

A Model for the Semantics of Attack Signatures in Misuse Detection Systems

Michael Meier

Brandenburg University of Technology Cottbus, Computer Science Department
P.O. Box 101344, 03013 Cottbus, Germany
mm{-at-}informatik.tu-cottbus.de

Abstract. Misuse Detection systems identify evidence of attacks by searching for patterns of known attacks (signatures). A main problem in this context is the modeling and specification of attack signatures. A couple of languages are proposed in the literature, which differ in the aspects of signatures that can be described. Some aspects that can be specified in one language cannot be expressed in another. In the area of Active Databases the specification of triggers constitutes a similar problem domain. Zimmer et al [9] have developed a Meta-Model for the semantics of complex events in Active Database systems. In this paper we discuss differences between active database triggers and attack signatures and adapt the Meta-Model to the domain of attack signatures. We present the adapted model, which systematically enumerates the different aspects that characterize attack signatures. The aspects are discussed in detail and their meaning is demonstrated using examples. The model for the semantics of attack signatures represents a kind of a checklist for the development of a signature specification language or for the comparison of existing signature specification languages.

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 using misuse detection systems is the modeling and specification of attack scenarios (signatures). For this purpose, different languages are used that are usually referred to as attack languages. Examples of such attack languages are STATL [1], ADeLe [2], Lambda [3], Sutekh [4], SHEDEL [5] as well as rule-based languages like RUSSEL [6] and P-BEST [7].

These languages differ in the set of aspects of signatures that can be described. Some aspects that can be specified in one language cannot be expressed in another. Thereby these languages offer different expressiveness, and therefore are only appropriate in distinct domains.

In the area of Active Databases the specification of triggers constitutes a similar problem domain. A number of different languages has been proposed for the description of events of interest and responses that should be triggered if these events

occur. Zimmer et al [9] have developed a Meta-Model for the semantics of complex events in Active Database systems. This model treats the different aspects of event semantics in a systematic way.

Inspired by the Work of Zimmer et al we developed a model for the semantics of attack signatures. Instead of starting to build a new model from the scratch we tried to adapt Zimmer's model to benefit from already gained insights in the Active Database domain. Because of some differences between Active Database triggers and attack signatures we needed to modify and extend this model. In the paper we discuss differences between active database triggers and attack signatures. We present the adapted model of the semantics of attack signatures, which systematically enumerates the different aspects that characterize attack signatures. These aspects are discussed in detail and their meaning in the context of misuse detection is demonstrated using examples. With this model we provide a kind of a checklist for the analysis and development of signature specification languages or for the comparison of existing signature specification languages.

The paper is organized as follows: After an introduction of misuse detection concepts in Section 2 we give an overview of existing languages for the specification of complex events and discuss differences between event semantics in misuse detection and active databases in Section 3. Section 4 introduces some basic definitions that we use for the detailed presentation of the model in Section 5. Finally we draw some conclusions and give an outlook on future work.

2 Misuse Detection

Misuse detection is an appropriate approach for the 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 (they are observable), and can be detected on the basis of this audit data (they are detectable). 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.

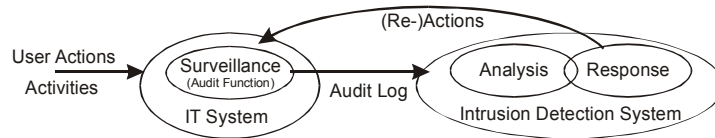


Fig. 1: Simple model of misuse detection

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 the same 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.

One of the core problems in misuse detection is the development of attack signatures. This process is time consuming and requires expertise about system vulnerabilities, audit mechanisms, misuse detection analysis techniques and a clear understanding of the semantics of attack signatures. We believe that wrong understanding of the semantics of attack signatures is one of the main sources of false alerts or undetected attacks. There is a growing need for methods that simplify the process of signature generation and models that clarify the semantics of attack signatures. A more detailed discussion of the signature development process and our experiences in this area can be found in [5].

To derive signatures it is necessary to analyze the constituent actions of an attack. This usually involves an in-depth investigation of the constituent actions of an attack. The derived signatures must be expressed in an adequate representation that can be used as a basis for the analysis process. Furthermore a description and encoding of the response actions and alert messages that are triggered, if an attack instance has been detected, is required.

Several kinds of information are relevant in the context of misuse detection. To encode information on attacks so-called attack languages are used. In [10] 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 [11] and TCPDUMP [12] 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 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.

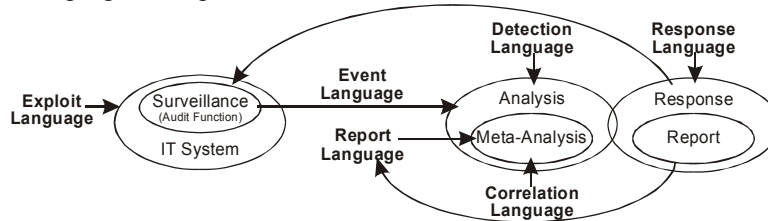


Fig. 2: Model of misuse detection with attack languages involved

3 Event Description Languages

Events are used as abstraction in the context of misuse detection signatures as well as in the context of active databases. In this section we give an overview of languages for the description of complex events. Further we discuss differences of event semantics in the both areas.

3.1 Active Database triggers

Active database systems have been developed for applications that need an automatic reaction in response to certain conditions being satisfied or certain events occurring. Active database systems use rules to monitor situations of interest and to trigger a response when these situations occur. Such rules can be used for a number of database tasks. For example, they can enforce integrity constraints, compute derived data and calculate statistics. These rules typically follow the ECA-Paradigm (Event Condition Action) [15]. When a triggering event occurs the conditions of related rules are evaluated. If a condition is fulfilled the related action is executed. Typical examples for triggering events are data manipulation operations. To react on more sophisticated situations complex events can be used, which are defined using operators of the event algebra of the event language. A number of event languages have been proposed. Examples are HiPAC[15], SNOOP[16], NAOS[17] and ACOOD[18]. All these languages describe complex events using an algebraic style.

3.2 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 possesses its own language. Most languages are dedicated to the detection mechanisms used to analyze the audit data stream. Detection languages can be classified in rule-based, state-transition-based and algebraic languages.

Examples of rule-based detection languages are P-BEST (Production-Based Expert System Toolset [7]), which is a component of SRI's Emerald and RUSSEL that is

used by ASAX (Advanced Security Audit Analysis for Unix [6]). In both languages signatures are described as rules. The language STATL [1] used by the STAT-Tool suite [13] and the detection language of the IDIOT IDS [14] are examples of state-transition-based languages. 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. Examples of algebraic detection languages are LAMBDA [3], ADeLe [2] and Sutekh [4]. Lambda and AdeLe are not pure detection languages. They are also used to describe other aspects of attacks. ADeLe tries to combine all knowledge available for a given attack. The detection language SHEDEL[5] uses a mixture of the algebraic and state-transition-based approach. Signature descriptions in SHEDEL are explicitly based on the abstraction event. The language of the system CARDS [19] also follows an algebraic style. This language as well as SHEDEL specifically support the introduction of abstract views in signature descriptions, which allows abstraction and simplifies the specification of attack signatures.

While a number of detection languages have been developed and used, to our knowledge there is no model of general characteristics which describe the aspects of attack signatures relevant to their detection. Most existing languages differ regarding their provided expressiveness that is difficult to assess and outline without a set of common criteria.

3.3 Differences of Event Semantics in Active Databases and Misuse Detection Systems

Before we discuss modifications and extensions of the Active Database Meta-model we shortly explain important difference between the application domains active databases and misuse detection.

As mentioned above rules for Active Databases typically follow the ECA-Paradigm. Attack signature specifications do not have this structure. Instead the three parts are combined in the complex event specification that represents the attack signature. In the context of misuse detection events only occur if they satisfy the specified condition. Further, actions may be executed also if no complex event has occurred yet. This can happen if an attack signature specifies that an action should be executed already after distinct parts of the complex event have occurred. This allows responses to attacks that have not been finished yet.

Another important difference is the handling of partial, not completed complex events. These are events that partially have occurred but are not yet finished. We will use the term *partial complex event instance* for these events in the rest of the paper. Active databases do not know the concept of partial complex event instances. This is because of a different method of instance creation (*late instance creation*). In active databases complex event instances are triggered by the final event that is part of the complex event. Thus there can never be a partial instance. In contrast, in misuse detection complex event instances are created by the first step that is part of the complex event (*early instance creation*). These instances are incomplete until all other events that are part of the complex event are occurred. It is also possible that a partial instance is never completed but is destroyed because an event occurs that make

it impossible for the instance to be completed. Misuse detection systems follow this transition-oriented procedure because it allows responding to partial detected attacks (e.g. raising a “yellow” alert).

Because of the fundamental differences discussed above modifications of the Active Database semantics model are necessary to be applied to misuse detection. After an introduction of some basic definitions we introduce the model for the semantics of attack signatures and show the meanings of several aspects using examples.

4 Basic Definitions

This section introduces the relevant basic definitions for the introduction of the model. The terminology of [9] was adapted and extended to be applicable for the misuse detection domain.

Definition 1: Event, Basic Event, Event Type, and Event Instance

The central abstraction we use is the event. An *event* is characterized by a set of features. There is a set of *basic events*, which represent the basic detectable units. A basic event is atomic and bound to a specific point in time.

An *event type* describes the features that are common to occurrences of events of that type. When specifying an attack signature actually the corresponding event type is described. Each concrete occurrence of an event type is called *event instance*. This distinction between type and instance is similar to that of programming languages.

Definition 2: Audit and Time Events

Basic events are classified into audit and time events. *Audit events* correspond to audit records of an audit trail that document security related actions. *Time events* are generated by a timer function. They describe specific points in time either absolutely (at seven o'clock) or relative (15 minutes after action x).

Definition 3: Complex Event, Step, and Parent Event Type

Complex events consist of a collection of events that we call subevents or *steps*. The steps of an event are in relationship to each other, temporally or by their features. Temporal relationships are expressed using operators of an event algebra. The event type of a complex event that contains steps is called the *parent event type* of these steps. An instance of an event type can be used as step in several complex events i.e. an instance can have several parent event types.

Definition 4: Event Trail

The *event trail*, sometimes also called history, is a partially ordered set of event instances. Its order reflects the occurrence times of its event instances.

Definition 5: Event Context, Intra-event condition, Inter-event condition

Events occur in a context that is often relevant to the detection of complex events. Context conditions define constraints regarding the context of an event. For the specification of complex events we distinguish intra-event conditions and inter-event-conditions. *Intra-event-conditions* consist of a Boolean formula with atoms being comparisons between features of a step and constants. They can be evaluated by merely inspecting some features of the corresponding step. *Inter-event conditions* are Boolean formulas with atoms that are comparisons between features of different steps.

Definition 6: Initial, Interior, Final, and Exit Steps

Steps that are a first step of a complex event, according to the temporal relations expressed using the event algebra, are called the *initial steps* of the complex event. The last step of an event is called the *final step*. After occurrence of a final step the complex event instance occurs. There are also steps that represent a kind of counter evidence for events. The occurrence of such a step, which we call *exit steps*, makes it impossible for a complex event instance to occur. When an exit step of an event instance occurs this instance is removed. As discussed in Section 3.3 Active Database systems do not create partial instances of complex events. Therefore Zimmer's model does not provide a notion of exit steps which is necessary in the context of misuse detection. All steps that are neither initial, final nor exit steps are named *interior steps*.

Notations:

We use upper case letters to denote event types and lower case letters for event instances. For event instances we encode the event type in the letter and use a subscript to indicate the order of occurrence times of the event instances. For example the trail $\{a_1 a_2 a_3\}$ contains a sequence of three instances of event type A .

Axioms:

1. Basic events are generated by the audit function of the monitored system (or in the case of time events by a timer function). This means that basic events occur when the corresponding audit log entry is generated and the occurrence time is determined by the time component of the log entry. The occurrence times reflect the order in which events occurred.
2. The occurrence time of a final step of a complex event determines the occurrence time of the complex event.
3. Events can occur simultaneously i.e. different event instances may get the same occurrence time.

5 The Model

In this section we present the model for the semantics of signature events in misuse detection systems. Following the structure of Zimmer's model [9] the semantics of

event descriptions are divided into three dimensions, whose semantic aspects are discussed in detail and demonstrated using examples.

The notation we use throughout the paper is not meant as proposal for a language to describe signatures. It is just a syntax used to discuss the examples.

5.1 Semantic Dimensions of Event Descriptions

Basically there are three questions that are to be answered by an event description to sufficiently define the semantics of the described event. From these questions a partitioning of event semantics in different dimensions can be derived [8].

Question 1: (Event Pattern)

Which event instances of a trail cause the occurrence of a complex event?

Several criteria have influence on the decision whether an instance is triggered by a given trail or not. For example the type, the order and number of event instances play a decisive role.

Question 2: (Step Instance Selection)

Which event instances of a subtrail that cause the occurrence of a complex event are bound to the steps of the complex event and are available for further event correlation and actions, which are taken in response to the occurrence of an event instance?

Since there may be multiple matching event instances in such a subtrail that can be bound to a step, it may be ambiguous which one is to be bound to this step.

Example 1: Consider an event description E_1 which describes a sequence of an event of type A followed by an event of type B followed by an event of type C and a trail $t_1 := \{b_1 x_1 a_1 c_1 y_1 b_2 x_2 b_3 y_2 c_2\}$. Figure 3 exemplifies how the event pattern selects event instances from the trail that cause the occurrence of a complex event E_1 . It also shows that there are alternatives for the selection of an event instance for the step of type B .

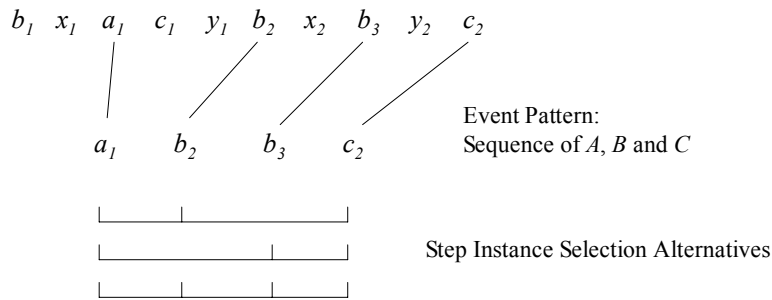


Fig. 3: Example Event Pattern and Step Instance Selection

Question 3: (Step Instance Consumption)

Should already occurred steps of a (partial) complex event instance be correlated only with the first or with all matching occurrences of another step?

Example 2: Consider an event description E_2 which searches for any sequence of the following events:

- a specific process p starts executing a program (event type A),
- process p creates a child process (event type B) and
- the child process starts executing a specific program (event type C).

Since a process can create any number of child processes, in this scenario events of type B differ from events of type A and C in a distinct characteristic. After an event instance a_1 has occurred, any subsequent events of type B and C must be correlated with a_1 . This is because the execution of a program by process p (event type A) establishes a state that is not consumed by the creation of a child process (event type B). Instead other child processes can be created by process p . Therefore event type B is *non-consuming* in this scenario; event instance a_1 is never consumed by an event of type B . A trail $t_2 = \{a_1 b_1 c_1 b_2 c_2\}$ triggers two instances $\{a_1 b_1 c_1\}$ and $\{a_1 b_2 c_2\}$ of the complex event E_2 .

The above-mentioned questions characterizes distinct dimension of the event semantics that play a role by specifying complex events. The event pattern defines the complex event to look for, while instance selection describes which of the matching step instances should be bound to the complex event instance. Instance consumption partially controls the further matching of event instances: when a set of steps, that are already bound to a partial complex event instance, is correlated with a matching instance of a step S , consumption defines whether the set of already occurred steps must be kept for correlation with future occurrences of step S or not.

As already mentioned in Section 3.3, step instance consumption constitutes the main difference between event semantics for active databases and misuse detection systems. In the sequel we discuss the characteristics of these dimensions in the context of misuse detection in detail.

5.2 Event pattern

Event descriptions of active database systems and attack signatures differ only slightly regarding the dimension event pattern. This section basically provides a discussion of the respective part of Zimmer's model [9] in the context of misuse detection. Significant differences are explained.

The event pattern of a complex event describes on an abstract level the search pattern, which is used to identify complex event instances within a trail. Five different aspects have to be considered by the event pattern, namely the type and order, repetition, continuity, concurrency and context conditions. Each aspect is discussed in the sequel.

Type and Order

The frame of a complex event is formed by the underlying steps and their order. Steps are typed and can be of a complex or basic event type. The order of steps is defined by a set of operators. Zimmer's model provides a *sequence* (\cdot), *disjunction* (OR), *conjunction* (AND) and *simultaneous* (\parallel) operator. (While Zimmer uses prefix operators we use infix operators for better readability of the examples in this paper.) Zimmer's model also offers a notion of negation but without a distinct negation

operator. We introduce a *negation* operator (*NOT*) to simplify the discussion of our examples (see below).

When specifying attack signatures the sequence operator is probably the most often used one. It describes patterns of attacks that are combinations of several consecutive actions. Instances have to occur in the order determined by the sequence operator. An instance of the type $(A; B; C; D)$ occurs if the step instances of type A , B , C and D occur in the given order.

The disjunction operator can be used to describe patterns of attacks that can be completed by alternative actions. Instances of the type $(A \text{ OR } B \text{ OR } C)$ occur if one instance of one of the given types occurs. Instead of describing each possible sequence of alternative actions, the use of the disjunction operator allows a more compact and concise description of signatures.

Some attacks are composed of sequences of actions where some subsequences can be executed in arbitrary order. Instead of describing attack signatures for all possible combinations of these subsequences the conjunction operator allows to describe this concurrent behavior in an explicit and more compact way. Instances of the type $(A \text{ AND } B \text{ AND } C)$ occur if instances of the types A , B and C have occurred, irrespective of their order. For example assume an attack that manifests themselves in an audit log in one of the following event type sequences: $(A; B; C; D)$ or $(A; C; B; D)$. Here the events of type B and C can occur in any order. Instead of writing signatures for each possible interleaving of the concurrent events, we can specify the corresponding signature using the disjunction operator as follows: $(A; (B \text{ AND } C); D)$.

An instance of the type $(A \parallel B \parallel C)$ occurs if instances of type A , B and C have occurred and their occurrence times are identical. The simultaneous operator is useful when correlating complex events or events coming from different distributed sources. In a distributed environment actions can be executed simultaneously. Further it is possible that instances of different complex event types contain final steps of the same type. Therefore one instance of the event type of the final steps can trigger instances of different complex event types. It is also possible that one final step instance causes the occurrence of several instances of the same complex event type. Since complex events derive their occurrence time from the occurrence time of the final step, events that share the final step instance occur simultaneously. The simultaneous operator is useful to further correlate such event instance in other complex events with respect to their simultaneous occurrence.

The negation operator only makes sense if applied to a time interval [8], which is specified as a sequence of at least two events. Instances of the event type $NOT(A, (B; C; \dots; D))$ occur if the specified sequence of events occur and between the occurrences of the instance of type B and the instance of type D no instance of type A occurs. In the context of signature specifications the negation operator is used to describe exit steps of an event. Exit steps model events that make a partial attack after a set of successful steps but yet before the final step impossible to be completed.

Example 3: Assume an attack that consists of a creation of a specific process (event type A) which executes three distinct actions in a specific order (sequence of events of types B , C and D). The termination of the process (event type E) represents an exit step for this attack. Event description $F := NOT(E, (A; B; C; D))$ matches successful occurrences of this attack.

Repetition

It is useful to be able to specify the number of event instances of a step that have to occur to satisfy the event pattern of the parent event type. This can be specified using a *delimiter* in front of a step declaration. The delimiter can be a number that describes how many step instances *exact* (n), *at-most* ($-n$) or *at-least* ($n-$) are required by the complex event pattern. A delimiter of the form *at-least* n and *at-most* m , with $n < m$, is also possible. If no delimiter is specified the delimiter ($1-$) (at-least one) is assumed. Note that [9] uses the delimiter (0) instead of an explicit negation operator. Without using the negation operator *NOT* the event description F from example 3 would look as follows: $F := A; (0)E; B; (0) E; C; (0) E; D$.

The different forms of a delimiter have distinct semantics in the context of attack signatures and can be used to describe specific attack patterns. The semantics of a delimiter for a step depends on whether the step instances have to occur in a distinct time interval or not. Such a time interval is defined by the predecessor and the successor steps of the step. A step with an at-most delimiter is already occurred if no step instance occurs. Therefore it makes no sense to use an at-most delimiter without a time interval. Further, if used outside of a time interval the semantics of an at-least delimiter is equal to that of the exact delimiter i.e. an instance of event type ($n-$) B occurs when the n th instance of type B occurs.

Delimiters for steps inside a time interval are much more useful. An exact delimiter can be used to describe the pattern of an attack that contains a distinct number of repetitions of a particular action. An instance of type $(A; (n) B; C)$ occurs if after an instance of type A , n instances of type B , and thereafter an instance of C occur. If the $(n+1)$ th instance of B occurs before C is occurred, the partial complex event instance cannot be completed. The $(n+1)$ th instance of B represents an implicit exit step for this event description.

At-least and exact Delimiters can be used to describe patterns of attacks that manifest themselves in a number of repetitions of an action that exceeds a distinct threshold within a specified time interval. Example 4 discusses such an attack signature.

Example 4: A kind of attack, which we call doorknob-rattling attack, consists of a number of repetitive login actions. An attacker can use this procedure to guess the password of an account. As an indicator for the presence of such an attack we use the occurrence of at least six failed logins within 30 seconds. The time interval defined by the first and the last failed login is constrained using a condition to be at most 30 seconds long. Assuming that failed logins are represented as events of type A we can describe the corresponding attack signature using an at-least delimiter as follows: $(A; (4) A; A)$. If we use explicit time events to model this signature it looks as follows: $(T_0; (6-) A; T_{30})$. This event occurs when an event of type T_{30} occurs after at least 6 failed logins occurred within 30 seconds. For this example the event description $NOT(T_{30}, (T_0; (6) A))$ is more appropriate since an instance of this type already occurs when the sixth failed login within 30 seconds occurs.

As mentioned earlier a step with an at-most delimiter already occurs if no step instance occurs. An at-most delimiter does not model the occurrence of events but the non-occurrence of events or events that did not occur often enough. Example 5 discusses a useful application of this delimiter semantics.

Example 5: Two departments of a company use an encrypted channel to exchange electronic information. The security policy of the company states that the encryption keys used for the channel have to be changed three times a day. To be able to detect violations of this policy automatically key changes are documented by generating an event log entry (event type C). The signature to detect these kinds of policy violations looks as follows $(T_1; (-2) C; T_2)$. Instances of this complex event type are caused if two or less instances of type C (key changes) occur in the time interval defined by T_1 (00:00 a.m.) and T_2 (11:59 p.m.).

An alternative event description for this signature is $NOT((3-) C, (T_1; T_2)$. Any event description of the form $(A; (-n) B; C)$ can be transformed to $NOT(((n+1)-) B, (A; C))$.

Steps with an at-least or exact delimiter describe a kind of loop. The step builds the body of the loop and in the case of an exact delimiter this delimiter specifies the break condition of the loop. Here the relevant criterion checked by the break condition is the number of occurrences of step instances. In the case of an at-least delimiter the number of occurrences of the step and the occurrence of the successor step are relevant criteria for the break condition. It can be valuable to be able to use further criteria as break condition. Therefore we extend the model with another kind of delimiter (*conditional delimiter*) that allows the break of a loop depending on the value of aggregated features of the steps. This can be used to model events that represent actions like reading a distinct amount of data from a socket for example. The amount of data can be read by an undefined number of read calls. Instead of counting the read calls the break condition can be based on the sum of the bytes read by the read calls. A corresponding event pattern may look as follows:
 $((\text{until } \text{sum}(R.\text{bytes}) > 1000) R)$ were R models a read call and the number of bytes read by one read call is saved in the feature bytes of the corresponding event. An instance of this complex event is released if more then 1000 bytes were read by one or more read calls.

Continuity

The continuity aspect clarifies the semantics of the event pattern. For example assume an event type $E_2 := (A; B; C)$. Different people may interpret such a pattern in two different ways:

1. At least one step of type A is followed by at least one step of type B that is followed by at-least one step of type C .
2. At least one step of type A is followed by at least one step of type B that is followed by at least one step of type C and there is no event of type C between the steps of type A and B and there is no event of type A between the steps of type B and C .

To clarify the meaning of such a pattern the model distinguishes two different continuity modes *non-continuous* (interpretation 1 and the default) and *continuous* (interpretation 2).

Concurrency

The concurrency modes *overlap* (default) and *non-overlap* define whether the time intervals associated with the steps may overlap or not. For example consider the

complex event type definitions $X := (A; B; C)$, $Y := (D; E; F)$, $Z_1 := \text{non-overlap}(X; Y)$ and $Z_2 := \text{overlap}(X; Y)$. Event pattern Z_1 requires the event instances of type A, B, C, D, E and F to occur in this order, while Z_2 also allows that the sequences of events of type A, B and C and the sequences of events of type D and E occur interleaved. Note that there is a difference between Z_2 and $Z_3 := (X \text{ AND } Y)$. Z_2 requires that step F of event type Y occurs after step C of event type X . In contrast Z_3 allows the step sequence of X and Y to occur completely interleaved. Figure 4 demonstrates the different semantics of the concurrency operators in these patterns. Time constraints expressed by the operators are shown as arrows.

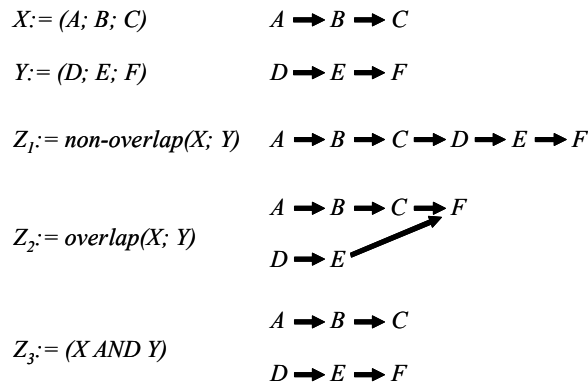


Fig. 4: Differences of concurrency operators

Context condition

Context conditions specify constraints on the context in which steps of a complex event type occur. For each step a set of intra-event conditions can be specified. They can be used for example to constraint an audit event to be caused by a specific user or on a specific host. Further context restrictions can be defined using inter-event conditions. They are useful for example to constrain two or more steps to represent audit events caused by the same process or affecting the same system object. Specifications of the repetition aspect discussed above may use conditional delimiters which represent a special case of context conditions.

5.3 Step instance selection

When a complex event has occurred an event instance selection process has to bind the correct event instances to the steps of the complex event. While the event pattern aspect specifies when a complex event occurs, the instance selection defines which step instances are kept for subsequent event correlations or actions to be executed in response to the occurrence of a complex event. Event pattern and step instance selection are therefore to an extent independent of each other [9], as is also suggested by figure 3.

Zimmer et al [9] proposes a four different instance selection modes. In the context of misuse detection signatures we consider only three of them as useful. We discuss these three modes using the following example.

Example 6: Consider the event pattern $E_3 := (A; B; C)$ where events of type B correspond to log events that periodically document the values of a performance counter e.g. the CPU load or the network load. Further assume a trail $t_2 := \{a_1 b_1 b_2 b_3 c_1\}$. Which instance of type B shall to be selected for the corresponding step? We distinguish the three modes *first* (default), *last* and *all* to control the instance selection process. If we use the mode *first* for the event of type B , trail t_2 cause the instance $\{a_1 b_1 c_1\}$. If we use *last* instead of *first* the instance $\{a_1 b_3 c_1\}$ is caused by t_2 . And if mode *all* is specified for the step of type B t_2 results in an instance $\{a_1 b_1 b_2 b_3 c_1\}$.

All three modes are useful for the specification of attack signatures. Mode *first* can be used if event type B represents a performance value that exceeds a specified threshold, and the caused complex event instance (and the generated alert) shall document the first exceeding of the threshold in the time interval defined by the events of type A and B . If an alert and the occurring complex event instance shall contain the last (current) value of a performance counter, *last* is the appropriate mode. Mode *all* causes all instance of type B that occurred in the defined interval to be contained in the complex event instance.

5.4 Step instance consumption

In Section 3.3 we have discussed that active database systems and misuse detection systems use different instance creation schemes. As a consequence the instance consumption aspect for misuse signatures follows a different paradigm than that proposed by [9].

Following to the occurrence of an instance a_1 of step A an instance b_1 of step B occurs and the partial complex event instance is further completed to $\{a_1 b_1 \dots\}$. If now another instance b_2 of step B occurs, can this instance b_2 together with the instance a_1 build another partial complex event instance $\{a_1 b_2 \dots\}$ or not? The answer to this question can be controlled using the instance consumptions modes. We distinguish two modes, which we call *consuming* (default) and *non-consuming* following the terminology of STATL [1]. If for step B the mode *consuming* is defined then instance b_1 consumes instance a_1 . In this case instance b_2 cannot build another partial complex event instance together with a_1 . In the other case, if step B is defined *non-consuming*, each instance of type B together with any instance of type A that have occurred before build an own partial complex event instance.

Example 7: Consider the complex event $E_3 := (A; \text{non-consuming } B; C)$ and trail $t_3 := \{a_1 a_2 b_1 b_2\}$. Trail t_3 results in four partial event instances of type E_3 , which are $\{a_1 b_1 \dots\}$, $\{a_2 b_1 \dots\}$, $\{a_1 b_2 \dots\}$ and $\{a_2 b_2 \dots\}$.

Which mode has to be chosen for a step is typically defined by the action that is modeled by this step. For example for a complex event that must keep track of a system call sequence of a process the consumption mode of the steps can be derived from the semantics of the action modeled by a step. For example fork calls of a process (creation of a child process) are typically modeled as *non-consuming* steps, because a process can create multiple child processes that can complete the attack. The construction of system objects (e.g. file or process creation) is typically modeled as *non-consuming* step. The *non-consuming* mode is also often found for final steps if certain repeatable event should always trigger a complex event instance.

A process exit call is modeled as a consuming step since a process can be terminated only once. Consuming steps are typically used to describe steps that document object destructions (file deletion, process termination, etc.) or the change of object properties (file renaming). Exit steps are always consuming. The consuming mode is also used in cycles where features of steps are aggregated or passes are counted.

In counting loops (for example in a sequence of sensor events that document values of some performance counters) the semantics is a little bit more complex. We come back to the so-called doorknob-rattling attack already mentioned in example 4. To simplify the discussion we modify the example and now search for three failed logins. Again we assume that an event of type A models a failed login. We discuss the semantics of the complex events $E_5 := (\text{consuming } A; \text{consuming } A; \text{consuming } A)$ and $E_6 := (\text{non-consuming } A; \text{non-consuming } A; \text{non-consuming } A)$ for the trail $t_4 := \{a_1 a_2 a_3 a_4 a_5 a_6\}$.

Since the initial step of E_5 is of type A , for every event instance of trail t_4 a new partial instance of E_5 is created. The overall trail t_4 causes the following four complex event instances of type E_5 to occur $\{a_1 a_2 a_3\}$, $\{a_2 a_3 a_4\}$, $\{a_3 a_4 a_5\}$ and $\{a_4 a_5 a_6\}$ and leaves the following two partial complex event instances $\{a_5, a_6 \dots\}$ and $\{a_6 \dots\}$. E_5 describes all combinations (without repetitions) of three consecutive instances of type A . The number of complex event instances of type A by a trail can be calculated by $n-3-1$ where n is the number of events of type A in the trail. Figure 5 shows which combinations of the event instances of t_4 are correlated by E_5 .

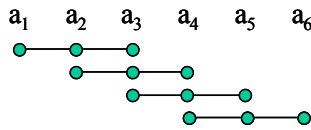


Fig. 5: Instances of E_5

Because of the non-consuming mode of the steps in E_6 much more complex event instances of type E_6 are caused by trail t_4 . This is because E_6 describes all combinations (without repetitions) of three (consecutive or not) instances of type A . Here the number of complex event instances of type E_6 caused by a trail containing n event instances of type A is determined by the binomial coefficient $n \text{ choose } 3$, which is 20. Figure 6 shows which combinations of event instances of t_4 are correlated by E_6 .

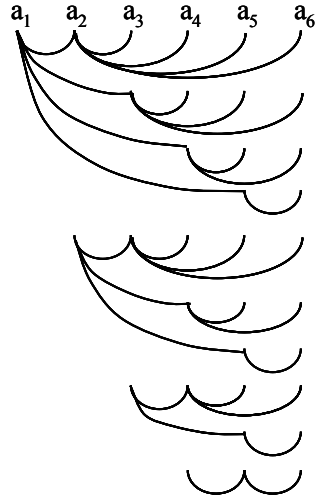


Fig. 6: Instances of E_6

This example shows that an event description consisting only of non-consuming steps does not make much sense, since the number of partial complex event instance is monotonically increasing in this case. Therefore an event description to be useful should contain at least one consuming step.

The above discussion shows that we have to determine via the consumption mode, which combinations should result in an alert. Specifically for the signature discussed in example 4 one may argue that none of consumption modes results in the appropriate event instances since in the described situation only the two event instances $\{a_1 a_2 a_3\}$ and $\{a_4 a_5 a_6\}$ should be released. Therefore we need further semantics of *disjoint instances* of consuming step chains, where each step is only involved in one event instance. To support these semantics we introduced the concept of *exclusive steps*. An event instance that was already bound to a step of any event instance cannot be bound to an exclusive step. Further we assume that if an event instance is checked to be bound to a step, partial complex event instances are always checked before initial steps of new (empty) complex event instances are checked. This means that an event instance can only be bound to an exclusive initial step if it cannot be bound to a step of any partial complex events. Using this concept the event that signals a doorknob rattling attack looks as follows: $E_7 := (\text{exclusive consuming } A; \text{consuming } A; \text{consuming } A)$. The instances of E_7 caused by t_4 are shown in figure 7.

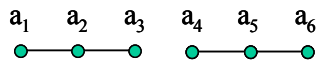


Fig. 7: Instances of E_7

6 Conclusions and future work

In this paper we have presented a model for the semantics of attack signatures that adapts Zimmer's model for event semantics in active database systems [8]. The model divides the semantics in the three dimensions event pattern, step instance selection and step instance consumption. The event pattern specifies the point in time events occur, the step instance selection defines which events are bound to a complex event. Step instance consumption controls, whether already occurred steps of a (partial) complex event instance have to be correlated only with the first or with all matching occurrences of another step. The event pattern is related to a complex event type itself. Step instance selection and step instance consumption are linked to the steps of complex event types.

Since the model presented in this paper systematically the semantic aspects of attack signatures, it contributes to the ongoing work in the area of misuse detection systems in several ways:

- The model provides a solid and in-depth understanding of properties of complex signature events.
- It explicitly mentions and distinguishes the different aspects of signature descriptions and thereby supports a systematic development of attack signatures.
- It offers a solid basis for the development of detection languages.
- To our knowledge no systematic comparison of existing detection languages regarding their expressiveness in distinct semantic aspects of signatures has been done yet. The presented model can be used as kind of checklist for such a comparison of detection languages.

Further we have discussed some semantic aspects of attack signatures, which have not been taken into account by existing detection languages. Only a few languages provide a conjunction operator for the definition of the event pattern. To our knowledge Lambda [3] is the only language that supports a concept similar to the simultaneous operator. The dimension step instance consumption was first considered by the language STATL [1] and the language of the IDIOT IDS [14], but in a restricted sense. None of the existing languages supports different step instance selection modes and none of these languages allows the specification of the doorknob-rattling attack signature using disjoint instances semantics as discussed in Section 5.4.

We have not presented a complete formal syntax for the specification of the semantics of complex signatures, since the syntactical constructs used in this paper are intended for explanation purposes only. A challenging task for future work is to develop a controllable and comfortable detection language that supports all aspects of semantics of complex events.

For some examples we discussed different descriptions using different operators of the model. Another topic for future work is the development of a kind of a cost model for the operators of the model that represents the resources required to analyze patterns using the operators. Such a cost model could be used to compare the resources required to analyze different descriptions of a signature.

References

1. Eckmann, Steven T.; Vigna, Giovanni; Kemmerer, Richard A.: STATL: an Attack Language for State-based Intrusion Detection, in Proc. of the ACM Workshop on Intrusion Detection, Athens, Greece, November 2000.
2. Michel, Cedric; Me, Ludovic: ADeLe: an Attack Description Language for Knowledge-based Intrusion Detection, in Proc. of the International Conference on Information Security, Kluwer, June 2001.
3. Cuppens, Frederic; Ortalo, Rodolphe: LAMBDA: A Language to Model a Database for Detection of Attacks. In: Debar, Hervé; Mé, Ludovic; Wu, Felix S.: Proc. of the Third International Workshop on Recent Advances in Intrusion Detection, LNCS 1907, Springer, 2000, pp. 197-216.
4. Pouzol, Jean-Philippe ; Ducassé, Mireille : From Declarative Signatures to Misuse IDS. In: Lee, Wenke; Mé, Ludovic; Wespi, Andreas (Eds.): Proc. of the Fourth International Symposium on Recent Advances in Intrusion Detection, LNCS 2212, Springer, 2001, pp. 1-21.
5. Meier, Michael; Bischof, Niels; Holz, Thomas: SHEDEL - A Simple Hierarchical Event Description Language for Specifying Attack Signatures. In: Ghonaimy, M. Adeb; El-Haddidi, Mahmoud T.; Aslan, Heba K.: Proceedings of the 17th International Conference on Information Security. Kluwer, 2002, pp. 559-571.
6. Mounji, Abdelaziz: Languages and Tools for Rule-Based Distributed Intrusion Detection, PhD thesis, Computer Security Institute, University of Namur, Belgium, September 1997.
7. Lindqvist, Ulf; Porras, Phillip 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.
8. Zimmer, Detlef: A Meta-Model for the Definition of the Semantics of Complex Events in Active Database Management Systems (in German). PhD Thesis, University of Paderborn, Shaker-Verlag, ISBN: 3-8265-3744-0, 1998.
9. Zimmer, Detlef; Unland, Rainer: On the Semantics of Complex Events in Active Database Management Systems. In Proc. Of the 15th International Conference on Data Engineering, IEEE Computer Society Press, 1999, pp. 392-399.
10. Vigna, Giovanni; Eckmann, Steven T.; Kemmerer, Richard A.: Attack Languages, in: Proc. of the IEEE Information Survivability Workshop, Boston, MA, October 2000.
11. Sun Microsystems, Inc.: SunSHIELD Basic Security Module Guide, Solaris 2.6 System Administrator Collection Vol 1, Mountain View, CA, 1997.
12. <http://www.tcpdump.org>
13. Vigna, Giovanni; Eckmann, Steven T.; Kemmerer, Richard A.: The STAT Tool Suite, in Proc. of DISCEX, Hilton Head, South Carolina, IEEE Computer Society Press, January, 2000.
14. Kumar, Sandeep: Classification and Detection of Computer Intrusions, PhD Thesis, Purdue University, 1995.
15. Dayal, U.,; Buchmann, A.; Chakravarthy, S.: The HiPAC Project. In: Widom, J.,; Ceri, S.: Active Database Systems. Morgan Kaufmann, ISBN: 1-55860-304-2, 1996.
16. Chakravarthy, S.; Krishnaprasad, V.; Anwar, E.; Kim, S.: Composite Events for Active Databases: Semantics, Contexts and Detection. In Proc. of the 20th International Conference on Very Large Databases, 1994, pp. 606-617.
17. Collet, C.; Coupaye, T.: Composite Events in NAOS. In Proc. Of the seventh International Conference on Database and Expert Systems Applications. 1996, pp. 475-481.
18. Berndtsson, M.: ACCOD: An approach to an Active Object-Oriented DBMS. Master thesis, University of Skövde, 1991.

19. Ning, P.; Jajodia, S.; Wang, X. S.: Abstraction-Based Intrusion Detection In Distributed Environments. In: *ACM Transactions on Information and System Security*, Vol. 4, No. 4, November 2001, pp. 407-452.