

SHEDEL - A SIMPLE HIERARCHICAL EVENT DESCRIPTION LANGUAGE FOR SPECIFYING ATTACK SIGNATURES*

Michael Meier, Niels Bischof, Thomas Holz

*BTU Cottbus, Computer Science Department, Computer Networks and Communication Systems,
P.O. Box 10 13 44, 03013 Cottbus, Germany
{mm, nb, thh}@informatik.tu-cottbus.de*

Abstract: A main problem for the detection of security violations in misuse detection systems is the manner how attack scenarios (signatures) are described. Attack languages are used to specify attack scenarios for misuse detection systems. Usually not only the attack signatures are described also some details controlling the detection process have to be noted. This is disadvantageous because it makes signature development more complicated and prone to errors. In this paper we propose an attack language for describing signatures without caring about the used detection techniques. The language further provides means to simplify the description of attack signatures.

Key words: Intrusion detection, misuse detection, attack languages, signature description

1. INTRODUCTION

In the area of computer security intrusion detection systems (IDSs) play an important and growing role for the automatic identification of security policy violations. In addition to preventive security mechanisms they provide post-mortem detection capabilities.

A main problem for the detection of security violations in misuse detection systems is the way in which attack scenarios (signatures) are described.

* The work described here was partially funded by Deutsche Forschungsgemeinschaft under contract number Ko 1273/13-1.

For this purpose, different languages are used that are usually referred to as attack languages. Examples of attack languages are STATL [1], ADeLe [2] and rule-based languages like RUSSEL [3] and P-BEST [4].

These languages do not only describe attack signatures but they also specify how these attacks are detected in an IDS. This is a disadvantage because it makes the process of describing attack signatures more complicated and susceptible to errors. In this paper an approach is presented that exclusively bases on a description of attack signatures without determining details of the attack detection mechanisms.

The paper is organized as follows: After an explanation of misuse detection concepts and a report about our experience in describing attack signatures in section 2 we give a short discussion of existing attack languages in section 3. An overview of SHEDEL is given in section 4. A presentation of the language syntax and informal semantics can be found in section 5. Section 6 contains examples of SHEDEL notations. Finally we give an outlook on future research work.

2. MISUSE DETECTION

Misuse detection is an appropriate approach for automatic detection of known intrusions. For effective misuse detection, there is a need for adequate monitoring mechanisms in the protected environment. Here security audit functions of computer systems play an important role.

Intrusions represent a set of related actions or events in a system, e.g. a sequence of system calls or network packets. The task of an audit function is to capture information about the execution of each of these security relevant actions by generating audit data records that can be used for later analysis. Misuse detection systems try to detect sequences that correspond to known signatures. Thereby is assumed that security violations do manifest themselves in distinct audit data records, and can be detected on the basis of this audit data. Therefore the suitability of the audit function is one factor that determines the quality of misuse detection results. In many cases inadequate audit data representations make signature development more complicated.

After identifying and encoding signatures are used for analyzing audit data streams. The analysis component searches the data stream for the patterns encoded in signatures. If it finds evidence that an attack has occurred a response action may be performed. Figure 1 summarizes this simple operational model of misuse detection.

The audit data records generated during an attack represent the **manifestation** (trace) of the attack. A **signature** of an attack describes the criteria (pattern) required to identify the manifestation of an attack in an audit data

stream. It is possible that several attacks of one type are executed simultaneously and proceed independently. Therefore it is necessary to be able to distinguish different instances of an attack. A **signature instance** is a set of criteria that clearly identifies the manifestation of one instance of an attack in the audit data stream.

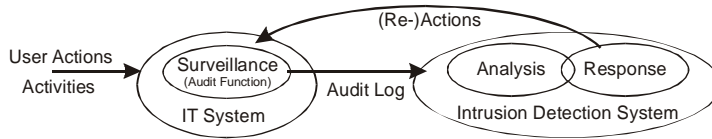


Figure 1. Simple model of misuse detection

One of the core problems in misuse detection is the development and maintenance of large signature bases. This process is time consuming and requires expertise about system vulnerabilities, audit mechanisms, and analysis techniques. Because of this it is often difficult to find appropriate personnel for these tasks. There is a growing need for methods that simplify the process of signature base maintenance or automate it at least partially.

We experienced these facts while developing and using the IDS AID (Adaptive Intrusion Detection system [5]). The misuse detection component of AID consists of an expert system that was implemented using the commercial expert system shell RTworks 3.5. Attack scenarios are modeled using finite deterministic automata and signatures are implemented as rules and data structures (frames) using the rule-based language of RTworks. State transitions are implemented as rules and state information is saved using frames. Thereby a signature is scattered over a set of rules. Further not only the signatures itself must be described also some operational details have to be noted. For example, inside the Then-Part of a rule usually new frames have to be created or existing frames must be changed explicitly to specify state changes or forking new signature instances. In addition the rule-based language does not provide adequate mechanisms that allow abstractions. This leads to a lack of clarity that makes signature development and maintenance very complicated and error-prone.

3. ATTACK LANGUAGES

The goal of misuse detection systems is to automatically identify manifestations of attacks in audit data generated by computer systems. Therefore first of all it's necessary to determine signatures. This usually involves an in-depth investigation of the constituent actions of an attack. Thereafter these signatures must be expressed in an adequate representation that can be used as a basis for the analysis process. Furthermore, one has to describe and en-

code the response actions and alert messages that should be generated, if an attack instance has been detected.

Thus several kinds of information are relevant in the context of misuse detection. To encode information on attacks so-called attack languages are used. In [6] six classes of attack languages are discussed. **Exploit languages** encode the actions to be undertaken to perform an attack. Usually these are programming or script languages such as C or Perl. **Event languages** are used to represent the events to be analyzed by the IDS. They mainly focus on format specifications of audit data. Examples are BSM [7] and TCPDUMP [8] record specifications. Signatures of attacks are specified using **detection languages**. **Response languages** allow the encoding of actions to be initiated in response to a detected intrusion. Current IDS mostly use library functions written in programming languages like C for this purpose. **Report languages** describe the format of alert messages with information about detected attacks. An example is the Intrusion Detection Message Exchange Format (IDMEF), currently undergoing the IETF standardization process. Note that information expressed in report languages can also form the input for higher-level analysis. Therefore report languages can be seen as a special kind of event languages. Higher-level analysis for example aims at detecting coordinated attacks. This process uses relations between attacks that are formulated using **correlation languages**. Sometimes correlation languages can be seen as detection languages on a higher level of abstraction. Figure 2 shows how information expressed in attack languages incorporate into our model of misuse detection.

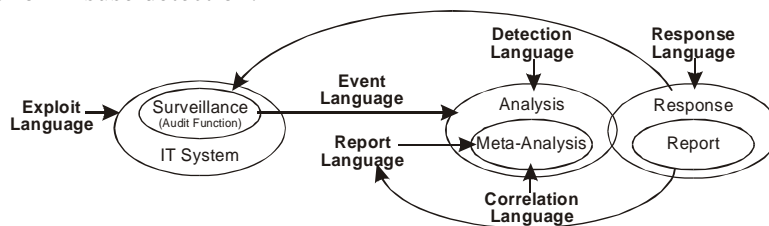


Figure 2. Model of misuse detection with attack languages involved

The information expressed with attack languages can be divided into the following three categories: **(re-) actions**, **log/audit data** and **detection/analysis criteria**. Some attack languages, especially those to encode log and audit data, are mainly designed and used by software manufacturers during software development. A collection of signatures usually comes with IDS products. Since no computer environment equals another IDS administrators have to adapt signatures to the local environment. Often administrators also must develop new environment specific signatures, matching the local security policy and conditions. Table 1 summarizes the relation between user groups, information and languages.

Configurations of computer systems are usually very volatile. Every time software is updated or a new vulnerability arises signatures must be adapted to ensure proper operation of the IDS. Therefore updating and developing signatures using detection languages is one of the most frequent, critical and even complex tasks during the deployment of misuse detection systems. Since IDS administrators must do at least local adaptations of the signatures using detection languages the user group of these languages seems to be larger than those of all other attack languages.

Table 1: Kinds of information, languages and their users

Kind of information:	(Re-) Actions	Log/Audit Data	Detection/Analysis Criteria
Encoded using:	Exploit and response languages	Event and report languages	Detection and correlation languages
User group:	(Attackers), developers, administrators	Developers	Developers, administrators

3.1 Detection Languages

Detection languages are used to encode criteria for identifying manifestations of attack instances. A number of detection languages has been proposed. Usually each IDS provides its own language. Most languages are specific to the detection mechanisms used to analyze the audit data stream. Examples of existing detection languages are P-BEST (Production-Based Expert System Toolset [4]), which is a component of SRI's Emerald and RUSSEL that is used by ASAX (Advanced Security Audit Analysis for Unix [3]). Both are rule-based languages, where signatures are expressed as rules similar to the signatures in AID. Other examples are STATL [1] used by the STAT-Tool suite [9] and the detection language of the IDIOT IDS [10]. In STATL signatures are described as compositions of states and transitions. The IDIOT IDS language uses a colored petri net representation to express attack signatures. Other languages are LAMBDA [11] and ADeLe [2], which are not pure detection languages. They describe information about different aspects of attacks. ADeLe tries to combine all knowledge available for a given attack. It is a detection, correlation, report and exploit language.

Although these languages have shown to be useful, they miss some important properties. Desirable properties are discussed in [10].

As shown above, users of detection languages are mainly IDS administrators. Therefore we argue that simplicity and usability of detection languages are main issues, especially because these languages must be appropriate to use for a heterogeneous group of people with different expertise.

With the most existing languages not only a description of signatures must be provided; some details on how the detection process should work

have to be noted. This makes the complex task of signature development and maintenance even more complicated and more error-prone.

For that reasons, the work described in this paper was focused mainly on simplicity and usability aspects of languages. We present an event-based language that tries to make signature description as simple as possible. Using our language SHEDEL only the signatures itself must be described and no information about how the detection process should work has to be noted.

4. BASIC FEATURES OF SHEDEL

In contrast to many other approaches the main goal of the SHEDEL approach is to provide means to describe signatures of intrusions without caring about their detection. The central abstraction in SHEDEL is the event. A signature is a description of an *event* that represents the completion of an attack. An event consists of a collection of events (subevents, steps), which are in relationship to each other, temporally or by their properties. These subevents are also specified as events. There is a set of basic events without subevents, which represent the basic recognizable units (audit records). Thus it is possible to specify a hierarchy of events that can be used to describe audit events, attack signatures, meta attack signatures and so on. To support the response to attacks (or any events) SHEDEL allows to link actions to a subevent of an event specification.

An event is specified by name. Every single step of an event has a type that refers to the name of another previously defined event. For each step, it must be specified what predecessor steps are required for this step to occur. There are four kinds of steps: initial, final, canceling and intermediate steps.

Steps without predecessor steps are **initial steps**. These are the first steps that have to occur within the event being specified. They occur if and only if conditions that relate to attributes of these steps are fulfilled. The occurrence of a **final step** entails the occurrence of the containing event. If, for instance, the event being specified represents an attack, then the incidence of a final step signals the completion of the attack.

In multistage attacks it may happen that already established preconditions of an attack are removed before the completion of the attack (e.g. the deletion of a file). In these cases it is not necessary further to observe this attack under these specific circumstances. The occurrence of a **canceling step** indicates that the containing event cannot occur in this context. All steps that are neither initial nor final nor canceling steps are called **intermediate steps**.

For the occurrence of a step prerequisites must be fulfilled. First all predecessor steps have to have occurred before unless it is an initial step. Second all specified conditions, relating to this step, have to be fulfilled.

To specify conditions for a step or conditions between the properties of subsequent steps in an event description it is necessary to declare the properties of an event. These event properties, called *features*, are defined by references to the properties of the steps of this event. A feature definition is a pair consisting of a name and a feature of a step contained by this event.

Figure 3 illustrates the definition of features of an event E, which consists of the steps A and B. The features *start* and *filename* of E are defined using the features *time* and *file* of step A. The *username* feature of E is mapped to the *user* feature of step B.

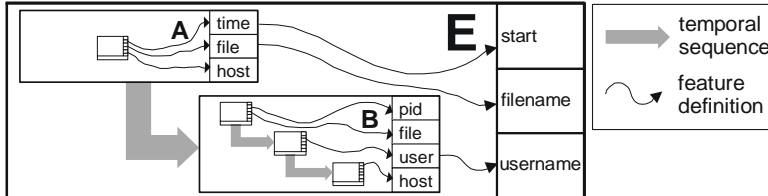


Figure 3. Definition of event features

5. SYNTAX AND INFORMAL SEMANTICS

A specification in SHEDEL is a sequence of EVENT clauses

```
EVENT name1 {
  <step spec1> <step spec2> ...
  <conditions spec1>
  <feature spec1> }
EVENT name2 {
  ...
```

5.1 Specifications of steps

The steps of an event are described in a <step spec> of the form:

```
STEP sname <kind spec> TYPE ename
  <occurred spec>
  <predecessor spec>
  <action spec>
```

sname is a unique identifier of the step within this event description. <kind spec> contains some of the keywords INITIAL, EXIT or ESCAPE to mark the step as initial, final or canceling step. *ename* represents the type of this event, that is the name of a already defined event description.

Within the optional `<occurred spec>` can be specified how often (`num1`) and within what period of time (`num2`) the event `ename` has to occur so that the step `sname` has occurred:

```
OCCURRED num1 TIMES WITHIN num2
```

This construct is useful especially for signatures where the frequency of an event is a significant criterion. Examples are signatures for detecting so called doorknob-rattling attacks, where a certain number of failed logins (e.g. 5) within a period of time (e.g. 60 seconds) indicates that someone tries to guess the correct password. Using `<occurred spec>` allows a compact description of this kind of signatures, as the following example shows:

```
EVENT LoginFailure { ... }
EVENT DoorknobRattling {
  STEP failure INITIAL EXIT TYPE LoginFailure
  OCCURRED 5 TIMES WITHIN LAST 60.0
  CONDITIONS
    [failure].hostname == failure.hostname,
    [failure].username == failure.username
  ACTIONS ALERT("Doorknob rattling ...") }
```

Temporal dependencies between steps are expressed using the `<predecessor spec>`:

```
REQUIRES step5 AND step6 OR step7
```

where after the `REQUIRES` keyword any `OR/AND` combination of step names of the same event description can be specified. Each step name in this expression can be extended by a time clause to specify time constraints:

```
WITHIN LAST num
```

where `num` is the number of seconds elapsed since that predecessor step.

Note that using the `<predecessor spec>` clause a sequence or a choice of subevents can be expressed. Not using this clause is a way of describing an independent set of subevents. Non-Occurrence of an event can be expressed using canceling steps.

For each step, actions can be specified. These are predefined functions (e.g. `ALERT()` in the example above) listed in `<action spec>`:

```
ACTIONS action1(par1, par2), action2(), action3()
```

The parameters of these functions are either constants or step attributes. Step attributes are properties of the current step; they are referred to using the identifier specified in the event description named after the `TYPE` keyword.

5.2 Conditions

A simple collection of subevents does not necessarily indicate the incidence of the whole event. Usually these steps have to relate to the same ob-

ject (e.g. file) or be associated with the same user, etc. This kind of conditions can be specified using a `<conditions spec>`:

```
CONDITIONS <conditionexp1>, <conditionexp2>, ...
```

`<conditionexp>` is a AND/OR combination of predefined Boolean operators or functions. Examples of these functions are string functions used to check for distinct sub-strings etc. The parameters of these functions are constants, functions or step attributes. Step attributes have the syntax

```
stepname.featurename
```

where `stepname` identifies the corresponding step in this event description and `featurename` is the name of the referred property of this step.

If step `stepname` is specified with a frequency statement (`<occurred spec>`) the repetitive events combined to this step can not uniquely identified with this identifier. In this case brackets can be used to distinguish the previous event (referred to by `[stepname]`) from the latest event (referred to by `stepname`). In the doorknob-rattling example above this mechanism is used to specify the conditions that the `username` and `hostname` properties of all `LoginFailure` events must be the same.

A step can only occur if all *relevant* conditions containing a step attribute of this step are fulfilled. *Relevant* conditions are those, which refer only to attributes of already occurred steps. For example in

```
EVENT event17 {
  STEP s1 INITIAL TYPE event1
  STEP s2          TYPE event2 REQUIRES s1
  STEP s3          TYPE event1 REQUIRES s2
  CONDITIONS s1.user == s2.user, s2.user == s3.user
  FEATURES user := s1.user}
```

after `s1` having occurred for the determination of the occurrence of `s2` is only the first condition a relevant condition because `s3` hasn't occurred yet.

5.3 Features of an event and data types

Named features are used to define properties of an event. Their meaning is defined by assigning a feature of an event step. The assignment of features is done in `<feature spec>`:

```
FEATURES
  feature1 := step1.attribute2,
  feature2 := (step2.attribute2 | step3.attribute4),
  feature3 := step4.attribute1
```

Since there can be different sequences of steps that entail the concerned event not all steps referred to in `<feature spec>` have to be occurred when a final step has occurred. Thus for some features (e.g. `feature2` in the example above) a list of step attributes has to be specified so that in every possible sequence of steps exactly one of the listed steps has occurred.

SHEDEL supports a simple type system. There are four types: `INTEGER`, `FLOAT`, `STRING` and `BOOLEAN`. Each feature of an event description has one of these types. However, they don't have to be declared because they are derived from the type of the contained subevents.

5.4 Basic Events

At the lowest level of event specifications are basic events. These events represent audit records, network packets, etc. The original data is preprocessed and converted to event objects. There are no event descriptions necessary for these events. They can be viewed as events without steps. They just have features and to define event descriptions based on these fundamental events it is necessary to know their features including their types.

Different domains of basic audit events can be used in parallel. This allows the description of signatures for heterogeneous environments. Currently two domains are supported: Solaris BSM and Windows 2000 Security Event Log. New domains can be easily integrated.

Signature descriptions in SHEDEL can be compiled into an internal format. An algorithm matching signatures in this format against audit data was implemented, but will not be discussed here in detail. In brief, an incoming audit record is mapped to the corresponding basic event. Further the occurrence of this basic event is signaled and all event descriptions referring to events of this type are checked. If all prerequisites of an event description are fulfilled the occurrence of an event corresponding to this event description is signaled. When all signaled events are processed, the next audit record is injected as basic event and the process starts again.

6. EXAMPLES

In the following we present examples of SHEDEL signatures. First we show how the language can be used to build abstractions that are appropriate to discourse about attack signatures. Second we demonstrate a more complicated example of a multistage signature that uses several abstractions.

6.1 Generalization and specialization

One of the problems that make signature description difficult is the some times inadequate encoding of information in audit records. For example, for a signature searching for failed logins in a Windows NT environment ten different types of event log records (NT event id's 529 – 539 w/o 538) have to be considered. To describe, for example, the doorknob-rattling signature

above, all these ten different events have to be considered. This can be simplified by describing an abstraction of these events. The new event description subsequently can be used as generalization of all Windows NT failed login events in every signature that looks for events of this kind. The event description of the abstract FailedLogin event looks as follows:

```
EVENT FailedLogin {
  STEP failure529  INITIAL EXIT  TYPE NT_EVENT_529
  STEP failure530  INITIAL EXIT  TYPE NT_EVENT_530
  ...
  STEP failure539  INITIAL EXIT  TYPE NT_EVENT_539
  FEATURES
    hostname = (failure529.host | ... | failure539.host)
    username = (failure529.user | ... | failure539.user)
  ...}
```

The specialization of events is another way to simplify the description of signatures. For example, many signatures relate to actions that grant rights of a privileged user. Usually there are several signatures to be described that are looking for failed logins against the administrator account. In this case it may be appropriate to introduce a new event that describes this situation. This can be achieved by specializing the above event in the following way:

```
EVENT AdministratorFailedLogin {
  STEP failure  INITIAL EXIT  TYPE FailedLogin
  CONDITIONS
    // only failed logins of the administrator account
    failure.username == "Administrator"
  FEATURES
  ...}
```

6.2 Solaris SUID-ROOT-Shell Signature

As second example we present a signature that can be used to detect exploitations of a vulnerability in SunOS 4.1.1. In this version of UNIX a user who executes a command file with a name starting with a hyphen (“-“), e.g. `-fn`, can obtain an interactive shell with administrative (`root`) privileges. We assume `script1` is a shell script owned by `root` with SUID bit set. The user creates a link named `-fn` to `script1`. When the user executes `-fn`, he receives a shell with `root` privileges. An illustration of the signature used to detect this attack is shown in figure 4.

Step `step_1` is the initial step that searches for the creation of a suspicious named link to a shell script using the criteria encapsulated in the event type `hyphenlink_to_shellscript`. The step `step_rename` looks for actions renaming the link to a new link name that still starts with a hyphen. The final step is `step_end`. It represents the execution of the link. The steps `step_rename_exit` and `step_exit` are canceling events.

They search for actions that rename the link to a name that does not start with a hyphen or that remove the link.

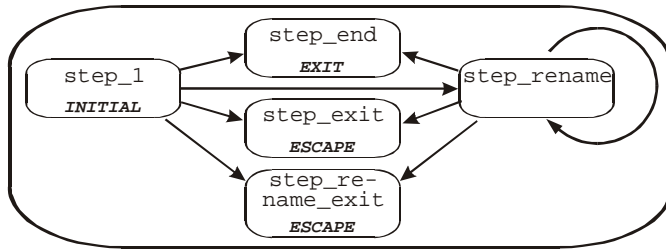


Figure 4. An example attack signature

This signature is described in SHEDEL as follows (the definition of the abstract event types is not shown here):

```

EVENT suidShellScriptAttack {
  // create a suspicious named link to a SUID shell script
  STEP step_1 INITIAL TYPE hyphenlink_to_shellscript
  // rename the link to a still suspicious link name
  STEP step_rename TYPE rename_to_hyphen
  REQUIRES step_1 OR step_rename
  // remove the link
  STEP step_exit ESCAPE TYPE audit_unlink
  REQUIRES step_1 OR step_rename
  // rename the link to new name, that doesn't starts with -
  STEP step_rename_exit ESCAPE TYPE rename_not_to_hyphen
  REQUIRES step_1 OR step_rename
  // execute the link
  STEP step_end EXIT TYPE execution
  REQUIRES step_1 OR step_rename
  ACTIONS ...
  CONDITIONS
  // the link renamed, removed or executed and the link
  // created in step_1 must be the same
  step_1.linkname == step_rename.oldname,
  step_1.linkname == step_exit.linkname,
  step_1.linkname == step_rename_exit.oldname,
  step_1.linkname == step_end.linkname,
  // the name of the link to be renamed must be equal to
  // result of the last renaming
  [step_rename].newname == step_rename.oldname,
  // the name of the removed or renamed link must be equal
  // to the last renaming result
  step_rename.newname == step_exit.linkname,
  step_rename.newname == step_rename_exit.oldname,
  // the name of the executed link must be equal to the
  // last renaming result
  step_rename.newname == step_end.linkname
  FEATURES ... }
  
```

7. CONCLUSION AND FUTURE WORK

We have presented the detection language SHEDEL. It has been designed with respect to simplicity and usability aspects. The language provides means to build abstractions to discourse about attack signatures and frees signature developers from caring about details of misuse detection techniques. All signatures developed for the IDS AID have been described in SHEDEL. Currently a graphical user interface is being developed that can be used for visual signature description.

Future work will focus on the development of event libraries for TCP/IP network packets and IDMEF alert messages. We also plan to develop compilers to translate SHEDEL signatures into other detection languages. In addition, we investigate opportunities to describe attack correlation criteria. Further we analyze how information about the static structure of event-based signature descriptions can be utilized for match algorithm optimizations.

8. REFERENCES

- [1] Eckmann, S. T.; Vigna, G.; Kemmerer, R. A.: STATL: an Attack Language for State-based Intrusion Detection, in Proc. of the ACM Workshop on Intrusion Detection, Athens, Greece, Nov. 2000.
- [2] Michel, C.; Me, L.: ADeLe: an Attack Description Language for Knowledge-based Intrusion Detection, in Proc. of the International Conference on Information Security, Kluwer, Jun. 2001.
- [3] Mounji, A.: Languages and Tools for Rule-Based Distributed Intrusion Detection, PhD thesis, University of Namur, Belgium, Sep. 1997.
- [4] Lindqvist, U.; Porrás, P. A.: Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST), in Proc. of the IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, Oakland, CA, May 1999, pp. 146-161.
- [5] Sobirey, M.; Richter, B.; König, H.: The Intrusion Detection System AID. Architecture, and experiences in automated audit analysis, in Proc. of the International Conference on Communication and Multimedia Security, Essen, Germany, Sep. 1996, Chapman & Hall, London, pp. 278-290.
- [6] Vigna, G.; Eckmann, S. T.; Kemmerer, R. A.: Attack Languages, in: Proc. of the IEEE Information Survivability Workshop, Boston, MA, Oct. 2000.
- [7] Sun Microsystems, Inc.: SunSHIELD Basic Security Module Guide, Solaris 2.6 System Administrator Collection Vol 1, Mountain View, CA, 1997.
- [8] <http://www.tcpdump.org>
- [9] Vigna, G.; Eckmann, S. T.; Kemmerer, R. A.: The STAT Tool Suite, in Proc. of DISCEX, Hilton Head, South Carolina, IEEE Computer Society Press, Jan. 2000.
- [10] Kumar, S.: Classification and Detection of Computer Intrusions, PhD Thesis, Purdue University, 1995.
- [11] Cuppens, F.; Ortalo, R.: LAMBDA: A Language to Model a Database for Detection of Attacks, in Proc. of the Third International Workshop on the Recent Advances in Intrusion Detection, LNCS 1907, Springer, 2000, pp. 197-216.