile. It is done to increase the accuracy of the classification. For these calculations the system submits positive and negative cases to a classifier and defines the detection threshold according to the classification results. Positive cases are taken from a stream of events of the same user (from the part that were not present in the training set on the first phase). Below, in the next section, we discuss what the most suitable lengths are for the learning phases.

As was discussed in Chapter 6 there is a concept drift issue when dealing with behavioral models: users' behaviors tend to change with time, therefore the effective lifetime of a static user profile is limited (false positives rate grows with time). To deal with this issue we have developed our methods in a way that they constantly update user profiles enabling new information to appear and old to disappear for the profiles. To test the ability to successfully update the user profiles we took all users' history in our experiments. Some users' data was containing up to 400000 actions covering more than six months of user activity. During this time the behavior has changed in one way or another. In our experiments the system was trained during the learning phase (first few weeks of the user data) and then tested on the rest covered by the user data interval (sometimes more than six months), updating profiles when necessary. Therefore, all our experiments were conducted on the dynamic profiles and results described here represent the system's performance in a real word environment.

During the intrusion detection system's implementation we were faced with the problem of the behavioral pattern representation for automatic processing (problem 2 from Section 8.1). In Chapter 6 we assembled all behavioral patterns of the same user into a temporal-probabilistic tree, which is very demonstrative and easy to understand, but at the beginning it is necessary to perform an initialization process. At this stage the system has a number of patterns. Each pattern is represented by a set of nodes connected with each other by edges. However, there are no connections yet between patterns (they are connected later at the end of the learning process[4]). Therefore, at the beginning of the learning process the system has to operate with the structures we call chains (Figure 8.7).

Here we concentrate on chains to provide some insights and give some examples of how the patterns are represented. Every state in the chain is connected with another one by a basic relation (in our case a solid line represents *before* relation, and the dotted line *during* relation). Each chain consists of two parts: conditions of transaction and possible destination states. It is possible to say that if the chain's conditions are met then there are a number of destination states and with a certain probability a user will choose one of those states. There

[4] This is done at the optimization state. See Section 6.2.2 for details.

are two examples given below: Figure 8.7 demonstrates the internal representation method used in the prototype and Figure 8.8 shows one of the real patterns taken from one of the user profiles.

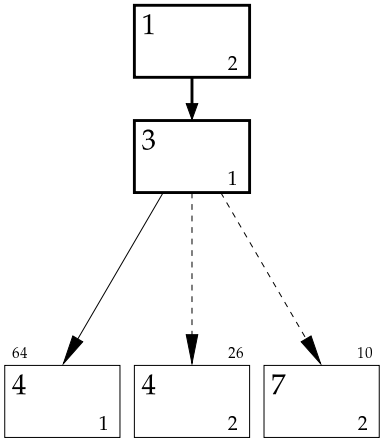


Figure 8.7 An example of a pattern

The conditions of the chain are expressed by several actions connected by relations. In our example (Figure 8.7) we have two actions: class ″1″(instance 2) *before* class ″3″ (instance 1). It is possible to increase the number of actions and increase the patterns' descriptive precision, but at the same time we may increase the rate of false negatives, since the patterns will describe too many details instead of the behavioral trends. We will explore this possibility in the following chapter.

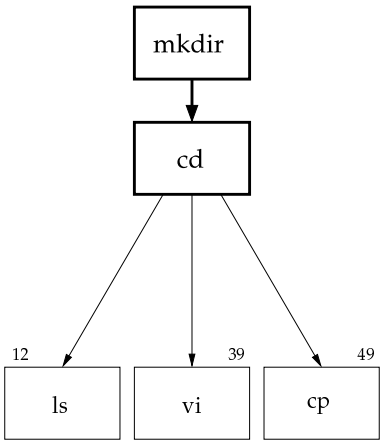


Figure 8.8 A real pattern example

The second part of every chain is a set of actions. These actions represent possible actions that a user may choose after performing those described in the condition part. In our example there are the following possibilities: 64% that the user

will perform an action from class ”4” (instance 1) after finishing the previous action, 26% and 10% that the action from class ”4” (instance 2) and ”7” (instance 2) respectively will start before the previous action is be finished. Temporal parameters that are stored with action and relation instances help to determinate how much time should pass between the previous action's end and the next action's start in the first case (for the relation is *before*). In the second case they show how the previous and the next action overlap each other (for *during*).

Here in Figure 8.8 is a real pattern taken from one of the user profiles: if the system observes *mkdir* - directory creation *before cd* - changing a current directory, then it should expect (relation *before*): *ls* - directory listing request with probability 12%, or *vi* - text editing - 39%, or *cp* - copy command - 49%.

Oversimplifying, it is possible to say that the chain's structure is an *if-then* rule, which contains an extensive sequential and temporal statistical information. Also, these chains may be assembled in a tree, for example for manual profile control.

## 8.4 Summary

In this chapter we discussed the main problems and our solutions during implementation of the prototype. We also presented a possible architecture for an anomaly detector employed for user behavior recognition. Host agents collect all necessary information and transmit it over a secure channel to a detection server, which determines whether there are masqueraders among normal users. The detection server accomplishes this by classifying the incoming actions by comparing them with a corresponding profile. It has an independent process in its memory for each of all active users. Every incoming action is dispatched to a corresponding classification process, which, in turn, performs the classification itself.

In the next chapter we are going to provide a description of the testbed used and discuss the obtained results comparing them with performance results of other anomaly intrusion detection systems. We are also going to discuss the obtained results with respect to the research objectives stated in Section 1.4.

# 9 EXPERIMENTAL SETTINGS AND OBTAINED RESULTS

This chapter describes experimental settings and the results obtained during experiments. The experiments were performed on a prototype described in the previous chapter. The prototype and employed by it approaches are designed to provide an efficient anomaly detection meeting the system considerations listed in Section 8.1.

## 9.1 Experimental Settings

In this section we concentrate on the experimental settings and data collection.

### 9.1.1 Note on the Evaluation and Simulation Process

In this thesis we use examples of vulnerabilities, descriptions of operating environments and commands derived from UNIX class of operating systems. Their source code is available for widespread public scrutiny. Thus, details of security vulnerabilities available from CERT advisories (1999) and Bugtraq electronic mailing list (1999) provide a good example to demonstrate our ideas. We believe that detection techniques and the principles of the detector described in this thesis, are largely applicable to other operating systems as well, even though the details of sensors may differ.

The UNIX operating system is widely used and has been extensively studied in the security community. It has a highly configurable user environment with a rich command language that permits a large range of possible behaviors. To define an information source we had to choose between GUI events, network packets, shell keystrokes, and system call traces.

Network packets do not tell much about a certain user. They may be effectively used for misuse signatures monitoring and are helpful in detecting intrusion attempts. However, they are not very useful in discovering the patterns of user behavior. It is very difficult to link a network packet with a certain user

especially if several users share the same computer or DHCP is used and the IP address of the same workstation is changing constantly.

Shell command input is not easy to use since it is very difficult to interpret some strings because of alias usage. It does not catch all users' actions (for example, if something is done using a mouse).

GUI events provide all necessary information (and even more). It is very easy to link a certain user with processes or objects. Thus, we considered them as the most suitable information collection mechanism for user recognition. System call traces may be used in the case that the user does not use GUI.

The experiments were carried out under the RedHat 7.1 operating system. The programming techniques and language features we have used are not unique for C++ and may be applied to other programming languages as well. The choice of the language was dictated by the following reasons, not all of which are unique to C++:

- The source code of the operating system under which our tests were run is on C language. Therefore, it will be easier to recompile a system code when we modify the kernel (for example implement sensors as a part of it). We have used the egcs-c++-1.1.2 C++ compiler (GCC manual, 2000) for our tests.

- Our familiarity with C++ and its development environment.

- The availability of a large collection of ready to use libraries. These libraries are already well optimized. Thus, we do not need to spend much time writing and optimizing the code.

- Availability of support tools in the form of grammar recognizer generators like yacc++ and lex++ (Bird, 1988).

To implement the prototype based on methods and algorithms developed in this work we have used C++ (Stroustrup, 1991) as a programming language. The prototype consists of different modules and tools that together contain more than 25000 lines of source code.

### 9.1.2 Data Collection

As was already mentioned in this thesis there is a problem with getting proper data to test intrusion detectors. The available data usually contains unnecessary features, it has no temporal info or it is not detailed enough. As a result, quite often intrusion detection approaches are being developed for a certain kind of data that is already available or the data is artificially generated to test the approaches' viability. In the first case the development process results in an ad-hoc intrusion detection system lacking flexibility and closely connected to the type of data being processed. In the second, it is very difficult to generalize the results

because assumptions according to which the data was generated may not hold in real life.

There is a third possibility - collection of one's own data. This way allows one to create flexible approaches for intrusion detection, because it is possible to collect features of user or process behavior that are required by a classification method without making compromises. However, it is often necessary to implement one's own data collection mechanisms since already available ones may not collect the required features or provide necessary details. It also takes a very long time to collect a statistically sufficient amount of data.

In this work we decided to develop and implement a host-based data collection mechanism in which host agents are responsible for the data collection. The host agent software was implemented in the earliest stages of the prototype implementation project. After this it was constantly used to collect information about users behavior during the rest of the project.

The host agents collected the audit from different operating systems: RedHat Linux 6.1, RedHat Linux 7.1, Debian GNU Linux 2.0, FreeBSD 4.1. Audit logs of 16 different users were collected for the experiments.

The audit data contains the history of 16 (U1-U16) different authorized users. The amount of data available varies among the users from 50000 to almost 400000 tokens, depending on their work activity rate and length of the time interval during which their activities were monitored. The sequences of the users actions were given as an input stream for the learning process and a temporal-probabilistic tree was constructed for every user. Finally, we would like to note that the nature of the collected information (all workstations were inside a single local area network) did not allow us to employ mobility measures described in Chapter 4 in the profiles.

### 9.1.3 Experiments

The experiments described in this section were performed on an AMD 1.4 GHz workstation with 512MB of memory running RedHat Linux 7.1 under a light load. When designing the experiments and evaluating the prototype we used IDS evaluation techniques (such as described in (Durst *et al.*, 1999) and (Puketza *et al.*, 1996)) to be able to compare results with other anomaly detection approaches.

Here in addition to the traditional accuracy characteristics we employ a maximum *time-to-alarm* measurement. It determines how quickly an anomalous situation can be detected. It should be short enough in order to detect hostile activities quickly, but long enough so that normal work is interrupted by false alarms as rare as possible. In this work we define the time-to-alarm as a maximum time interval during which an anomaly will be detected if it is present in the current behavioral sequence.

For our experiments we collected output streams that were produced by host agents for different users. Later those outputs were given as an input for

profile building techniques. Lacking the traces of intrusive behavior, we tested the approaches' viability on the task of differentiating between different valid users. All anomalous situations were simulated by testing a profile of a valid user against sequences of events issued by the other users. This testing approach does not cover all possible misuse scenarios, but it gives the possibility to evaluate the viability of our approaches. Being able to constantly authenticate a user (according to his behavior) in real-time the system is able to detect impostors and, therefore, perform the anomaly intrusion detection.

For the sake of experiments we implemented two modes of classification: normal and batch mode. During a normal operation the system functions as described in Chapter 6, i.e. waits for user real-time actions and classifies them. Since we had already collected event sequences[1] we had to implement additional functionality to the host agent - user simulation. In this mode each host agent is resident in workstation's memory and has a file with sequences of events of some user. The host agent takes actions from a local file and sends it to the detection server keeping the actions lengths and time intervals between them the same as a user performed them.

On the experimental phase we had to run certain experiments tens of times to debug the system, choose the correct parameters, and collect the necessary statistics. Therefore, to save time we have implemented a batch mode. In this mode the system does not need to wait when a new user action comes. All user event sequences are submitted to the classifier at the same time. Since all the events in these sequences are supplied with temporal information there is no loss of sequential or temporal data.

The following results were obtained when our prototype of the intrusion detection system was trained on the user logs. Both temporal-probabilistic tree and relational matrix approaches were applied in order to compare them.

## 9.2 Performance

In this section we are going to motivate the choice of parameters and discuss the obtained results.

### 9.2.1 Space Requirements

Here we consider the possible space requirements for user profile storage. Figure 9.1 shows the sizes of user profiles for the relational matrix approach. The mean size of the profiles is 3413 Bytes. There are several factors involved in the profile's size: vector of classes and relational matrix. Their sizes depend on the overall number of action and relation classes and average number of instances in those classes, especially the number of classes affects the sizes of the vector and the matrix. However, the latter grows much faster then the former. For our case

[1] Event sequences used in this work were collected during different time intervals and in different places.

the size of vector varies between 420-944 Bytes and the size of matrix between 1132-5300 Bytes.

An average number of instances per each class is approximately the same for all users - 2,5 and it shows that the size of the average profile depends mostly on the number of classes used for profile creation and later for classification. In other words the size depends on the number of actions/programs employed by the user.

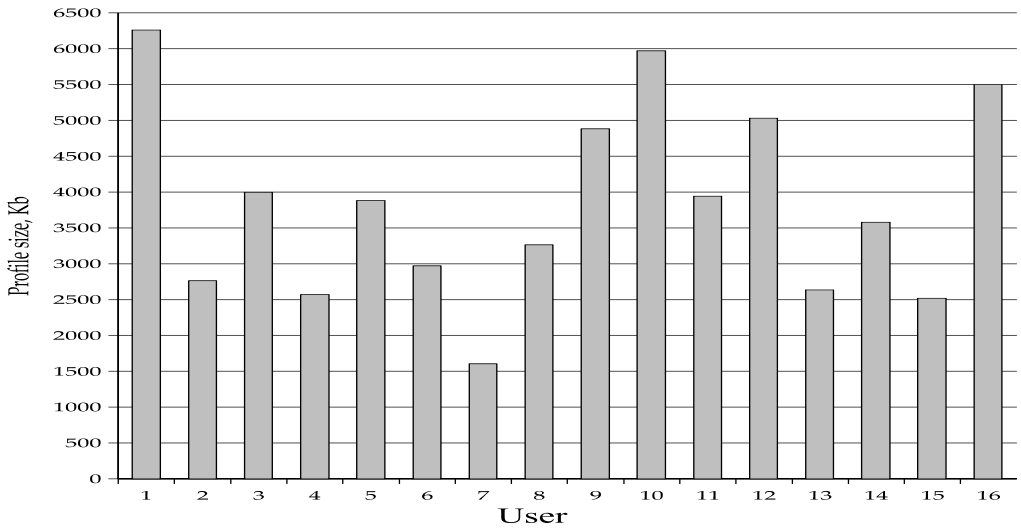


Figure 9.1 The size in Kb of each user profile (matrix approach)

Figure 9.2 depicts the space requirement for each profile that uses a temporal-probabilistic tree for behavior representation.

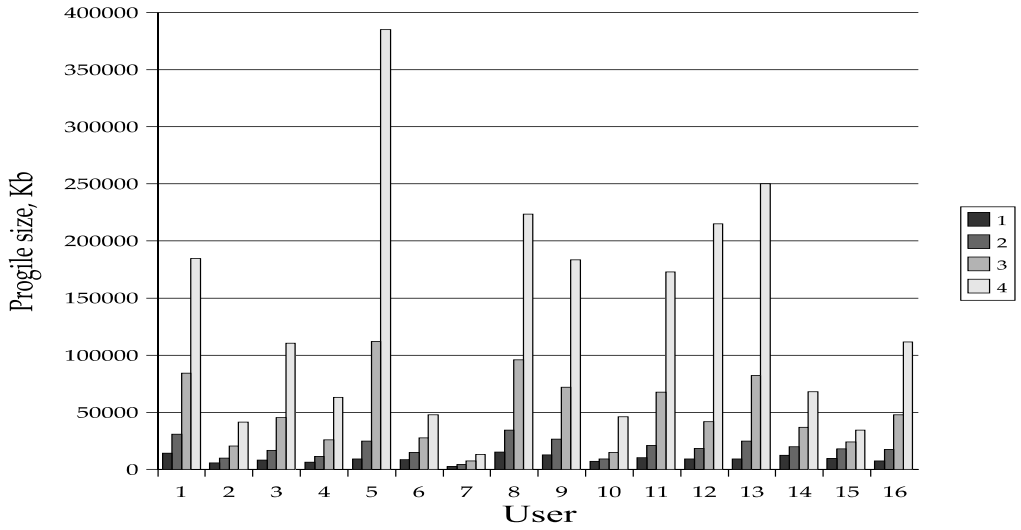


Figure 9.2 The size in Kb of each user profile (temporal-probabilistic tree approach) for different number of conditions

As can be seen the figure shows four different cases for every user. These cases correspond to a different number of conditions in behavioral patterns. For our experiments we have chosen two conditions for all users (later in this chapter we discuss how the number of conditions affects the classification accuracy). The mean size of the profile is 21832 Bytes and the size of the profile varies between 4176 Bytes and 34476 Bytes.

There are two factors that may significantly affect the average size of the profile: the number of classes and the number of conditions for the patterns. According to our observations (it can also be seen form Figure 9.2) the number of classes does not affect the size of the profile as much as the number of encoded conditions. Therefore, fixation of the conditions number (with two conditions we can get more or less reasonable size of profile) gives us an ability to keep the size of the profile in a certain range, and, thus, estimate the space required to store profiles.

Finally, we would like to notice that space required to store a single profile based on a temporal-probabilistic tree is approximately six times bigger than necessary for a profile based on a relational matrix. Consider a case when our prototype is monitoring 10 000 users. We can approximate that we will need to reserve in order to store all profiles about 32.6Mb in the first case and 175.1Mb in the second case.

### 9.2.2 Accuracy

There is a number of different parameters that may affect classification accuracy. Here we try to carefully select those in a way that minimizes an overall error rate. Each profile was tested against the corresponding test set for each user. A test against a user's own profile allows us to examine false negative (false rejection, FR) alarm rates, while testing against other profiles allows us to determine false positive (false acceptance, FA) rates. Below we are going to discuss our choice of parameters for the relational matrix and the temporal-probabilistic tree approaches.

#### Sliding Window Length

Before choosing parameters of the classification we decided to give some considerations to one of its most crucial attributes - the length of the sliding window. There is a certain interval inside which it is possible to choose this length. It was shown that the amount of events inside the window can not be smaller than six (Tan and Maxion, 2002) to achieve the reasonable accuracy rate. From another side this interval is limited by acceptable time-to-alarm length. If the longest possible time-to-alarm is four hours and our system gets about 369 actions from each user on average, then it is possible to see that the length of the window lies in an interval of between six and 180, and it should be chosen from this interval depending on the required acceptable accuracy of the classifier.

It was suggested that conditional entropy can be used to determine the appropriate window size for probabilistic classifiers (Lee and Xiang, 2001). It was shown that there is a correspondence between a fall off in entropy and the appropriate window size of the classifier.

Assuming that each profile contains different amount of regularities we decided to check whether conditional entropy affects our system. To perform the necessary tests an audit log was taken that contains the history of five different users. To make sure that these profiles differ only in terms of irregularities (measured as conditional entropy) we identified five action classes that were common among all of the users. Other classes were not taken into account. In other words, we identified five actions that all of the users were performing and built profiles basing them on usage only of these classes. During the classification we did not observe significant differences in the classification accuracy between profiles. They were lying in intervals of less than 10% of the average detection accuracy. This suggests that although our classifier uses probabilistic features the conditional entropy can not be used as a main method to determine the appropriate window length.

**Relational matrix**

First of all, it is necessary to determine a sliding window length. This length explicitly affects time-to-alarm rate: the longer the window the longer the time-to-alarm interval. We observed that during an eight hour working day a system receives on average 369 actions per user. The average time between actions is about 78 seconds per user. This means that if we take the window length as 20 actions our system will have a minimum of 26 minutes time-to-alarm, which is, in our opinion, quite suitable. Thus, since the system determines the length of the window during the learning stage, we established the minimum length of 20 actions and the maximum of 60 (time-to-alarm lies in [26,78] minutes interval). The length depends on user activity and the size of patterns in the user profile. We would like to point out that in our experiment the system was choosing the window length to minimize the overall error rate. However, in real life some additional requirements, such as shorter time-to-alarm, may be applied during this process.

In Figure 9.3[2] it is possible to see how the error rate changes over time during training when different window lengths are chosen. It is possible to notice that it takes about two weeks to train our system choosing the length of the sliding window as 20. After this its error rate will not be higher than 17%.

**Temporal-probabilistic tree**

Firstly, it is necessary to determine a number of conditions in patterns in our prototype.

[2] Tables with accuracy results can be found in the Appendix 3.

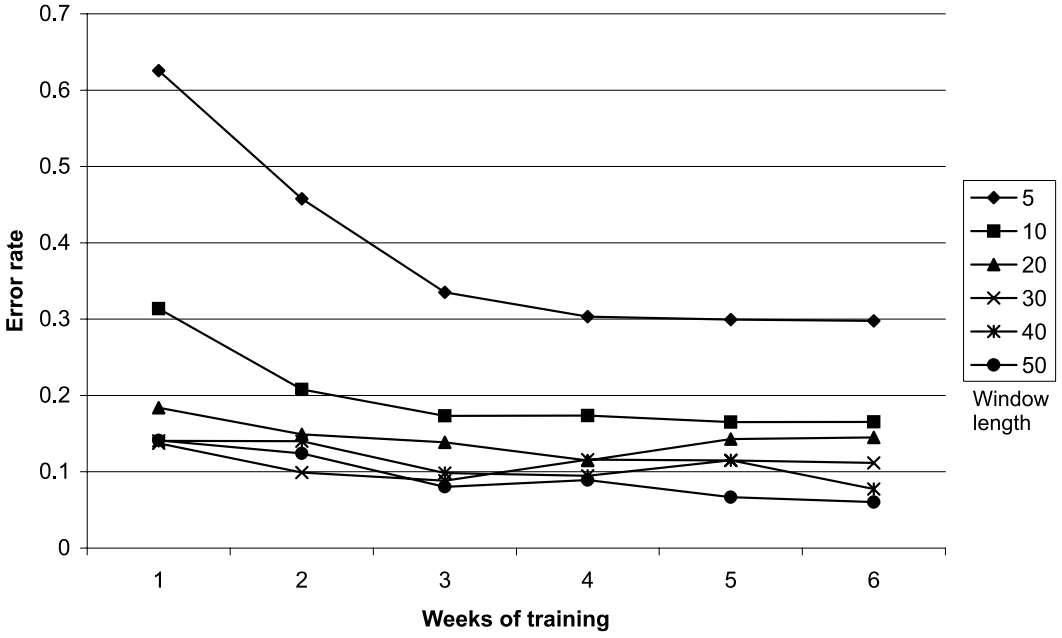


Figure 9.3 Class approach: dependence of the overall error rate on the length of the training period for different window lengths

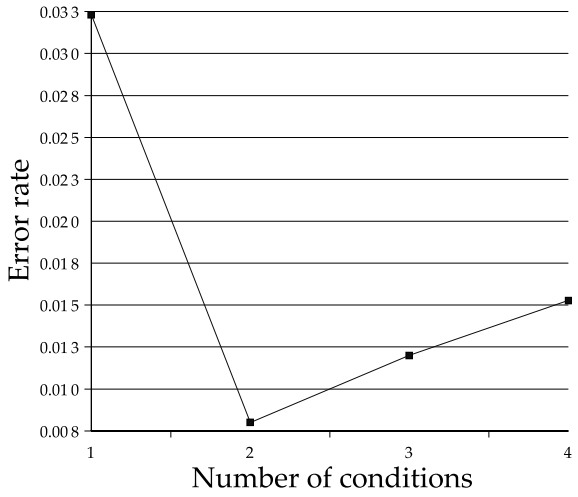


Figure 9.4 Dependence of the overall error rate on the number of conditions for the prototype

In Figure 9.4 it is possible to see that the two conditions are the best choice for the prototype. When the number of conditions is one, the patterns encode too general a model of behavior, therefore the false positive rate is high. With the growth of the conditions' amount the patterns will become more and more specific. They will describe small unnecessary details rather than trends in user behavior. As a result we will encounter more false negative cases.

Figure 9.5 shows that the length of the sliding window, which was chosen at the beginning of this section, is also quite suitable for a temporal-probabilistic tree. The system is ready to use after approximately one or two weeks of training. After this its error rate will not be higher than 8%.

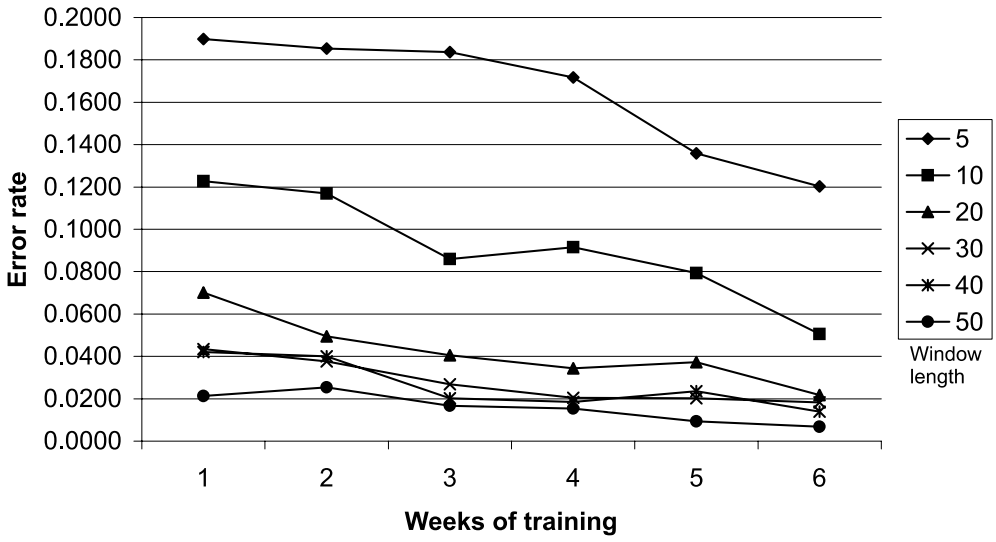


Figure 9.5 Temporal-probabilistic tree approach: dependence of the overall error rate on the length of the training period for different window lengths

It is possible to see from Figures 9.3 and 9.5 that for long window lengths an overall error rate may slightly grow during a training period. It may happen because a sliding window is used not only for classification but also for learning. Parameters of each profile are calculated over the window. It means that taking a long window length the system over-generalizes parameters for each profile. In other words, the system starts to accept a more general model of behavior increasing the number of false positives.

There is a differential analysis that has been put as a foundation of user behavior classification, in which the detection accuracy can not depend on the number of profiles in the system (in contrast to absolute analysis). We train the system giving it only ”good” examples, i.e. only event sequences of a single user for which we are building a profile. Then during the classification each classification process has a profile of a single user and only compares this user actions with his/her profile. Therefore, growth of the number of profiles in the system will increase a number of classification processes working in parallel, but not the error rate. In our experiments there is a possibility of increasing an error rate with the number of profiles growth because we have tested each profile against all other profiles, which never happens in real life. Figure 9.6 demonstrates the dependence of the detection accuracy from the overall number of profiles in the system.

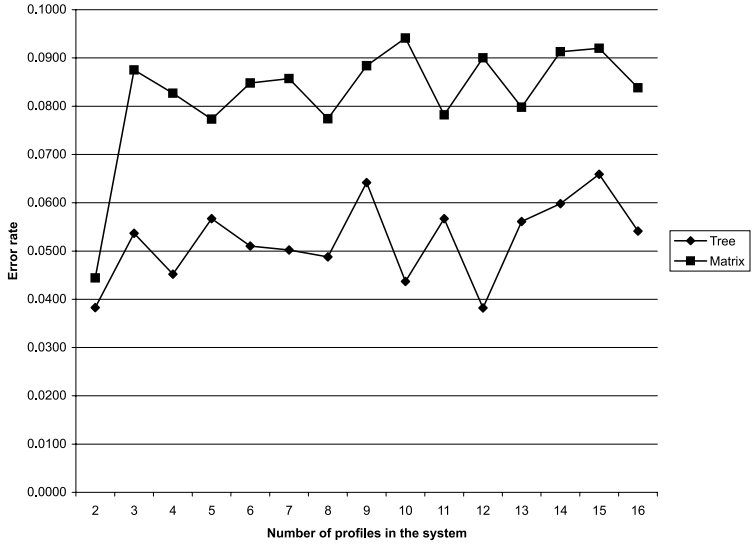


Figure 9.6 Dependance of the accuracy on the number of profiles in the system

To assess the classification accuracy we have used Receiver Operating Characteristic (ROC) curves (Swets, 1988). They visualize the trade-off between false alarms and the detection rate with respect to chosen thresholds. In other words, by choosing different classification thresholds it is possible to see how the detection and false alarm rates are linked. Relaxing the classification requirements (for example, by lowering thresholds) we can increase the detection rate, but also the misclassification rate will be increased, i.e. the classifier will falsely accept abnormal cases, and conversely the same can be said for the case if we strengthen the classification requirements.

To apply the ROC curves approach we need to clearly define what constitutes ”normal” and ”abnormal” behavior. In our case, we were taking a single user profile each time and therefore, we were defining ”normal” behavior as the behavior defined in the current profile and ”abnormal” as described in the rest of the user profiles in the system.

Plotting the detection rate as a function of the false alarm rate for both of our approaches we reached Figures 9.7 and 9.8.

The accuracy of the test depends on how well the classifier separates the behavior being tested into ”normal” and ”abnormal”. Accuracy is measured by the area under the ROC curve. An area of 1 represents a perfect test; an area of 0.5 represents a worthless test. If we classify the performances of our classifiers in the same way we can notice that in Figure 9.7 this area is more than 0.80 and in Figure 9.8 it covers more than 0.9 of all space. This estimation shows that the classifiers have performed fairly well.

The accuracy results achieved during our experiments are quite promising, however they should be treated carefully. Although the experiments showed that the approaches developed in this work may be used for an online user ver-

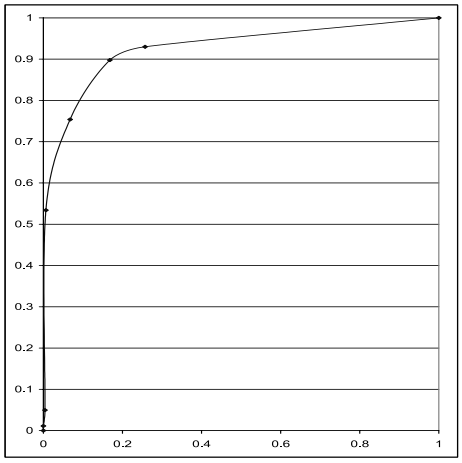


Figure 9.7 ROC curve for the classifier based on the matrix approach

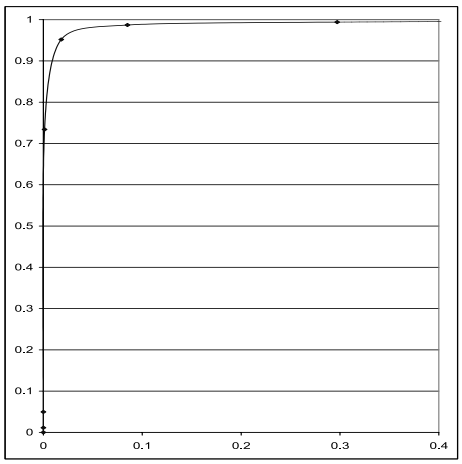


Figure 9.8 ROC curve for the classifier based on the temporal-probabilistic tree approach

ification, it is possible that for different user categories (not present in the used test set) the parameters of the classifiers should be different to achieve the same accuracy. Also, we could not test or simulate all possible variants and cases of user behavior, therefore we were not able to build a complete model of user behavior for each profile. This applies to all anomaly-based classifiers. It is not possible to construct a complete model of normal (misuse) behavior, however the efficient use of strict anomaly detection does not necessarily require a complete model to be constructed - a subset is also acceptable (Beetle, 2000).

### 9.2.3 Timing Results

In order to measure classification time we have created an audit file that contains 14625 actions. We applied both approaches on this audit log eight times. The resulting time consists of two intervals: classification time for actions inside of a sliding window and the time necessary to update the coefficient of reliability

before moving the window. Obviously the former one depends on the window length and it forms a variable overhead. The latter one is based on certain formulas that require a constant amount of time per one calculation, thus it does not depend on classification parameters and forms fixed overhead.

As it is possible to see from Table 9.1, for the window size (20) that we choose our prototype has spent 32.23$\mu s$ for classification of a single action.

Table 9.1 Timing results for the relation matrix approach

| Window length | Fixed overhead, s | Variable overhead, s | Total time, s | Time per action, $\mu$s |
|---|---|---|---|---|
| 5 | 2.723 | 0.908 | 3.361 | 31.03 |
| 10 | 2.723 | 0.957 | 3.68 | 31.45 |
| 20 | 2.723 | 1.048 | 3.771 | 32.23 |
| 30 | 2.723 | 1.12 | 3.843 | 32.85 |
| 40 | 2.723 | 1.198 | 3.921 | 33.51 |
| 50 | 2.723 | 1.28 | 4.003 | 34.22 |

Table 9.2 Timing results for the temporal-probabilistic tree approach

| Window length | Fixed overhead, s | Variable overhead, s | Total time, s | Time per action, $\mu$s |
|---|---|---|---|---|
| 5 | 6.097 | 0.908 | 3.361 | 59.87 |
| 10 | 6.097 | 0.957 | 3.68 | 60.29 |
| 20 | 6.097 | 1.048 | 3.771 | 61.06 |
| 30 | 6.097 | 1.12 | 3.843 | 61.68 |
| 40 | 6.097 | 1.198 | 3.921 | 62.35 |
| 50 | 6.097 | 1.28 | 4.003 | 63.05 |

In case of a temporal-probabilistic tree the classifier has spent 61.06$\mu s$ to classify a single action, which is twice as long as in the first case. Consider the extrapolation of these results to estimate the performance in real settings. Consider that there are 10 000 users in the system. If they are all active the system will get 461 540 actions per hour. In this situation a classifier that uses the relational matrix approach will take 15s to classify all actions, and for temporal-probabilistic classifier it will be 28.2s. This is about ≈0.5% and ≈1% of the CPU's hourly activity.

### 9.2.4 Analysis

In this section we analyze how a selection of different hardware could affect the results described in this section. This gives the possibility to make comparisons with other systems.

*Faster system.* If these experiments were run on a faster system then the classification time will slightly decrease. However, according to our

observations, the CPU overhead due to the classification processes is insignificant. If the detection server is devoted for a classification task (as in our experiments) then it handles 10 000 users without any problems and therefore, making the system faster does not change the overall performance significantly.

*Faster network connections.* The amount of information sent over a local network depends mainly on the number of active users. Our prototype receives in average 40 bytes per one action. Therefore, for 10 000 users the network traffic overhead will be 513 bytes per second, which is insignificant amount for modern networks.

We can conclude that the major factor that affects the system's performance most significantly is the number of users that are active at the moment.

## 9.3 Profile Cross-Validation

So far the system has a case of intrusion, it predicted a possible damage degree. Now it is possible to separate the intrusion. What can the intrusion detection system do in order to expose a real source of abnormality? In this section we investigate the possibilities to determine a source of the intrusion in case of a masquerader.

The main objective of the intrusion detection system is to detect an intrusion, but knowing that an intrusion incident occurred, does not, however, reveal the full story. A second issue of importance is a source and method of the intrusion. This information is needed for catching a person(s) who stands behind the intrusion and guards the system against the same kind of hostile situations in future, i.e. plug the holes through which the intruder has managed to break in.

Since most of the intrusions and abuses are carried out by insiders or other well-informed users (Lane and Brodley, 1997b), it may therefore be helpful in detecting the intrusion source to inspect the user profiles in the system. In the case when an abnormal event stream is detected, it may mean that the events of the stream do not belong to the person he is claimed to be, i.e. there is an intruder in the system masquerading as a normal user. Thus, making the assumption that the intruder is an insider, the system tries to match the abnormal event stream against other users' profiles and analyzes the acquired coefficients of reliability. If at the end it finds a coefficient that is higher than the current coefficient threshold then the system has suspicions that this legitimate user abuses someone else's network account and the system reports this to the system administrator. In other words the system has a pattern of abnormal behavior and it tries to find the same pattern in user profiles as part of normal behavior.

The audit data that was taken for the experiments contained history of eight (U1-U8) different authorized users. We have also chosen one user (U0) as a masquerader and took his normal behavior data. Then the history data of nine

users was given as an input stream for the learning process and a temporal-probabilistic tree was constructed for every user. After finishing the learning process, it is supposed that the user U0 begins to behave as an intruder masquerading as one of the users U1-U8. User U0 was logged in as user U1, U2, ..., U8 and was performing some activities peculiar to masqueraders: reading email messages, copying files from "victim's" directory to own home directory, installing "backdoor", etc.

After an intrusion was detected the intrusion detection system has been matching an intrusive pattern against profiles of other users. The results are collected in the Table 9.3. Rows represent the user who U0 pretends to be. Columns - represent the probability that the found abnormal sequence belongs to a certain user. "x" shows an abnormal behavior case. In other words, we can see in the table the probability that an abnormal sequence is issued by the user $U_j$ when the user U0 caused it masquerading as user $U_i$.

Table 9.3 The results of masquerader detection

| | **U0** | **U1** | **U2** | **U3** | **U4** | **U5** | **U6** | **U7** | **U8** |
|---|---|---|---|---|---|---|---|---|---|
| $r_{U_0}$ | 88.3 | **24.9** | **32.7** | **34.8** | **28.1** | **29.3** | **36.8** | **27.2** | **35.1** |
| $r_{U_1}$ | 8.7 | x | 4.3 | 7.4 | 13.6 | 9.6 | 8.1 | 3 | 7.7 |
| $r_{U_2}$ | 11.9 | 1.5 | x | 10.7 | 12.1 | 6.5 | 8.9 | 10.6 | 11.2 |
| $r_{U_3}$ | 2.1 | 16.5 | 6.9 | x | 9.4 | 1.7 | 12.1 | 14.7 | 24 |
| $r_{U_4}$ | 5.8 | 12.1 | 3.7 | 1.4 | x | 2.4 | 1.1 | 13.2 | 7 |
| $r_{U_5}$ | 1.2 | 6.9 | 15.1 | 4.1 | 2.1 | x | 6.4 | 7.2 | 11.7 |
| $r_{U_6}$ | 8.4 | 4.3 | 12.9 | 8.3 | 10.5 | 13.5 | x | 7.8 | 9.1 |
| $r_{U_7}$ | 6 | 3.3 | 5.3 | 6.1 | 15.9 | 10.6 | 2.3 | x | 7.1 |
| $r_{U_8}$ | 7.3 | 2.7 | 2.4 | 8.1 | 8.7 | 13.6 | 10.1 | 3.3 | x |

In the first column it is possible to see a case of normal behavior. A sequence of events issued by the user U0 was taken and tested against every profile. We can see that the system recognized user U0 with a rather high probability - 88.3% compared to the other users. In the other columns the system has detected an abnormal sequence due to a low final coefficient of reliability. Then the system has tried to match this sequence against U1-U8 user profiles. It can be noticed that the probabilities that the system recognizes the masquerader as any user, except U0 (false detections) $\leq$16.5% and as a real masquerader (user U0) is between [24.9%, 36.8%] (emphasized with bold fonts).

In Figure 9.9 (a,b) it is possible to see visualized results of Table 9.3. It is possible to notice from Figure 9.9.b that all results may be separated into three categories/clusters. The first cluster is formed by a single point - it identifies a point where coefficients of reliability are situated, of the users that behave normally. The second cluster separates cases when, after detecting an intrusive sequence, a source of intrusion was successfully detected. Finally, the last cluster

(at the bottom of the figure) shows the cases in which the intrusion source was detected incorrectly.

After observing these clusters several questions arise. It is possible to see in Figure 9.9 that the coefficient of reliability for a user U0 is quite high $\geq 0.883$ when he is not masquerading. Relatively to this the same coefficient is lower $\in$[0.249, 0.368] for the same user when the same user pretends to be someone else. Why do we observe this deviation? Perhaps because the person when masquerading does not behave in the same way when he is using his own account. Typically, the masquerader uses accounts as a base for continuing attacks on other accounts/workstations, to increase the current privileges, or for information stealing purposes. This set of actions very rarely corresponds to his normal behavior, unless he uses his own account for breaking into another systems or accounts and the system recognizes this set as a normal behavior. The second point is that the command history for the user U0 is quite short relatively to the other eight users since it was created artificially (i.e. it is imitation of normal work), therefore it may not cover all the aspects of the user's behavior.

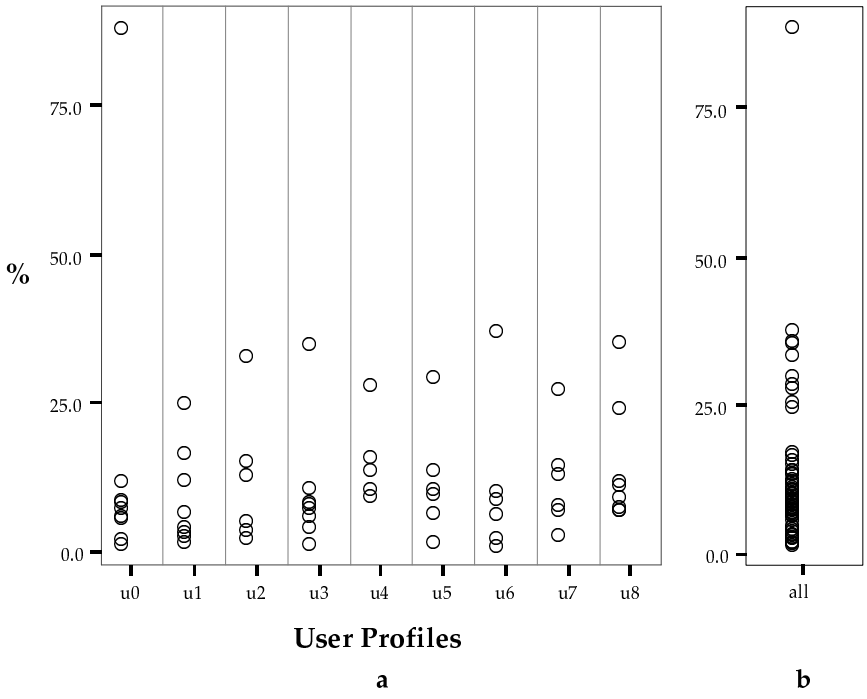


Figure 9.9 Coefficients of reliability for all users: (a) each column displays a single profile tested against all test sets; (b) all profiles tested against all test sets

From Table 9.3 we can see that when the system detects the abnormal behavior the probability that this behavior belongs to the user U0 (i.e. true masquerader) varies between 24.9-36.8% and the probability that the same pattern belongs to any other user $\leq$16.5%. Thus, it seems that there is an obvious deviation between these intervals and we can establish a threshold that splits these two sets and

recognizes new cases with a high probability (since these intervals are not too close to each other).

We would like to summarize this section by our observation that when the temporal-probabilistic tree grows the probability of exposing the real intruder also grows. To recognize a certain user among the others the system needs to catch a "general" line of user behavior (i.e. observing deviation). While detecting a source of intrusions the system needs to know, distinguishing details (observing similarities), which are sometimes hidden rather deep, since when masquerading the intruder does not follow his/her normal way of behavior.

According to experiments, the system, based on our approach is able to correctly recognize a certain user in more than 87.3% cases[3]. In case of an insider intrusion it is able to expose the real source in 24.9% to 36.8% of cases.

## 9.4 Comparison with Other Approaches

In this section we compare the performance of our approaches with other different anomaly detectors. Due to reasons presented at the beginning of the previous chapter it is very difficult to make quantitative comparisons between classifiers, thus we try to provide some qualitative ones. The overview of the approaches can be found in Chapter 2.

*Instance-based Learning (IBL)*

The approach for the anomaly detection based on the instance-based learning was developed by T. Lane and C. Brodley and published in (Lane and Brodley, 1997a), (Lane and Brodley, 1997b), (Lane and Brodley, 1999). Shell command data from eight users over the course of more than two years was used to evaluate this approach. Due to computational constraints a subset of 7000 tokens from each user was employed, representing approximately three months of computer usage.

For experiments static profiles were constructed by training the classifier over approximately three weeks of user activity with a sliding window length set to 100. During the experiments all false positives as well as false negatives were identified. The classifier's detection accuracy lies in 0.7-1 interval. In our experiments we were able to keep the detection accuracy in 0.83-1 for a classifier based on the relational matrix and 0.92-1 for the temporal-probabilistic tree detector.

Time-to-alarm rate was measured in tokens, where each token represents a single user action. Most of alarms were made during an interval between one and 100 actions. In our case the time-to-alarm was defined using the window length as 20 actions. There is an explanation for this. If we define a token as a

[3] As can be seen from the Section 9.1.3, after some improvements we were able to increase the detection rate to the minimal of 92%.

feature involved in the classification process and take a sequence of ten actions for classification then the IBL method uses only ten tokens, i.e. makes the classification basing it on the sequential information (actions themselves). In turn, our methods would use 40 tokens: actions themselves, their temporal lengths, relations between actions, and the relations' temporal lengths. That is why our methods reacted faster.

*Neural Networks Approach*

Here we refer to a report on experiments where neural networks were used for statistical anomaly intrusion detection (Zhang *et al.*, 2001). The used testbed had eleven sources (workstations) and one detection server. After this a classifier was built to detect anomalies in the network traffic. For a simulation scenario 10000 records of network traffic were collected. They were separated: 6000 records were used for training and 4000 for testing. Each type of neural networks was trained for 100 epochs. The misclassification rate was measured as the percentage of inputs that were misclassified: false positives and false negatives.

Five types of neural networks were tested: perceptron, backpropagation, perceptron-backpropagation, fuzzy ARTMAP, and radial-basis function. The studies showed that backpropagation and perceptron-backpropagation networks have similar performance and both networks perform better than the other three types. The results for these networks: mean squared root errors $<0.2$ and misclassification rates $<0.09$.

*NIDES based Frequentist Detector*

This and the following two anomaly detectors were implemented and described in Ali (2000). As a main objective of this work three classifiers for anomaly detection (frequentist-, data mining-, and generic programming-based) were implemented and evaluated. To test these three approaches eleven test data sets with an alphabet size of six were artificially generated with a different amount of noise. Then 160 anomalies were injected into each of these sets.

The frequentist detector is a statistical anomaly classifier, which takes frequencies of various components in the training data as a measure of normalcy of the data. For this approach the hit rate varied between 0.8 and 1 depending on the level of noise in the training data. However, the false alarm rate was sometimes quite high - up to 0.4.

*A Data-mining based Detector*

The detector was based on RIPPER (Lee *et al.*, 1999a) which is a rule learning program. System call traces and *tcpdump* data was used to evaluate the approach. In this experiment the detector had a negligible rate of false alarms (almost 0.00001), however, the hit rate of the detector was surprisingly low - up

to 0.25. That shows that the detector cannot probably be used alone in an IDS (Ali, 2000).

Significantly better results were achieved in another implementation of the data mining-based classifier (Lee *et al.*, 1999b). The experiments were conducted using DARPA data prepared and used by MIT Lincoln Labs. This data is an extensively gathered audit data with different kinds of attacks simulated in a military network environment. The data was collected using *tcpdump* and covers seven weeks of network traffic.

The intrusion detection models were produced off-line. The RIPPER was used to build rules. Results of the experiments were summarized in ROC curves. A classifier with the best set of features was able to detect about 65% (highest value) of anomalies in traffic. At the same time the amount of false positives was approximately 2% and false negatives 33% respectively. Since the experiments were conducted off-line there was no time-to-alarm estimation provided.

*A Generic Programming (GP) based Detector*

This is a classifier built using supervised learning (Ali, 2000). All experiments were performed with a population size of 50, the number of generations was 20 and the maximum depth of the GP tree was 45. Pattern size was varied from three to twelve during the tests. The author claims that the best results were achieved for pattern sizes nine and twelve. The hit rate was in the 0.5-1 interval and the false alarm rate between 0-0.55 for the best case.

*Predictive Pattern Generation*

The probabilistic user models were constructed as a result of learning the classifier that uses the predictive pattern generation (Lee *et al.*, 2000). These models were used to analyze and predict user behavior. Details of the algorithm with its advantages and disadvantages were described in Chapter 2.

The data collection was performed for four different real users. The length of the time interval covering the activities of each user is different: four, six, five, and 25 weeks. The Unix shell commands and system responses were gathered. During the experiments the prediction accuracy of an ideal model varied between 45% and 98% depending on the user profile currently used.

*State-based Approach*

This study used an n-gram based classifier (Michael and Ghosh, 2000). On the learning stage the classifier recorded n-gram of audit events that occured in the training data. Later during the detection these n-grams were compared with incoming data. New n-grams were considered as evidence of an intrusion.

The experiments were conducted on user data covering eleven weeks of work. Unfortunately, the details of the used data were not provided in the pa-

per. The results of the classifier are presented as an ROC curve. The n-gram based classifier could detect up to approximately 93% of attack keeping the false positives at quite low level - 1.5%. However, during additional experiments it was discovered that this approach is sensitive to the data in a training set. The rate of false positive errors varied between 1.5%-34% depending on the context of data and its amount in the training set.

*Chi-square statistic*

The classifier (Ye and Chen, 2001) used to build statistical models of normal behavior is based on chi-square statistic. In the implementation a profile of normal events in an information systems was built. Two sources of data were used to capture normal events: network traffic and audit logs. Network traffic data represented packets traveling over communication links (LAN) between machines. Audit data was taken from audit logs of the host machines (Unix-based). There were 248 types of events considered in the study - the number of Basic Security Module (BSM) allows. The first part of the audit - 1613 events was used to build a model of normal behavior and the second - 1405 was used to test the classifier. Also the intrusive data (1225 events) was involved on the testing stage.

For all intrusion sessions the detection rate was between 35% and 86%. It is necessary to note that all intrusive activities were detected at a very early stage: 71% by the first event and 29% by the second. This shows that the classifier has a short time-to-alarm. However, the estimations of the time-to-alarm interval length are not present in this work.

To summarize this section we would like to note that it is very difficult to find experimental performance results of different anomaly detectors. In publications only accuracy results are provided, and if there is some extra information - it is given in relative values, which are very difficult to compare with the absolute values obtained from our experiments. Moreover, we have not seen any actual profile size, neither classification time analysis for the anomaly detector. Therefore, we cannot provide any comparison of them.

From the comparisons we were able to make, it is possible to see that our classifiers perform sometimes slightly, sometimes significantly better, supporting our initial claim that this work improves the aspects of the current anomaly intrusion systems. Additionally, in contrast to our work, non of the described above classifiers used dynamic profiles to represent behavioral models. Therefore, the provided accuracy results are valid during a short period of time after the learning period is complete.

## 9.5 Evaluation of Results

In this section we summarize the developed in this thesis approaches and compare them with the related approaches presented and discussed in Chapter 2. By doing this we are trying to show how we managed to overcome the stated

above disadvantages, i.e. to assess how well the objectives of this work are met. Section 2.5 discusses the different approaches in terms of how easy they are to program and manage, their source independence, completeness of the model, and their ability to use dynamic profiles. We attempt to classify our approach in the same terms. Table 9.4 presents a summary of characteristics for the relational matrix and temporal-probabilistic tree approaches.

Table 9.4 A summary of characteristics of intrusion detection approaches developed in this thesis

| **Intrusion detection approach** | **Easy to program** | **Easy to manage** | **Online model update** | **Completeness of the model** | **Source independence** |
|---|---|---|---|---|---|
| Relational Matrix | + | + | + | + | + |
| Temporal-Probabilistic Tree | + | + | + | + | + |

We consider in details and justify our classification.

- *Easy to program*. In both our approaches the only thing necessary to define manually is an action database that contains descriptions of action classes. This is done only once for each kind of information source (operating system). After this the system automatically collects statistics and builds behavioral models.

- *Easy to manage*. There are a few parameters described in Chapters 5 and 6 that allow a system administrator to manage the profiles. It is not necessary to perform any manual actions in order to manipulate the profiles. Everything is done by changing a correspondent variable that allows changing accuracy, size, speed of update, etc.

- *Online model update*. The results of the experiments presented here were obtained used dynamic profiles. Thus, we conclude that the approaches are able to update profiles to reflect normal changes in user behavior.

- *Completeness of the model*. Although we argued that it is not possible to build a really complete model of normal (intrusive) behavior due to the impossibility to take into account all possible cases of user behavior and its variations, the models used in our experiments showed that they can be used for anomaly intrusion detection. Also, during the prototype implementation and experiments we did not encounter any situations where we could not represent some scenarios. Therefore, we put "+" here.

- *Source independence*. The approaches were developed to be portable and they are able to use different sources of information. To create the profiles used for experiments four different operating systems were used

as sources of information about user behavior. Although all of them are Unix-based, they have some differences in representing information inside the kernel and details that accompany some of the messages.

In the previous section there is a comparison between the results obtained during evaluation of our prototype with results of the implementations of different approaches. The presented summary of results was made to compare approaches, not implementations. Comparison of Tables 9.4 and 2.2 supports our initial claim that this work improves some of characteristics and overcomes some disadvantages of intrusion detection approaches.

## 9.6 Summary

In this chapter we described the experiments, outlined the main results of the prototype testing, and compared the obtained results with performance of other anomaly detectors. Two algorithms were employed for classification. As can be concluded from the results these approaches have different characteristics and should be used under different circumstances. The approach based on a relational matrix is less computationally expensive, but does not provide such good results as the approach based on the temporal-probabilistic tree. According to its characteristics, it may be efficiently used when there is a large number of users logged in at the same time or for users who log in occasionally. If an average session length is short it makes it very difficult to find patterns inside a session which does not contain many actions. Thus, the relation matrix approach is efficient for these cases since it has been developed for short patterns.

# 10 CONCLUSIONS AND FUTURE WORK

In this chapter the conclusions to this work are presented outlining the main scientific contribution and showing the limitations of the presented work. Additionally, possible future research directions are outlined.

## 10.1 Conclusions

We hope that this thesis has advanced current knowledge on intrusion detection by providing insights into representation and recognition issues of user behavior. Here we are going to highlight the main contribution of this thesis.

At the beginning of this work we have introduced a hybrid architecture for intrusion detection. The architecture employs two approaches: the anomaly and misuse intrusion detection, trying to benefit from their advantages through their cooperation. It also employs the network agents (Zamboni *et al.*, 1998) and is able to distribute detection tasks over the network. The anomaly detector exposes anomalies by profiling user behavior. The misuse detector provides fast recognition of malicious actions that have been exposed in the past.

A new information representation method that translates basic audit trail log events into a more general and, therefore a more understandable and source-independent form was developed in this thesis. The method is based on the assumption that the user's behavior includes regularities, which can be detected and coded as a number of patterns. The information derived from these patterns could be used to detect the abnormal behavior and to train the system.

Two new anomaly intrusion detection approaches were suggested. They dynamically model the patterns of user behavior; they are independent of the system, type of input, and the specific intrusions to be monitored. These approaches are based on the new information representation method. They use temporal-probabilistic trees to represent and monitor a users' behavior. In particular they are able to automatically find the normal behavior patterns from the audit data, encode and match them against the current event streams in order

to observe deviations from normal behavior; then analyze and decide whether it is an intrusion or normal behavior changes. The concept of temporal pattern learning for their further usage is a central part of this thesis and it is one of the main contributions of the thesis.

The developed anomaly detection approach translates the main decision problem into three scenarios: *what* is happening - this is the problem of identification of different aspects of a user's behavior; *when?* - in order to answer to this question we try to observe the temporal aspects of a user's behavior (when the activity is happening and how long) by analyzing and tracing it in its temporal context using temporal algebra (Hirsch, 1996) to describe relationships between temporal intervals or actions; and *how much?* - would help us to determine the possible danger. The mechanism of trustworthiness is based on pattern deviation punishments and is controlled by coefficients of reliability, which are assigned to every activity. Therefore, the answer to these three questions gives us the possibility to capture more behavior aspects, which leads to additional flexibility in managing information stored in profiles.

Since our anomaly detection approaches are based on machine learning techniques there is a theoretical possibility that their profiles can be maliciously trained. In order to solve this problem the system must be able to differentiate abnormal learning attempts from a concept drift. We have considered this issue in the thesis and devised methods to efficiently handle the concept drift and detect the undesirable learning attempts (intellectual attacks).

Based on the HIDSUR architecture and employing the developed anomaly detection methods a prototype was built. Using this prototype different experiments were conducted. They provided results and gave us the opportunity to discuss performance of different parts of the prototype and compare it to other systems.

The main goals of our work have been to adequately represent and efficiently recognize users (or distinguish among them) rather than attempt to represent and detect every conceivable intrusion scenario. We have been successful in achieving these goals. According to our experiments we could represent typical user behavior and recognize it correctly no less than in 92% of cases using the temporal-probabilistic tree approach, and in 83% using the relational matrix approach.

In our experiments all unrecognized (false negative) cases could be divided into two categories: sequence issued by a legitimate user and the system recognizes it as being issued by another user (type 1) or the system reports it as not belonging to any known user (type 2). We had none of the type 1 errors; all false rejections in our work can be classified as type 2. This means that our methods sometimes fail to handle fast user behavior changes. In our future work we are going to concentrate on this problem.

Based on the prototype that we implemented for our approaches, we derived the performance results that show that the approach presented in this the-

sis to discover, and represent behavioral patterns, and later use them for intrusion detection is practical. Anomaly intrusion detection systems have not been widely used mainly because of their space requirements and performance impact. In this work we have shown that it is possible to efficiently use an anomaly intrusion detection system based on pattern matching. Extrapolation of our experimental results show that the overhead of monitoring 10 000 users should be under 1% of the system CPU performance.

In this thesis we also have suggested a new way of organizing an intrusion detection system based on anomaly detection keeping false alarms at a relatively low level, having a high detection probability, performing with the same speed as misuse intrusion detection systems, and being able to recognize unknown attacks.

The HIDSUR architecture consists of two detectors: anomaly and misuse. The anomaly detector exposes anomalies basing it on the user behavioral model stored in a profile. The misuse detector was introduced into HIDSUR because it has a significantly shorter time-to-alarm interval. To avoid as much as possible human supervision than is necessary for all misuse detection systems, we have presented an idea of how to use the anomaly detector as a pattern supplier for the misuse detector. It has shown that it is possible to automatically create and use intrusion patterns. However, this part of the thesis requires additional research efforts in order to develop the method acceptable for widespread use.

Finally, we believe that this thesis provided new approaches to intrusion detection and hope that it will spawn further work in this direction.

## 10.2 Directions for Future Work

Our work presented in this thesis can be continued further in the following directions.

*Prototype Optimization*

It is necessary to investigate a possibility to optimize the prototype. It was implemented to prove the "viability" of the methods and models described in this thesis, therefore we may achieve better performance and accuracy results after optimizing it. By optimization we mean making the functions inline where possible to minimize function calls, embedding a classifier code into the kernel to avoid unnecessary disk swapping, eliminate code duplications, etc.

*Implement Other Features*

In this work we built a prototype as a concept proof. Using the prototype we have performed different tests to determine how our main approaches (temporal-probabilistic tree and relational matrix) perform in real conditions. However, to test other ideas described in this thesis we built a custom environment for each

experiment. Therefore, it would be beneficial to implement those parts in the prototype and test them in a real environment.

In most cases of successful intrusions the investigation reveals some facts. Such as, it is not enough for an intruder to know the operating systems and have excellent programming skills, he should also have some luck, i.e. ”be in the right place at the right time”.

According to a vulnerability analysis (Krsul, 1998) there are many factors that promote successful intrusions, such as: software bugs, failures, administrator and user mistakes or negligence. The ordinary user may stand behind these factors which are possibly caused by his actions. Therefore, a legitimate person using an operating system is able to create breaches in the system, which may be successfully used by intruders.

The user can do these actions for a purpose - in this case the system administrator has to suppress this with the help of an intrusion detection system. What if the user does not have any idea about what he/she is doing in the sense of making the system vulnerable? Most users have a very weak conception of computer security. Any ordinary and every day program that is used does not posses a security risk, but may accidentally be misconfigured by someone and later used by a malicious user to get additional privileges. It would be practical for the system to detect these cases, also since it would be possible not only to detect an intrusion, but also to prevent it before it happens.

In user recognition we rely on previous ”good” actions, but what if the user systematically runs different hackers’ programs (for example, for password cracking, denial-of-service attacks, etc.) and the system on the learning stage has accepted and encoded these actions as a part of normal behavior? To solve this problem the system should consider for classification of new actions, not only the user’s previous behavior, but also how the system responded to the previous behavior (system call activity, input-output operations, etc.). In other words it is necessary to combine the user-oriented view (take into account actions performed by a user) with system-oriented (system’s response to user’s actions), since the user can cause potential damage without knowing it. For these purposes we have introduced *system profile* into the architecture of our system at Figure 7.3, which has to be developed further.

*Distributed Classification*

As it is possible to see from the performance results in Chapter 9 that the performance overhead for the classification is very low. Therefore, it may be reasonable to completely distribute the classification task without devoting a separate server for it. Each host agent will track a local user(s) behavior and at the same time compare it with profile(s) (also locally) only sending alarms to a system administrator. This kind of system will not require a separate server and also the amount of sensitive information sent over the network will be minimized.

*Fuzzy Sets Approach for Instances Separation*

After testing the presented in this work approaches for user behavior classification we obtained quite promising results. However, one possible reason for this is that the user classes did not have many instances inside or/and these instances were situated far from each other, which made it easy for the system to separate instances inside each class. There is a potential risk that in some user groups the instances will be situated close to each other, which, in turn, may lead to a misclassification rate growth, since if two or more distributions overlap each other it is difficult to automatically establish fixed cut-off points for a decision. Also a fixed cut-off point technique does not correctly handle cases in overlapped areas. Thus, for future work we propose to use a fuzzy sets approach to separate instances inside each class. Each class will have its own thresholds on each classification step. This will make the classification process more flexible and precise and in a way guarantee that the system will perform equally well on different user groups.

*Kernel Matching*

It might be possible to embed a highly optimized custom sensor and classifier inside the kernel on a local workstation. This means that system call invocations no longer need to write audit data to disk. Instead, required information is collected in the kernel, at the point of call, and goes directly to the intrusion detection system. This results in an intrusion detector and classifier that can be truly real-time, because it removes the latency between the occurrence of an event and its notification to the detector. This latency may occur because of different reasons such as disk cache, system failures, sending it over the network, etc.

The implementation procedure of the detector is also important. It may be possible to implement learning, matching, etc. procedures as a shared library. Every profile may be constructed as a separate program that controls itself. In this case the program will call required procedures from a shared library. This will give us the following advantages:

- the possibility to process distributed data and thus, minimize the time of information flow between sensors and detector,
- protection of intrusion detection system's procedures from unauthorized modifications by storing them on read-only storage,
- the possibility to manipulate active profiles (programs) from the common control center,
- allow the program migration over the network (for example if the server is busy, it may send some programs to another server in order to avoid degradation of intrusion detection performance).

*Additional Protection*

Chapter 3 describes an architecture of our intrusion detection system. As can be seen in Figures 3.2 and 3.3 that there are different components which our intrusion detection system consist of. Those components are distributed over a network and, thus, they need to exchange information between each other. Therefore, some additional work has to be done to ensure their immunity to denial of service attacks and physical modifications on local workstations or in a network.

*User Interface*

It would be beneficial to provide a GUI interface tool that is able to visualize a chosen temporal-probabilistic tree and transactions happening in it. This would assist users in understanding and controlling the processes that control a user profile.

*Enhance Reporting Capabilities*

Since our prototype was built to test our ideas, we did not explicitly consider advanced reporting capabilities as a distinguished requirement of the prototype. However, a carefully designed reporting module is an important part of an intrusion detection system. It should not create many messages, but every issued message has to contain enough information for a system administrator.

*Multiple User Identity Handling*

A user may log in on the same network from different hosts and using different identities. It would be useful if our system can attribute multiple user session activities to the same user. As the user opens a new session from some host his identity should be tracked and attributed to the unique identity (for example, network user identity NID (Ko *et al.*, 1993)). Also, temporal patterns of user behavior may depend on the workstation's hardware. For example, if a user logs in from a different workstation, which is faster (or slower) than the one he/she normally uses, then the temporal lengths of his/her actions and relations between them may change simply because the software, he/she uses, performs differently on this particular hardware. As a result a system may produce false alarms. Therefore, it is necessary to investigate the possibility of taking this into account in order to avoid such cases.

*Investigate Privacy Issues*

In this work we did not pay much attention to privacy issues. We collected detailed user information to build user profiles. For our experiments we have impersonalized all obtained information by removing user names from it. How-

ever, if we consider such a system working in real life we can see that it is possible for a system administrator to trace back all user activities, being able to see when and what each particular user was doing. Thus, it would be beneficial to devise a method to protect a users' privacy. It is necessary to tokenize or encode user behavior information in a way that it would be impossible to trace back the users' activities, but still possible to discover and use behavioral patterns for online user verification.

*Investigate Applicability of our Approaches to Other Problems*

Described in this work the temporal-probabilistic tree represents a model of user behavior, thus, it probably can be used for solving other problems than intrusion detection. For example, in mobile networks there is an important task to anticipate what service a user will request during the next connection of his mobile terminal to a server. It is possible to create a profile for every user during a learning period where sequential and temporal patterns of his behavior are stored. The sequential patterns allow the prediction of certain probability to what kind of service the user may request next. Using the temporal patterns it is possible to calculate a time interval during which this service is most likely to be requested, and as a result a server may prepare and upload the service or information to a mobile terminal during the next connection making it unnecessary to establish a new connection when the user actually requests the service or information. This technique minimizes the number of sessions between a mobile terminal and a server, or/and shorten the length of each connection.

# YHTEENVETO (FINNISH SUMMARY)

Yhteiskuntamme on tullut yhä riippuvaisemmaksi tiedon nopeasta saannista ja käsittelystä, joita tuetaan aikaisempaa suurempien tietomäärien tietokoneavusteisella tallennettamisella ja käsittelyllä. Maailmanlaajuinen verkostoituminen mahdollistaa aikaisempaa nopeamman laajojen ja monipuolisten tietojen saannin lähes mistä tahansa maailmassa. Tietokonelaitteiden ja -verkkojen hintojen nopea lasku ja yleistyminen ovat samalla myös lisänneet tietojen luvattoman käytön ja muuntelun riskejä.

Kaikesta kehityksestä huolimatta tietokonejärjestelmät eivät ole täysin suojattuja useista syistä johtuen. Näitä ovat esimerkiksi laitteiden ja ohjelmistojen toimintavirheet, ohjelmistoihin alunperin sisältyvät puutteet ja ohjelmointivirheet sekä tietokonejärjestelmien käytössä ja ylläpidossa tapahtuvat virheet. Koska täydellisen tietoturvan saavuttaminen ei ole vielä näköpiirissä on edelleenkin välttämätöntä etsiä keinoja tietomurtojen paljastamiseksi. Sen vuoksi tarvitaan keinoja tunkeutujan ja tunkeutumisen tunnistamiseksi ja todistusaineiston keräämiseksi myös mahdollista oikeudellista jatkokäsittelyä varten.

Työn aihealueena olevat tunkeutumisen tunnistamisjärjestelmät on kehitetty palvelemaan nimenomaan tätä tarkoitusta ja niiden voidaan katsoa yleisesti kuuluvan tietokonejärjestelmän suojaustoimenpiteiden viimeiseen puolustuslinjaan. Nämä tunnistamisjärjestelmät ovat käyttökelpoisia sekä onnistuneiden tietomurtojen tunnistamisessa että tietomurtoyritysten tietojen kokoamisessa uusien vastatoimenpiteiden suunnittelua varten. Täten tunkeutumisen tunnistamisjärjestelmät ovat käyttökelpoisia silloinkin kun käytössä on vahvoja, korkean turvallisuustason tarjoamia suojamekanismeja.

Tunkeutumisen tunnistamisjärjestelmät voidaan jakaa kahteen päätyyppiin: väärinkäytön tunnistamiseen, jossa tunnistaminen perustuu siihen, että tunkeutuminen noudattaa ennalta tarkasti tunnettua menettelytapaa, sekä käyttötavan tunnistamiseen, jossa tunnistaminen perustuu käyttäjän tavanomaisesta poikkeavaan tapaan käyttää tietokonejärjestelmää. Väitöskirjan alussa esitetään tunkeutumisen tunnistamisjärjestelmän hajautettu arkkitehtuuri, joka yhdistää molempien päätyyppien piirteitä pyrkien hyödyntämään niiden kummankin vahvuuksia. Arkkitehtuurin osalta työ keskitty käyttötapaan perustuvaan tunnistamisosuuteen. Väärinkäyttöön perustuvaa tunnistamista käsitellään vain esittäen kuinka käyttötapaan perustuvalla tunnistamisella voitaisiin tuottaa sen tarvitsemia tarkasti tunnettuja tunkeutumistapoja.

Väitöskirjassa esitetään käyttäjien käyttötapojen eksplisiittisen esityksen muodostamista ja siihen pohjautuvaa tunnistamista tunkeutumisen tunnistamisongelman ratkaisuksi. Tällöin keskeiseksi ongelmaksi muodostuvat dynaamisen toimintaympäristön tapahtumien säännönmukaisuuksien tulkinta ja niiden vertaaminen käyttötapakuvauksiin. Väitöskirjatyössä ei pyritä kaikkien mahdollisten tunkeutumisskenaarioiden esittämiseen ja paljastamiseen vaan

työn keskeiseksi tavoitteeksi on rajattu käyttötapaan perustuva käyttäjän tehokas tunnistaminen ja sen kannalta riittävän käyttötapaesityksen muodostaminen.

Työssä käyttäjän ja tietokonejärjestelmän välisen vuorovaikutuksen katsotaan ylimmällä tasolla koostuvan *toimista*, joita käyttäjä suorittaa saavuttaakseen tavoitteensa. Nämä toimet koostuvat käyttäjän peräkkäin suorittamista *toimenpiteistä*. Kukin toimenpide edelleen aiheuttaa käyttöjärjestelmätasolla sarjan *tapahtumia*. Väitöskirjassa kehitetty tunkeutumisen tunnistamisjärjestelmä perustuu sille oletukselle, että käyttäjät noudattavat toimissaan heille kullekin luonteenomaisia peräkkäisiä toiminpiteitä, joiden ominaispiirteiden perusteella tunkeutuja voidaan havaita käyttäjäjoukosta.

Työssä on kehitetty käyttötapa tiedon esittämiseksi kolmetasoinen ratkaisu vastaten käyttäjän ja tietokonejärjestelmän vuorovaikutuksen jäsentämistä toimiin, toimenpiteisiin ja tapahtumiin. Alimmalla tasolla tapahtumat rekisteröidään lokitiedostossa käytetyllä yksityiskohtaisella tarkkuudella. Tapahtumien pohjalta johdetaan paikka- ja laiteriippumattomat keskimmäisen tason käyttäjän toimenpidekuvaukset, jotka vastaavat sitä ”mitä käyttäjä tekee”. Toimenpidekuvausten pohjalta johdetaan ylimmän abtraktiotason ohjelmistoriippumaton kuvaus siitä ”mihin tavoitteisiin käyttäjä pyrkii”. Ylimmän tason ohjaamana kootaan sitten yhteen tietoa käyttäjän käyttötavasta, ts. vastaamaan kysymykseen ”miten toimien käyttäjä normaalisti saavuttaa tavoitteensa”.

Väitöskirjatyössä esitetty tunkeutumisen tunnistamisjärjestelmä perustuu käyttäjän käyttötavan esittämiseen todennäköisyyksiä hyödyntävänä verkkokuvauksena järjestelmä-, syöte- ja tunkeutumistapariippumattomasti. Loogiselle verkkokuvaukselle esitetään kaksi toteutustapaa: taulukkoesitys sekä aika- ja todennäköisyysarvoja sisältävä puuesitys. Tunnistamisvarmuutta arvioidaan käyttäen luotettavuuskertoimia, joiden arvoja havaittujen tapahtumien ja tallennettujen käyttötapojen erojen perusteella lisätään tai vähennetään.

Työssä kehitetty tunkeutumisen tunnistamistapa perustuu koneoppimisen tekniikoihin ja niinpä on olemassa teoreettinen mahdollisuus sille, että käyttötapaprofiileja yritetään harkitusti opettaa hyväksymään tunkeutumisen sisältävä käyttötapa normaaliksi käyttötavaksi (niin kutsuttu älykäs hyökkäys). Työssä on tarkasteltu paitsi menetelmiä normaalien käyttötapamuutosten tekemiseksi myös niiden erottamiseksi harkitusta käyttötapamuutoksesta, joka mahdollisesti tähtää myöhemmin tapahtuvaan tunkeutumiseen.

Väitöskirjatyön osana on toteutettu esitettyyn arkkitehtuuriin ja kehitettyyn käyttötapapohjaiseen tunnistamiseen perustuva järjestelmäprototyyppi, joka on toteutettu tarkoitusta varten pystytetyssä testi ympäristössä. Toteutuksessa käytettiin Linux RedHat7.1 käyttöjärjestelmää ja prototyyppi rakennettiin C++ kieltä käyttäen jolloin prototyyppiohjelmiston kooksi muodostui hieman yli 25 000 ohjelmariviä. Prototyypillä suoritetuissa kokeiluissa havaittiin käytetyllä aineistolla saavutettavan vähintäänkin 92% onnistuminen käyttäjän käyttötavan tunnistamisessa silloin kun käytettiin aika- ja todennäköisyysarvoja

sisältävää puuesitystä ja 83% onnistuminen silloin kun käytettiin relaatioperusteista taulukkoesitystä.

Käyttötapaan perustuvia tunkeutumisen tunnistamisjärjestelmiä ei ole yleisesti kovin laajasti käytetty johtuen niiden tilavaatimuksista ja alentavasta vaikutuksesta tietokonejärjestelmän suoritustehoon. Väitöskirjatyössä sovelletun lähestymistavan kokeilujen pohjalta ekstrapoloitu 10 000 käyttäjän käyttötavan seuranta näyttäisi vaativan vain alle 1% tietokonejärjestelmän keskusyksikön suorituskyvystä ja näinollen näyttäisi olevan myös käytännössä mahdollista toteuttaa käyttötapaan perustuva tunkeutujan tunnistamisjärjestelmä. Saavutettujen positiivisten tulosten valossa näyttäisikin olevan mielekästä jatkaa ponnisteluja käyttötapaan perustuvan lähestymistavan kehittämiseksi osaksi tietoturvajärjestelmiä.

## Bibliography

Acid, S. and de Campos, L. 1996, Benedict: An algorithm for learning probabilistic belief networks, *in 4th Conference of Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Granada, Spain, pp. 979–984.

Aha, D., Kibler, D. and Albert, M. 1991, Instance-based learning algorithms, *Machine Learning* **6**(1), 37–66.

Albrecht, D., Zukerman, I., Nicholson, A. and Bud, A. 1997, Towards a Bayesian model for keyhole plan recognition in large domains, in A. Jameson, C. Paris and C. Tasso (eds), *in 6th International Conference on User Modeling*, Springer, New York, USA, pp. 365–376.

Ali, S. 2000, Adventures in anomaly detection, *in 3rd International Workshop on Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science, Springer Verlag, Toulouse, France.

Allen, J. 1983, Maintaining knowledge about temporal intervals, *Communications of the ACM* **26**(11), 832–843.

Allen, J., Christie, A., Fithen, W., McHugh, J., Pickel, J. and Stoner, E. 1999, *State of the practice of intrusion detection technologies*, Technical report CMU/SEI-99-TR-028, Carnegie Mellon Software Engineering Institute.

Allen, J. and Ferguson, G. 1994, *Actions and events in interval temporal logic*, Technical Report TR521, Computer Science Department, Rochester.

Anderson, D., Lunt, T., Javitz, H., Tamaru, A. and Valdes, A. 1995, *Safeguard final report: Detecting unusual program behavior using the NIDES statistical component*, Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, USA.

Anderson, J. 1980, *Computer security threat monitoring and surveillance*, Technical Report 79F296400, James P. Anderson Co., Fort Washington, Pennsylvania.

Avizienis, A., Laprie, J.-C. and Randell, B. 2001, *Fundamental concepts of dependability*, Technical report 01145, LAAS: Laboratory Analysis and Architecture Systems, France.

Baur, A. and Weiss, W. 1988, *Audit trail analysis tool for system with high demands regarding security and access control*, Technical Report ZFE F2 SOF 42, Siemens Nixdorf Software, Germany.

Beetle, S. 2000, A strict anomaly detection model for IDS, *Phrack Magazine,* Available from *http://www.phrack.com/show.php?p=56&a=11,* **10**(56).

Bellazzi, R., Magni, P. and De Nicolao, G. 1998, Bayesian function learning using MCMC methods, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **20**(12), 1319–1331.

Bird, R. 1988, *The Professional Programmers Guide to UNIX*, Pitman Publishing, London.

Borden, V. 2002, Course of Statistics, Available from *http://www.imir.iupui.edu/psyb305/week4.htm*, [Read 3.05.2002].

Bugtraq electronic mailing list 1999, Available at: *http://www.securityfocus.com*, [Read 12.02.1999].

Carpenter, G., Grossberg, S., Markuzon, N., Reynolds, J. and Rosen, D. 1992, Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps, *IEEE Transactions on Neural Networks* (3), 698–713.

CERT advisories 1999, Available by ftp from *ftp://cert.sei.cmu.edu,* [Read 12.02.1999].

Charniak, E. 1991, Bayesian networks without tears, *AI Magazine* **12**(4), 50–63.

Cheeseman, P., Hanson, R. and Stutz, J. 1991, Bayesian classification with correlation and inheritance, in M. Kaufmann (ed.), *in 12th International Joint Conference on Artificial Intelligence*, Vol. 2, Sydney, Australia, pp. 692–698.

Cheeseman and Stutz, J. 1995, *Bayesian Classification (AutoClass): Theory and Results*, AAAI Press, Menlo Park, California, USA, pp. 152–180.

Chen, K. 1988, *An Inductive Engine for the Acquisition of Temporal Knowledge*, Phd thesis, University of Illinois, USA.

CNN.com/SCI-TECH 2002, FBI warns companies about Russian hacker attacks, Available from *http://www.cnn.ru/2001/TECH/internet/03/08/hacker.attacks/index.html*, [Read 7.05.2002].

computerheadline.com 2002, Gain competitive advantage by placing security high up the corporate agenda, Available from *http://www.computerheadline.com/BusinessMatters/February2002/computerassoc.asp*, [Read 7.05.2002].

Crosbie, M., Dole, B., Ellis, T., Krsul, I. and Spafford, E. 1996, *IDIOT-user's guide*, Technical Report TR-96-050, COAST Laboratory, Purdue University, USA.

Crosbie, M. and Spafford, G. 1995, *Active defense of a computer system using autonomous agents*, Technical Report TR-95-008, COAST Laboratory, Purdue University, USA.

Davison, B. and Hirsh, H. 1998, Predicting sequences of user actions, *in Proceedings of AAAI-98/ICML-98 Workshop, published as Technical Report WS-98-07*, AAAI Press, Madison, WI, pp. 5–12.

Debar, H., Becker, M. and Siboni, D. 1992, A neural network component for an intrusion detection system, *in IEEE Symposium of Research in Computer Security and Privacy*, IEEE Computer Society Press, Oakland, California, USA, pp. 240–250.

Denning, D. 1982, *Cryptography and Data Security*, Addison-Wesley, Massachusetts, USA.

Denning, D. 1987, An intrusion detection model, *IEEE Transactions on Software Engineering* **13**(2), 222–232.

Durst, R., Champion, T., Witten, B., Miller, E. and Spagnuolo, L. 1999, Testing and evaluating computer intrusion detection systems, *Communications of the ACM* **42**(7), 53–61.

Eckmann, S., Vigna, G. and Kemmerer, R. 2001, Statl: An attack language for state-based intrusion detection, *Journal of Computer Security* .

Eskin 2000, Anomaly detection over noisy data using learned probability distributions, *in 17th International Conf. on Machine Learning*, Morgan Kaufmann, San Francisco, CA, pp. 255–262.

Focardi, R. and Gorrieri, R. (eds) 2000, *Foundations of security analysis and design*, Lecture Notes in Computer Science 2171, Springer, Berlin, Germany.

Fox, K., Henning, R., Reed, J. and Simonian, R. 1990, A neural network approach towards intrusion detection, *in The 13th National Computer Security Conference*, National Institute of Standards and Technology, National Computer Security Center, Washington D.C., USA, pp. 125–134.

Garfinkel, S. and Spafford, G. 1991, *Practical Unix Security*, O'Reilly and Associates, Sebastopol, California, USA.

Garvey, T. and Lunt, T. 1991, Model-based intrusion detection, *in 14th National Computer Security Conference*, National Institute of Standards and Technology, National Computer Security Center, Washington D.C., USA, pp. 372–385.

GCC manual 2000, Available from *http://gcc.gnu.org/*, [Read 12.02.1999].

Gent, C. and Sheppard, C. 1992, Predicting time series by a fully connected neural network trained by back propagation, *Computing and Control Engineering Journal (May issue)*, pp. 109–112.

Giarratano, J. 1992, *Clips version 5.1 user's guide*, User guide, NASA, Information Systems Directorate, Software Technology Branch.

Heady, R., Luger, G., Maccabe, A. and Servilla, M. 1990, *The architecture of a network level intrusion detection system*, Technical Report CS90-20, Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, USA.

Heberlein, L., Levitt, K. and Mukherjee, B. 1991, A method to detect intrusive activity in a networked environment, *in The 14th National Computer Security Conference*, National Institute of Standards and Technology, National Computer Security Center, Washington D.C., USA, pp. 362–371.

Heberlein, T. and Bishop, M. 1998, *Attack Class: Address Spoofing*, Addison-Wesley Pub Co.

Hirsch, R. 1996, Relation algebra of intervals, *Artificial Intelligence* **83**(2), 267–295.

Hochberg, J., Jackson, K., Sttallins, C., McClary, J., DuBois, D. and Ford, J. 1993, NADIR: an automated system for detecting network intrusion and misuse, *Computer & Security* **12**(3), 235–248.

Holsheimer, H. and Siebes, A. 1994, *Data mining: the search for knowledge in databases.*, Technical Report CS-R9406, CWI: Department of Algorithms and Architecture.

Ilgun, K., Kemmerer, R. and Porras, P. 1995, State transition analysis: A rule-based intrusion detection approach, *IEEE Transactions on Software Engineering* **21**(3), 181–199.

ISO 1991, International organization of standardization: Information technology: Open systems interconnection: Security frameworks for open systems, (ISO/IEC DIS 10181).

Javitz, H., Valdes, A., Lunt, T., Tamaru, A., Tyson, M. and Loerance, J. 1993, *Next generation intrusion detection expert system (NIDES)*, Technical Report A016-Rationales, Computer Science Laboratory, SRI International, Menlo Park, California, USA.

Jensen, C., Dyreson, C., Böhlen, M., Clifford, J., Elmasri, R., Gadia, S., Grandi, F., Hayes, P., Jajodia, S., Käfer, W., Kline, N., Lorentzos, N., Mitsopoulis, Y., Montanari, A., Nonen, D., Peressi, E., Pernici, B., Roddick, J., Sarada, N., Scalas, R., Segev, A., Snodgrass, R., Soo, M., Tansel, A., Tiberio, P. and Wiederhold, G. 1998, The consensus glossary of temporal database concepts - february 1998 version, *Temporal Databases - Research and Practice* **1399**, 367–405.

Kautz, H. and Ladkin, P. 1991, Integrating metric and qualitative temporal reasoning, *in 9th National Conference of Artificial Intelligence*, American Association for Artificial Intelligence, Anaheim, CA, USA, pp. 241–246.

Ko, C., Frincke, D., Goan, T., Herberlain, L., Mukherjee, B. and Wee, C. 1993, Analysis of an algorithm for distributed recognition and accountability, *in 1st ACM Conference on Computer Communication Security*, ACM , USA, pp. 154–164.

Kondratoff, Y. and Michalski, R. 1990, *Machine Learning*, Vol. 3, Morgan Kaufmann, San Mateo, CA, USA, pp. 611–638.

Krishnan, P. 1995, *Online Prediction Algorithms for Databases and Operating Systems*, Phd thesis, Brown University.

Krsul, I. 1998, *Software Vulnerability Analysis*, Phd thesis, Purdue University.

Kumar, S. 1995, *Classification and Detection of Computer Intrusions*, Phd thesis, Purdue University.

Kumar, S. and Spafford, E. 1994, A pattern matching model for misuse intrusion detection, *in 17th National Computer Security Conference*, National Institute of Standards and Technology, National Computer Security Center, Washington D.C., USA, pp. 11–21.

Kumar, S. and Spafford, E. 1995, *A software architecture to support misuse intrusion detection*, Technical Report CSDTR -95-009, Department of Computer Sciences, Purdue University.

Lane, T. and Brodley, C. 1997a, An application of machine learning to anomaly detection, *in 20th Annual National Information Systems Security Conference*, Vol. 1, National Computer Security Center, Baltimore, Maryland, USA, pp. 366–380.

Lane, T. and Brodley, C. 1997b, *Detecting the abnormal: Machine learning in computer security*, Technical Report ECE-97-1, Department of Electrical and Computer Engineering, Purdue University.

Lane, T. and Brodley, C. 1999, Temporal sequence learning and data reduction

for anomaly detection, *ACM Transactions on Information and System Security* **2**(3), 295–331.

Lee, J., Mccartney, R. and Santos, E. 2000, Learning Predictive Patterns of User Resource Usage, Available from *http://citeseer.nj.nec.com/article/lee00learning.html*, [Read 02.06.2002].

Lee, W., Stolfo, S. and Mok, K. 1998, Mining audit data to build intrusion detection models, *in ACM SIGKDD 4th International Conference on Knowledge Discovery and Data Mining*, ACM, New York, NY, USA, pp. 66–72.

Lee, W., Stolfo, S. and Mok, K. 1999a, A data mining framework for building intrusion detection models, *in IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, CA, USA, pp. 120–132.

Lee, W., Stolfo, S. and Mok, K. 1999b, Mining in a data-flow environment: Experience in network intrusion detection, *in 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, San Diego, CA, USA, pp. 114–124.

Lee, W. and Xiang, D. 2001, Information-theoretic measures for anomaly detection, *in 2001 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, California, USA, pp. 130–143.

Liepins, G. and Vaccaro, H. 1989, in anomaly detection: Purpose and framework, *12th National Computer Security Conference*, pp. 495–504.

Lindqvist, U. 1999, *On the Fundamentals of Analysis and detection of computer Misuse*, Phd thesis, Department of Computer Engineering, University of Göteborg, Sweden.

Longley, D. and Shain, M. 1987, *Data and Computer Security: Dictionary of Standards, Concepts and Terms*, Stockton Press, New York.

Lowe, G. 1996, some new attacks upon security protocols, *in 9th computer Security foundations workshop*, IEEE Computer Press, Kenmare, County Kerry, Ireland, pp. 139–146.

Lunt, T. 1993, Detecting intruders in computer systems, *in 6th Conference on Auditing and Computer Technology*, Canada.

Lunt, T., Jagannathan, R., Lee, R., Whitehurst, A. and Listgarten, S. 1989, Knowledge based intrusion detection, *in Annual AI Systems in Government Conference*, Washington, DC, USA, pp. 102–107.

Lunt, T., Tamaru, A., Gilham, F., Jagannathan, R., Neumann, P., Javitz, H., Valdes, A. and Garvey, T. 1992, *A real-time intrusion detection expert system (IDES) - final technical report*, Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, USA.

Melissa Virus 1999, Description available from: *http://www.melissavirus.com*, [Read 12.03.2002].

Mell, P. 1999, Computer Attacks: What They Are and How to Defend Against Them, Available from *csrc.nist.gov/staff/mell/compattack.pdf*, [Read 7.05.2002].

Michael, C. and Ghosh, A. 2000, Two state-based approaches to program-based

anomaly detection, *in Annual Computer Security Applications Conference*, New Orleans, Louisiana, USA.

Mitchel, T. 1997, *Machine Learning*, WCB.McGraw-Hill, USA.

Mounji, A. 1997, *Languages and Tools for Rule-Based Distributed Intrusion Detection*, Phd thesis, Namur, Belgium.

Nilsson, N. 1998, *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publishers, Inc., San Francisco, California, USA.

Porras, P. and Neumann, P. 1997, EMERALD: event monitoring enabling responses to autonomous live disturbances, *in 20th National Information Systems Security Conference*, National Institute of Standards and Technology/National Computer security Center, Baltimore, Maryland, USA, pp. 353–365.

Powell, D., Adelsbasch, A., Cachin, C., Creese, S., Dacier, M., Deswarte, Y., McCutcheon, T., Neves, N., Pfitzmann, B., Randell, B., Stroud, R., Verissimo, P. and Waidner, M. 2001, MAFTIA (Malicious- and Accidental- Fault Tolerance for Internet Applications), *in International conference on dependable systems and networks*, IEEE Computer Press, Göteborg, Sweden, pp. 32–35.

Power, R. 1998, *Current and Future Danger: A CSI Primer on Computer Crime and Information Warfare*, Computer Security Institute, San Francisco, California, USA.

Power, R. 2002, 2002 SCI/FBI computer crime and security survey, *Computer Security Issues & Trends* **8**(1).

Puketza, N., Zhang, K., Chung, M., Mukherjee, B. and Olsson, R. 1996, A methodology for testing intrusion detection systems, *IEEE Transactions on Software Engineering* **22**(10), 719–729.

Russel, D. and Gangemi, G. 1991, *Computer Security Basics*, O'Reilly&Associates Inc., Sebastopol, California, USA.

Schlimmer, J. 1987, *Concept acquisition through representational adjustment*, Phd thesis, University of California, Irvine.

Sebring, M., Shellhouse, E., Hanna, M. and Whitehurst, A. 1988, Expert systems in intrusion detection: a case study, *in 11th National Computer Security Conference*, Baltimore, Maryland, USA, pp. 74–81.

Seleznyov, A. 2000a, A hybrid model for intrusion detection, in S. Quing and J. Eloff (eds), *in 16th IFIP World Computer Congress*, Kluwer, Beijing, China, pp. 164–167.

Seleznyov, A. 2000b, Temporal-probabilistic networks in intrusion detection: Detecting abnormal learning, in S. Quing and J. Eloff (eds), *in 16th IFIP World Computer Congress*, Kluwer, Beijing, China, pp. 168–171.

Seleznyov, A. 2000c, Using temporal-probabilistic network approach for automatic pattern generation for misuse detection, in Y. Karsligil, G. Yavuz, E. Karsligil, T. Inan, B. Diri, E. Ergün and S. Albayrak (eds), *in The 15th International Symposium on Computer and Information Sciences*, Academy Yayincilik (Ankara), Istanbul, Turkey, pp. 366–373.

Seleznyov, A. 2001, A methodology to detect anomalies in user behavior basing on its temporal regularities, in M. Dupuy and P. Paradinas (eds), *in IFIP/SEC2001: 16th International Conference on Information Security*, Kluwer (USA), Paris, France, pp. 327–338.

Seleznyov, A. and Mazhelis, O. 2002, Learning temporal patterns for anomaly intrusion detection, *in ACM SAC 2002: 17th ACM Symposium on Applied Computing*, ACM, Madrid, Spain, pp. 209–213.

Seleznyov, A., Mazhelis, O. and Puuronen, S. 2000, Detecting abnormal behavior using temporal-probabilistic networks, in Y. Karsligil, G. Yavuz, E. Karsligil, T. Inan, B. Diri, E. Ergün and S. Albayrak (eds), *in The 15th International Symposium on Computer and Information Sciences*, Academy Yayincilik (Ankara), Istanbul, Turkey, pp. 495–503.

Seleznyov, A., Mazhelis, O. and Puuronen, S. 2001, Learning temporal regularities of user behavior for anomaly detection, in V. Gorodetski, V. Skormin and L. Popyack (eds), *in International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security*, LNCS 2052, Springer, St.Petersburg, Russia, pp. 143–152.

Seleznyov, A., Mazhelis, O. and Puuronen, S. 2002, An anomaly intrusion detection system based on online user recognition, in S. Furnell, and P. Dowland (eds), *in CD-ROM Proceedings of the 3rd International Network Conference*, Plymouth, UK.

Seleznyov, A. and Puuronen, S. 1999, Anomaly intrusion detection systems: Handling temporal relations between events, *in 2nd International Workshop on Recent Advances in Intrusion Detection*, Purdue, Lafayette, Indiana, USA.

Seleznyov, A. and Puuronen, S. 2000, HIDSUR: A hybrid intrusion detection system based on real-time user recognition, in A. Tjoa, R. Wagner and Al-Zobaidie (eds), *in 11th International Workshop on Database and Expert Systems Applications*, IEEE Computer Society Press, Greenwich-London, England, pp. 41–45.

Seleznyov, A., Terziyan, V. and Puuronen, S. 2000, Temporal-probabilistic network approach for anomaly intrusion detection, *in 12th Annual Computer Security Incident Handling Conference*, Chicago, USA.

Sendmail Mail Program 2000, Description and new version available from *http://www.sendmail.org*, [Read 17.04.2000].

Shieh, S. and Gligor, V. 1991, A pattern-oriented intrusion-detection model and its applications, *IEEE Transactions on Data and Knowledge Engineering* **9**(4), 661–668.

Smaha, S. 1988, Haystack: An intrusion detection system, *in 14th Aerospace Computer Security Applications Conference*, Tracor Applied Science Inc., Austin, Texas, USA, pp. 37–44.

Smaha, S. 1992, Questions about CMAD, *in Workshop on Future Directions in Computer Misuse and Anomaly Detection*, Davis, CA, USA, pp. 17–21.

Smaha, S. 1993, Tools for misuse detection, *in Annual Information Security Confer-*

*ence*, Information System Security Association, Crystal City, VA, USA.

Snapp, S. and Smaha, S. 1992, in signature analysis model definition and formalism, *4th Workshop on Computer Security Incident Handling*, Denver, Colorado, USA.

Snapp, S., Smaha, S., Teal, D. and Grance, T. 1992, The DIDS (distributed intrusion detection system prototype), *in USENIX Conference*, USENIX Association, San Antonio, Texas, USA, pp. 227–233.

Sobirey, M. 2002, Michael Sobirey's Intrusion Detection Systems Page , Available at: *http://www-rnks.informatik.tu-cottbus.de/∼sobirey/ids.html*, [Read 20.01.2002].

Song, D., Wagner, D. and Tian, X. 2001, Timing analysis of keystrokes and timing attacks on SSH, *in 10th USENIX Security Symposium*, Washington, D.C., USA.

Spafford, E. 1989, Crisis and aftermath, *Communications of the ACM* **32**(6), 678–687.

Spitzner, L. 2000, To build a honeynet, *in 12th Annual FIRST Conference*, Denver, Colorado, USA.

Stroustrup, B. 1991, *The C++ Programming Language*, Addison-Wesley.

Sundaram, A. 1998, *An Introduction to Intrusion Detection*, ACM Crossroads, Available from *http://www.acm.org/crossroads/xrds2-4/intrus.html*, [Read 1.05.2002].

Swets, J. 1988, Measuring the accuracy of diagnostic systems, *Science* **240**(4857), 1285–1293.

Tan, K. and Maxion, R. 2002, "Why 6?" Defining the operational limits of STIDE, *in 2002 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, California, USA, pp. 188–200.

Teng, H., Chen, K. and Lu, S. 1990, Security audit trail analysis using inductively generated predictive rules, *in 6th IEEE Conference of Artificial Intelligence Applications*, IEEE Computer Society Press, Piscataway, NJ, USA, pp. 24–29.

U.S.C. 2002, US Code Collection, Available from *http://www4.law.cornell.edu/uscode/18/*, [Read 20.05.2002].

Wee, C. 1998, Network delay as a IDS response, Available from *seclab.cs.ucdavis.edu/response/reports/*, [Read 10.06.2002].

Weinberg, N. 2001, Enterasys' IDS Dragon offers best performance for network based intrusion detection, Network World Newsletter, available from *www.ps.avnet.com/au/hallmark/bulletins/Bulletin_13.pdf*, [Read 10.06.2002].

Ye, N. and Chen, Q. 2001, An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems, *Quality and Reliability Engineering International* (17), 105–112.

Ylönen, T., Kivinen, T., Saarinen, M., Rinne, T. and Lehtinen, T. 2002, SSH Protocol Architecture, Available from *http://www.globecom.net/ietf/draft/draft-ietf-secsh-architecture-07.html*, [Read 7.05.2002].

Zamboni, D., Balasubramaniyan, J., Garsia-Fernandez, J., Isacoff, D. and Spaf-

ford, E. 1998, *An architecture for intrusion detection using autonomous agents*, CERIAS Technical Report TR9805, COAST Laboratory, Purdue University.

Zhang, Z., Li, J., Manikopoulos, C., Jorgenson, J. and Ucles, J. 2001, Neural networks in statistical intrusion detection, *in CD-ROM Proceedings of the 5th World Multi-Conference on Circuits, Systems, Communications and Computers*, Electrical and Computer Engineering International Reference Book, WSES Press, Crete Island, Greece.

# Appendix 1 Pattern Generation for Misuse Detection

In Section 2.4 we described the major limitations of existing intrusion detection approaches. Comparing two models: anomaly and misuse detection we can notice that anomaly detection systems try to detect the complement of ”bad” behavior, whereas misuse detection systems try to recognize already known types of ”bad” behavior (Sundaram, 1998). Nowadays anomaly detection systems use online learning techniques, which include several data analyzing levels to reduce the false alarm rate. Sometimes because of computational complexity the data is being analyzed during the night (or whenever the intrusion detection system has spare CPU time). Therefore, this multi-level data analysis often results in a significant delay between time of intrusion and its detection. Thus, it is reasonable to use misuse techniques together with anomaly detection, since the misuse intrusion detection system is able to perform real-time detection.

In a HIDSUR architecture we presented both misuse and anomaly detection models, that are used in a way to support each other. Once an intrusive sequence of events is detected by the anomaly detector, it is being analyzed and encoded as a pattern for misuse detection. If this kind of ”bad” behavior is present again in an input sequence of events the misuse detector detects it in real-time and the intrusion detection system is able to separate the intrusion and prevent possible future damage.

After the system is able to distinguish abnormal behavior sequences from natural behavior changes (concept drift), we can then store abnormal patterns for future real-time recognition of the same kind of ”bad” behavior. The abnormal sequence of events is analyzed by an ”automatic pattern generator”. All events are separated into groups according to relationships between them. Then events inside every group are analyzed and a coefficient of security significance is assumed for every group (it depends on the coefficient of every specific event in the group). All groups are separated by a threshold. Only groups (key groups), that have the coefficient of a security significance higher than a certain value, are taken into account. Finally, the events that are being contained by defined key groups are encoded into a pattern for the misuse detector.

## 1.1 System Architecture

In this section we describe the pattern generation process. We begin with a part of the architecture of the intrusion detection system’s description, where the pattern generation is involved. We only describe those components that directly take part in the described process. After this we provide definitions of notions that are necessary for pattern generation description, and describe the architecture of the pattern generation component with the involved algorithm.

The automatic pattern generator is implemented as a part of our hybrid model (Chapter 3). It is implemented as a connector between anomaly and misuse detectors. In Figure 1.1 it is possible to see part of our model that involves the automatic pattern generator. Since the system needs to wait for consistency checking and to have several levels of information analysis it may take a significant amount of time before the actual detection of an intrusion. Therefore, a misuse detector part is intended to make a real-time detection.

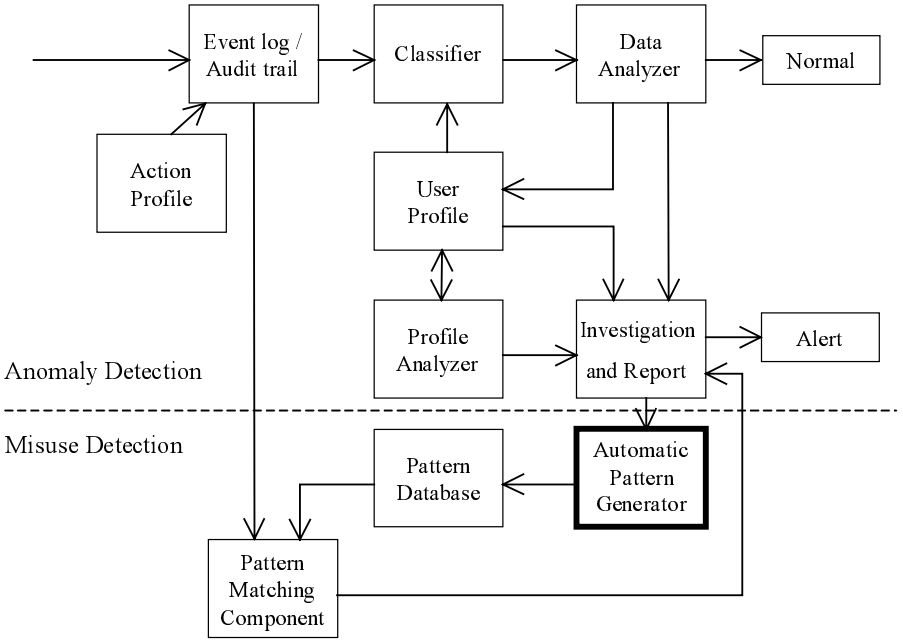


Figure 1.1 Part of HIDSUR with pattern generator

The abnormal activity detection is based on online learning to catch and encode and then update variations of users' behavior in the knowledge base. It takes as input a stream of events. The stream may include different kinds of events such as shell command input, GUI events, network packet traffic, system call traces, etc. This information is stored in the *event log* until it is needed for processing.

The *action profile* contains simple relationships between events, temporal algebra formulas, which events must satisfy to be generalized into actions. In other words a set of actions is defined here by defining events and necessary relationships between them. Also security significance coefficients are defined in the action profile for every possible action. These coefficients may be manipulated by the system when it detects an intellectual attack. If the coefficient of reliability only increases for a certain profile this means that this profile may be under attack. If the user has begun to use applications that are typical of him/her, but in a new and not peculiar to this user's way there is an intellectual attack suspicion and it needs to be rigorously investigated: firstly, by "investigation and report component" and, secondly, if necessary, by the system administrator. The system may manipulate the coefficients of security significance

and monitor future use of commands with higher coefficients of security significance. Thus, after a learning stage, the system maintains these coefficients for every user and for every event.

## 1.2 Pattern Generation Process

When the intrusion detection system detects an abnormal sequence, the main goal is to separate the *key events* for encoding them into a pattern. *Key event* is an event that may represent an important landmark of a certain attack. In other words a short enough sequence of related key events may describe a certain attack and be used for its recognition. A *key group* of events is a group where key events have strong relationships between each other (manipulations on the same file, etc.). Therefore, the main problem of pattern generation is an automatic separation of key events from an incoming sequence of events.

To demonstrate the pattern generation process we use an example of a penetration scenario (see Figure 1.2). Using a simple user session example (Figure 4.1) we injected a penetration scenario (the scenario has been described in Section 2.2.2) in it.

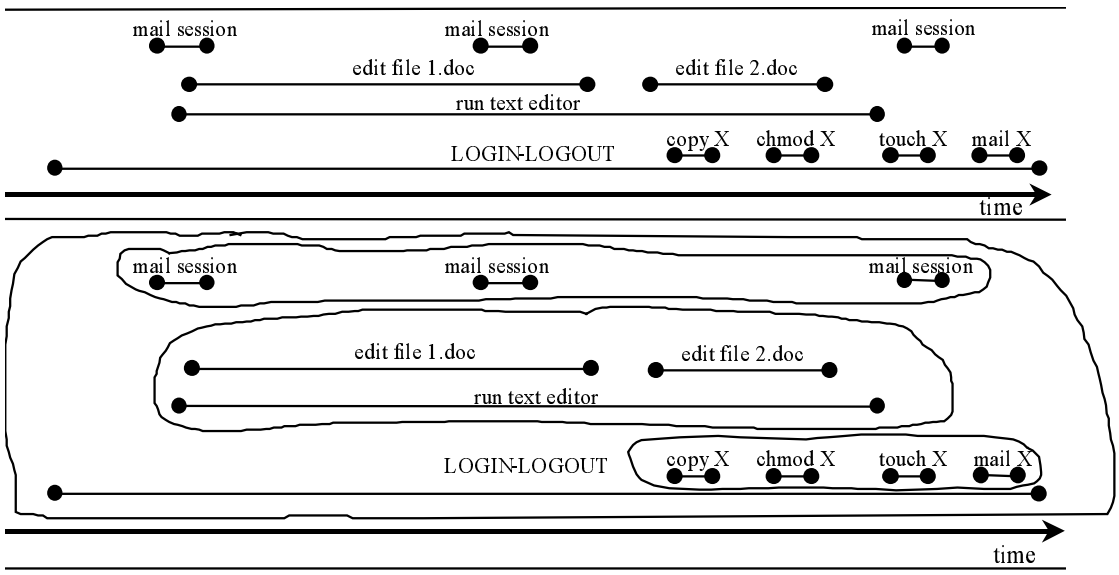


Figure 1.2 An example of a session with root attack

At the end of this scenario the attacker obtains a copy of the interactive shell interpreter $/bin/sh$, which is owned by root, $setuid$ enabled, and executable by everyone by exploiting a security vulnerability that existed in the UNIX 4.2BSD mail program (CERT advisories, 1999). The first command creates a shell copy under the root's mailbox and the second command sets the $setuid$ bit of the created copy. The third step is not vital to the successful completion of the penetration; the attacker only needs a bogus file to send to root as a mail file. In the last step, the attacker sends the bogus mail file to root, and the result is that the

mail program changes the ownership of the root's mailbox to root without resetting its *setuid* bit. An exploit command sequence, where it may be identified has three key steps (Figure 1.3):
>cp /bin/sh /usr/spool/mail/root
>chmod 4755 /usr/spool/mail/root
>touch X
>mail root <X

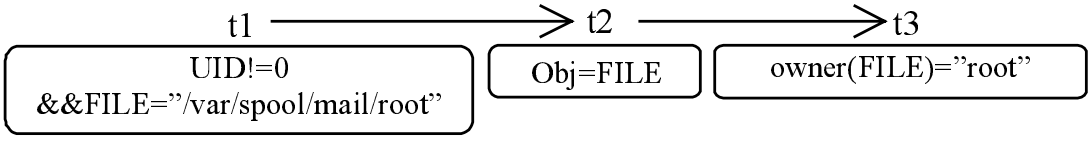


Figure 1.3 Landmarks of attack for gaining root privileges

In order to distinguish the key events the intrusion detection system analyzes coefficients of security significance of events in the input stream. In Figure 1.4 an architecture of pattern generator is shown.

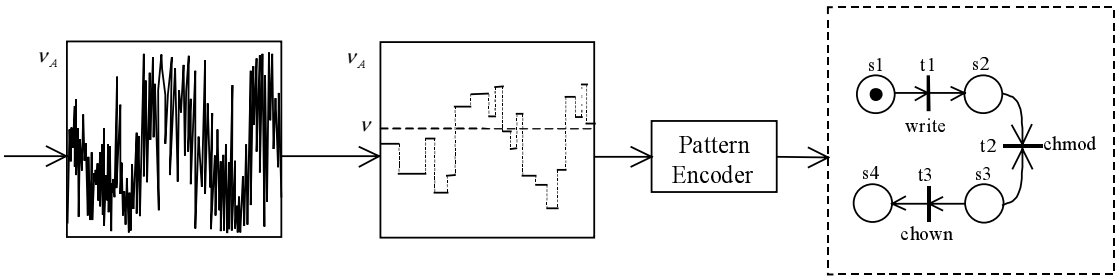


Figure 1.4 Architecture of the automatic pattern generator

To catch the hidden regularities of an attacker's behavior in an incoming stream the intrusion detection system has to define key events among all of them. It is done by analyzing the relationships between them and separating them into groups according to discovered relations. In other words, first, the system needs to create a second and then a third layer of information (event→action→activity) as described in Chapter 4. By scanning the input sequence of events the pattern generator creates a new information layer (action) and substitutes the original event layer in the queue by the highest - activity. On the third (activity) layer, as can be seen in Figure 1.2, the information that describes the attacker's behavior is separated into groups. As can be seen in Figure 1.2 the misuse events are spread among normal ones and on the third layer they are grouped in a separate group. This is a key group that contains key events (it is not necessary that all key events will be concentrated in a single group).

How can the intrusion detection system recognize key groups? A security significance coefficient is assigned to every action and taken from the action

profile (first box in Figure 1.4). The system uses these coefficients to calculate the coefficient of security significance for every group. In our experiments we took an average events' coefficient as a group coefficient. If deviations between the highest and lowest coefficients of security significance inside one group is higher than 20%, the system needs to repeat the relation search and the splitting inside of one group. This will assure the system that the events with very high coefficients of security significance will not disappear among the huge amount of non-important information that do not have any relation to the intrusive activity. In other words the system splits one group into several ones. In Figure 1.4 in the second box we can see the result of groups with their average coefficient of reliability calculated in every group.

As can be seen in the second box in Figure 1.4 all groups are separated by a threshold $\nu$ into two sets: key groups (above line) and other (below). The threshold should be defined experimentally. If it is set too high, patterns will be very general (consist of few states) and produce false alarms. If the threshold set is too low, the pattern will include not only key events that define an attack, but also personal features of the person's behavior who carried it out. It means that the system will detect this attack if it is performed by the same user and may overlook the same attack issued by another person.

The *pattern matching* component implements one of the pattern matching algorithms (for example, in our case it is Colored Petri Nets (Kumar, 1995)). It takes the events from key groups and defines states and transitions by encoding the commands into pattern and relations that grouped the events in this key group.

This approach is used for the detection of intelligent attacks. This attack forces the system to train misuse activities and not to classify them as abnormal in the future. The intrusion detection system checks the coefficients of reliability (assigned after classification) of key events and in the case these coefficients have high values, it means that a certain profile was already trained to accept misuse activities and the system reports it to the system administrator, which takes care of it.

For our experiments we chose to examine UNIX command data. This data contains the history of eight ($U0$-$U7$) different authorized users. Then it was given as an input stream for the learning process and a temporal-probabilistic tree was constructed for every user. After finishing the learning process we begin to inject events that represent different attacks. The threshold $\nu$ was varied under human supervision to create most representative patterns. Then after the attack detection and pattern creation (pattern creation process was not supervised by human) the attacks were repeated many times to get a statistic. Our pilot experiments showed that approximately 85% of repeated attacks were detected in real-time by automatically created patterns.

Our experiments also showed that in an automatically created pattern, features of personal behavior are still present. This means that the system detects

the same attack performed by the same person who created the pattern, with a higher probability than if it is performed by someone else. Therefore, we propose this direction as an area for future work.

## 1.3 Summary

Here we have shown a possibility for automatic pattern generation for intrusion detection. This approach uses probabilistic networks in the form of trees, which are adapted to catch temporal aspects of a users' behavior. It gives the possibility to establish cooperation between anomaly and misuse detection components in a way that the former one provides patterns for the latter one.

The main shortcoming is that the value of the $\nu$ constant is very difficult to select. Also only the basic kinds of attacks were tested, i.e. modern and complicated attacks have not been tested (such as attacks that are spread over several sessions). Thus, the material presented in here serves as a demonstration of an idea with some preliminary results. It shows the possibility to develop and use such approaches. Therefore, further research efforts, numerous tests and evaluations with real implementation are needed to verify the practical aspects of the approach.

# Appendix 2 Signals Used by Host Agent

Table 2.1 Types of signals used by host agent

| Signal | Description |
|---|---|
| *SIGABRT* | Issued if a process executes a system call *abort(2)*. |
| *SIGALRM* | Issued when a timer, set previously by *alarm(2)* and *settimer(2)*, expires. |
| *SIGBUS* | Hardware error. |
| *SIGCHLD* | Sent to a parent process when its child execution is finished. |
| *SIGEGV* | Access to a restricted memory address or a process does not have enough access rights. |
| *SIGPRE* | Error interrupt (division by zero, etc.). |
| *SIGHUP* | Sent to a group owner that has a control terminal in case if kernel detects that the terminal is disconnected. |
| *SIGILL* | Issued by the kernel if a process has performed an incorrect operation. |
| *SIGINT* | Sent to all processes when $< Ctrl > + < C >$ was pressed. |
| *SIGKILL* | Process termination signal. |
| *SIGPIPE* | Sent during an attempt of writing to pipe or socket to a recipient, which execution was already terminated. |
| *SIGPOLL* | Issued when an event arrives for a device being examined. |
| *SIGPWR* | Danger of power lost. |
| *SIGQUIT* | Sent to all processes when $< Ctrl > + < \backslash >$ was pressed. |
| *SIGSTOP* | Sent to all processes of a current group if a $< Ctrl > + < Z >$ was pressed. |
| *SIGSYS* | Incorrect system call. |
| *SIGTERM* | Warning of process termination. |
| *SIGTTIN* | Issued by kernel if a background process is trying to read from a control terminal. |
| *SIGTTOU* | Issued by kernel if a background process is trying to write to a a control terminal. |
| *SIGUSR*1 and 2 | Application defined signals. |

Table 2.2 System calls for file system operations

| Name | Description |
|---|---|
| *open(2)* | Opens a file for reading or/and writing. |
| *create(2)* | Creates a file. |
| *close(2)* | Closes a file descriptor associated with a previously opened file. |
| *dup(2)* | Duplicates a file descriptor. |
| *dup2(2)* | Duplicates a file descriptor allowing to specify its value. |
| *lseek(2)* | Sets a file pointer. |
| *read(2)* | Reads from file. |
| *write(2)* | Writes to a file. |
| *pipe(2)* | Creates a pipe. |
| *fcntl(2)* | Provides an interface for manipulation of an opened file. |

# Appendix 3 Prototype's Detection Accuracy Test Results

Table 3.1 Relational matrix approach: dependence of the detection accuracy change on the training time and sliding window size

| **Window size** | **Training time, weeks** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| 5 | 0.6255 | 0.4575 | 0.3351 | 0.3033 | 0.2995 | 0.2976 |
| 10 | 0.3137 | 0.2078 | 0.1731 | 0.1734 | 0.1651 | 0.1654 |
| 20 | 0.184 | 0.1691 | 0.1386 | 0.1148 | 0.1429 | 0.1449 |
| 30 | 0.1611 | 0.099 | 0.0884 | 0.1156 | 0.1127 | 0.1125 |
| 40 | 0.1433 | 0.0139 | 0.0984 | 0.0947 | 0.1151 | 0.077 |
| 50 | 0.1412 | 0.124 | 0.0802 | 0.0891 | 0.0663 | 0.06 |
| 100 | 0.0768 | 0.0195 | 0.0206 | 0.0197 | 0.0001 | 0.0169 |
| 200 | 0.0248 | 0.0121 | 0.0021 | 0.0109 | 0.0083 | 0.009 |

Table 3.2 Temporal-probabilistic tree approach: dependence of the detection accuracy on the sliding window size and length of the training period

| **Window size** | **Training time, weeks** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| 5 | 0.1899 | 0.1853 | 0.1837 | 0.1717 | 0.1359 | 0.1203 |
| 10 | 0.1227 | 0.1169 | 0.086 | 0.0915 | 0.0793 | 0.0506 |
| 20 | 0.0701 | 0.0494 | 0.0405 | 0.0343 | 0.0373 | 0.0217 |
| 30 | 0.1611 | 0.099 | 0.0884 | 0.1156 | 0.1127 | 0.1125 |
| 40 | 0.0434 | 0.0376 | 0.0268 | 0.0204 | 0.0202 | 0.0183 |
| 50 | 0.1213 | 0.1254 | 0.0167 | 0.0153 | 0.0093 | 0.068 |
| 100 | 0.0637 | 0.0295 | 0.0176 | 0.0187 | 0.001 | 0.0169 |
| 200 | 0.0344 | 0.0323 | 0.011 | 0.145 | 0.0001 | 0.0023 |

# Appendix 4 Terminology

This appendix provides an explanation of several terms used throughout the thesis. Some of them are widely accepted among security professionals, others are used in a specific manner in this dissertation[1].

*Action*
A stream of actions represents an *interval or action layer*, where the events with their relations (i.e. actions) are described. The action is considered as a temporal interval.

*Action Class*
Describes the action. It provides a formal description of an action without providing any specific details. Action class contains descriptions of events that start and end that action and possible events between them.

*Action Class Instance*
It is an instance that describes a certain group of actions that belong to the same action class and have similar temporal characteristics. By similar temporal characteristics we imply temporal distances (time lengths) that characterize actions. These distances must be distributed normally in order to be grouped into the same instance.

*Activity*
The most complicated level is the *activity layer*, which is represented by a stream of actions (temporal intervals) and relationships between them, because the actions are extended in time, different actions may overlap in time and interact. A single occurrence on this level is called an activity.

*Audit record*
An audit record is an entry of an audit trail. It is also referred to in this dissertation as an *event*.

*Audit trail/Event stream*
An audit trail is defined in Longley and Shain (1987) as a chronological record of system activities that is sufficient to enable the reconstruction, review and examination of the sequence of environments and activities surrounding or leading to each event in the path of a transaction from its inception to output of final results. The term *event stream* is used in the dissertation in the same sense as an audit trail.

*Event*[2]
In this dissertation an event refers to a single record in an audit trail. The number

[1] In this dissertation we attempt to use all definitions and terms in accordance with (Jensen *et al.*, 1998), (Allen and Ferguson, 1994), and (Longley and Shain, 1987).
[2] *Event*, *action* and *activity* terms are used in a special manner in this dissertation.

of distinct event types is finite and known as a priori. An event is described by a single instant relative to a particular user. We add to this description, the name of the event with an integer index that defines the number of the event relative to the user and also a place from which the user caused this event. Also, the last field is reserved for the event's particular information, which differs according to the type of event: it may be the name of the user, the name of the computer, an error code, etc. In other words, it is auxiliary information.

*Exploitation*

Exploitation is a set of actions that result in a violation of the security policy of a computer system (Krsul, 1998). Intruders exploit system vulnerabilities or flaws to gain unauthorized access to the system.

*Flaw*

A flaw is defined in Longley and Shain (1987) as an error of commission, omission or oversight in a system that allows protection mechanisms to be bypassed.

*Relation/Relationship*

It is a relationship between any two actions/events. It is characterized by a temporal distance between these actions/events. Thus, the name is a qualitative parameter that describes what kind of relation it is, and the temporal distance is a quantitative parameter describing temporal characteristic of the relation.

*Relation Class*

It is a notion that describes one of all possible relationships between any two actions: $Action_i$ and $Action_j$. It is defined as one of Allen's interval temporal relations (Allen, 1983).

*Relation Class Instance*

Relation class instance describes some set of relations that have similar temporal characteristics and each of them belongs to the same relation class. In other words, a relation instance has a *Name* and describes some distribution of temporal parameters of relations grouped by this instance.

*Relational Matrix*

To represent allowed relationships between actions a square matrix $N \times N$ - *relational matrix* - is used. In a cell $\{i, j\}$, the matrix holds relational classes that are allowed between the two classes $i$ and $j$. Thus, cells $i, j \ when \ j > i$ contain direct relations for $Ai$ and $Aj$, and cells $i, j \ when \ j < i$ contain the reverse ones. Using the relational matrix we may check whether there is a certain relation between any of the two classes and if it fits into a certain relation instance.

*Security policy*

A security policy defines the requirements of acceptable usage of the computer recourses and establishes correct procedures for their usage.

*Temporal-Probabilistic Tree*
It is a representation method to represent a typical model of user behavior. It contains the probabilistic information as well as temporal.

*Vulnerability*
Vulnerability is defined in Longley and Shain (1987) as a weakness in automated system security procedures, administrative controls, internal controls etc. that could be exploited by a threat to gain unauthorized access to information or to disrupt critical processing.

1 ROPPONEN, JANNE, Software risk management - foundations, principles and empirical findings. 273 p. Yhteenveto 1 p. 1999.
2 KUZMIN, DMITRI, Numerical simulation of reactive bubbly flows. 110 p. Yhteenveto 1 p. 1999.
3 KARSTEN, HELENA, Weaving tapestry: collaborative information technology and organisational change. 266 p. Yhteenveto 3 p. 2000.
4 KOSKINEN, JUSSI, Automated transient hypertext support for software maintenance. 98 p. (250 p.) Yhteenveto 1 p. 2000.
5 RISTANIEMI, TAPANI, Synchronization and blind signal processing in CDMA systems. - Synkronointi ja sokea signaalinkäsittely CDMA järjestelmässä. 112 p. Yhteenveto 1 p. 2000.
6 LAITINEN, MIKA, Mathematical modelling of conductive-radiative heat transfer. 20 p. (108 p.) Yhteenveto 1 p. 2000.
7 KOSKINEN, MINNA, Process metamodelling. Conceptual foundations and application. 213 p. Yhteenveto 1 p. 2000.
8 SMOLIANSKI, ANTON, Numerical modeling of two-fluid interfacial flows. 109 p. Yhteenveto 1 p. 2001.
9 NAHAR, NAZMUN, Information technology supported technology transfer process. A multi-site case study of high-tech enterprises. 377 p. Yhteenveto 3 p. 2001.
10 FOMIN, VLADISLAV V., The process of standard making. The case of cellular mobile telephony. - Standardin kehittämisen prosessi. Tapaustutkimus solukkoverkkoon perustuvasta matkapuhelintekniikasta. 107 p. (208 p.) Yhteenveto 1 p. 2001.
11 PÄIVÄRINTA, TERO, A genre-based approach to developing electronic document management in the organization. 190 p. Yhteenveto 1 p. 2001.
12 HÄKKINEN, ERKKI, Design, implementation and evaluation of neural data analysis environment. 229 p. Yhteenveto 1 p. 2001.
13 HIRVONEN, KULLERVO, Towards Better Employment Using Adaptive Control of Labour Costs of an Enterprise. 118 p. Yhteenveto 4 p. 2001.
14 MAJAVA, KIRSI, Optimization-based techniques for image restoration. 27 p. (142 p.) Yhteenveto 1 p. 2001.
15 SAARINEN, KARI, Near infra-red measurement based control system for thermo-mechanical refiners. 84 p. (186 p.) Yhteenveto 1 p. 2001.
16 FORSELL, MARKO, Improving Component Reuse in Software Development. 169 p. Yhteenveto 1 p. 2002.
17 VIRTANEN, PAULI, Neuro-fuzzy expert systems in financial and control engineering. 245 p. Yhteenveto 1 p. 2002.
18 KOVALAINEN, MIKKO, Computer mediated organizational memory for process control. Moving CSCW research from an idea to a product. 57 p. (146 p.) Yhteenveto 4 p. 2002.
19 HÄMÄLÄINEN, TIMO, Broadband network quality of service and pricing. 140 p. Yhteenveto 1 p. 2002.
20 MARTIKAINEN, JANNE, Efficient solvers for discretized elliptic vector-valued problems. 25 p. (109 p.) Yhteenveto 1 p. 2002.
21 MURSU, ANJA, Information systems development in developing countries. Risk management and sustainability analysis in Nigerian software companies. 296 p. Yhteenveto 3 p. 2002.
22 SELEZNYOV, ALEXANDR, An anomaly intrusion detection system based on intelligent user recognition. 186 p. Yhteenveto 3 p. 2002.