Identifying Modeling Errors in Signatures by Model Checking

Sebastian Schmerl, Michael Vogel and Hartmut König
Brandenburg University of Technology Cottbus
Computer Science Department,
P.O. Box 10 13 44,
03013 Cottbus, Germany
{sbs,mv,koenig}//at//informatik.tu-cottbus.de

Abstract: Most intrusion detection systems deployed today apply misuse detection as analysis method. Misuse detection searches for attack traces in the recorded audit data using predefined patterns. The matching rules are called signatures. The definition of signatures is up to now an empirical process based on expert knowledge and experience. The analysis success and accordingly the acceptance of intrusion detection systems in general depend essentially on the topicality of the deployed signatures. Methods for a systematic development of signatures have scarcely been reported yet, so the modeling of a new signature is a time-consuming, cumbersome, and error-prone process. The modeled signatures have to be validated and corrected to improve their quality. So far only signature testing is applied for this. Signature testing is still a rather empirical and time-consuming process to detect modeling errors. In this paper we present the first approach for verifying signature specifications using the SPIN model checker. The signatures are modeled in the specification language EDL which leans on colored Petri nets. We show how the signature specification is transformed into a PROMELA model and how characteristic specification errors can be found by SPIN.

Keywords: Computer Security, Intrusion Detection, Misuse Detection, Attack Signatures, Signature Verification, PROMELA, SPIN model checker

1 Motivation

The increasing dependence of human society on information technology (IT) systems requires appropriate measures to cope with their misuse. The enlarging technological complexity of IT systems increases the range of threats to endanger them. Besides traditional preventive security measures, such as encryption, authentication, access control mechanisms, etc, reactive approaches are more and more applied to counter these threats. Reactive approaches allow responses and counter-measures to security violations to prevent further damage. Intrusion detection systems (IDSs) have proved as one of the most important means to protect IT-systems. A wide range of commercial intrusion detection products is available, especially for misuse detection. Intrusion detection is based on the monitoring of IT-systems to detect security violations. The decision which activities are considered as security violations in a given context is defined by the used security policy. Two main complementary approaches are applied: anomaly and misuse detection. *Anomaly detection* aims at the exposure of abnormal user behavior. It requires a comprehensive set of data describing the normal user behavior. Although much research has been done in this area, it has still

a limited practical importance because it is difficult to provide appropriate profile data. *Misuse detection* focuses on the (automated) detection of known attacks described by patterns which are used to identify an attack in an audit data stream. The matching rules are called *signatures*. Misuse detection is applied by the majority of the systems used in practice.

The detection power of misuse detection though is still limited. First of all many intrusion detection systems are dedicated to the detection of simple structured network attacks, often still in a post-mortem mode. These are simple single-step attacks and the detection process is mainly a pattern matching process. Sophisticated multi-step or even distributed attacks, which are applied to intrude in dedicated computer systems, are not covered. These attacks are getting an increasing importance, especially in host based intrusion detection.

The crucial factors for high detection rates in misuse detection are the accuracy and the topicality of the signatures used in the analysis process. Imprecise signatures confine strongly the detection capability and cause false positives or false negatives. The former trigger not desired false alarms, while the latter represent undetected attacks. This lack of reliability together with high false alarm rates has questioned the efficiency of intrusion detection systems in practice [8]. The reasons for this detection inaccuracy lie in the signature derivation process itself rather than in the quality of the monitored audit data. Signatures are derived from an exploit. This is the program that executes the attack. The latter represents a sequence of actions that exploit security vulnerabilities in an application, an operating system, or a network. The signatures, in contrast, describe rules, how traces of these actions can be found in an audit or network data stream. In practice, signatures are derived empirically based on expert knowledge and experience. Methods for a systematic derivation have scarcely been reported yet. Automated approaches for reusing design and modeling decisions of available signatures do not exist, yet. Therefore, new signatures are still manually derived. The modeling process is time consuming and the resulting signatures often contain errors. In order to identify these errors the new modeled signatures have to be validated, corrected, and improved iteratively. This process can take several months until the corrected signature is found. As long as this process is not finished the affect systems are vulnerable to the related attack because the intrusion detection system cannot protect them. This period is therefore also called vulnerability window.

Although signatures are not derived systematically, they are usually described in a formal way, e.g. as finite state machines. Examples of such signature description languages are STATL [5], [3], Bro [6], IDIOT [7], and EDL [2], which define a strict semantic for the signatures. These languages though are mostly related to a concrete intrusion detection system. Astonishingly, these languages have not been used for the verification of signatures. The main validation method for signatures in practice is testing which proves with the help of an intrusion detection system, whether the derived signature is capable to exactly detect the related attack in an audit trail. For this, the signatures are applied to various audit trails. To test the different features of a signature, test cases are derived, which modify the signature to reveal detection deficits. Signature testing is a heuristic process. There exists no methodology like in protocol testing. Only some first approaches are reported [4]. Signature testing is a time consuming and costly process which requires manuals steps to derive test cases and to evaluate the test outcome. Testing is, however, not the right process to identify errors in signature modeling. Many of these errors may be already found by verifying the modeled signature. The objective of a signature verification stage should be to prove, whether the modeled signature is actually capable to detect the attack in an audit trail and to ensure that the signature is not in conflict with itself. Typical errors in signature modeling are mutually exclusive constraints, tautologies, or constraints which will be never checked.

In this paper we present the first approach for the verification of signatures. It aims at the verification of multi-step signatures which are described in EDL [2] to demonstrate the principle. EDL supports the specification of complex multi-step attacks and possesses a high expressiveness [1] and nevertheless allows efficient analysis. For the verification, we use the model checker SPIN [11]. We choose SPIN because it supports large state spaces, provides a good tool performance, and is well documented. The transformation of EDL signature specifications into PROMELA is the kernel of this approach. We provide rules how this transformation has to be performed. The verification proves the absence of typical specification errors. These are formulated by linear temporal logic (LTL) conditions which are generated depending on the concrete signature under verification. The remainder of the paper is structured as follows. In Section 2 we consider the signature derivation process. We shortly introduce the signature modeling language EDL and outline the reasons for specification errors when modeling signatures. Section 3 describes the semantic equivalent transformation of EDL into PROMELA. We further show that the translation into PROMELA, which has a well defined semantics, is another way to give a formal semantics to a signature model. In Section 4 we present the verification procedure and show how typical specification errors can be detected. Thereafter in Section 5 we give an overview of a concrete evaluation example. Some final remarks conclude the paper.

2 On the modeling of complex signatures

An attack consists of a sequence of related security relevant actions or events which must be executed in the attacked system. This may be, for example, a sequence of system calls or network packets. These sequences usually consist of several events which form complex patterns. A *signature* of an attack describes criteria (patterns) which must be fulfilled to identify the manifestation of an attack in an audit trail. All the relations and constraints between the attack events must be modeled in the signature. A signature description which correlates several events can readily possess more than 30 constraints. This leads to very complex signatures. In addition, it is possible that several attacks of the same type are executed simultaneously and proceed independently, so that different instances of an attack have to be distinguished as well. This fact raises the complexity of the analysis.

To our knowledge there have been no approaches to identify specification errors in signatures by verification. This is remarkable due the fact that most signature languages have an underlying strict semantic model (e.g. STATL [5], Bro [6], EDL [2]). The approach demonstrated here uses the signature description language EDL (*Event Description Language*) as example of a signature modeling language. EDL leans on colored Petri nets and supports a more detailed modeling of signatures compared to other modeling languages. In particular, it allows a detailed modeling of constraints which must be fulfilled in transitions for attack progress. The definition of EDL is given in [2], the semantic model is described in [1]. Before describing the transformations of EDL signatures into PROMELA models, we outline the essential features of EDL.

2.1 Modeling signatures with EDL

The description of signatures in EDL consists of places and transitions which are connected by directed edges. *Places* represent states of the system which the related attack has

to traverse. *Transitions* represent state changes which are triggered by events e.g. security relevant actions. These events are contained in the audit data stream recorded during attack execution. The progress of an attack, which corresponds to a signature execution, is represented by a *token* which flows from state to state. A token can be labeled with features as in colored Petri nets. The values of these features are assigned when the token passes the places. Several tokens can exist simultaneously. They represent different signature instances.

Places describe the relevant system states of an attack. They are characterized by a set of features and a place type. Features specify the properties located in a place and their types. The values of these properties are assigned to the token. The information contained in a token can change from place to place. EDL distinguishes four place types: *initial*, *interior*, *escape*, and *exit places*. *Initial places* are the starting places of a signature (and thus of the attack). They are marked with an initial token at the beginning of the analysis. Each signature has exactly one *exit place* that describes the final place of the signature. If a token reaches this place the signature has identified a manifestation of an attack in the audit data stream, i.e. the attack has performed successfully. *Escape places* stop the analysis of an attack instance because events have occurred which make the completion of the attack impossible, i.e. the observed behavior represents normal, allowed behavior but not an attack. Tokens which reach these places are discarded. All other places are *interior places*. Figure 1 shows a simple signature with four places P_1 to P_4 for illustration.

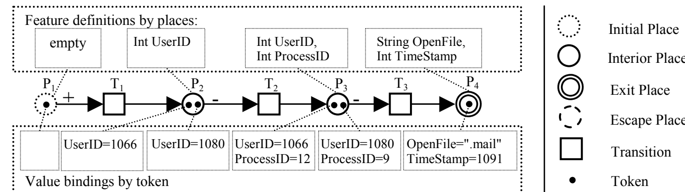

Figure 1: Places and features

Transitions represent events which trigger state changes of signature instances. A transition is characterized by input places, output places, event type, conditions, feature mappings, consumption mode, and actions. *Input places* of transition *T* are places with an edge leading to the transition *T*. They describe the required state of the system before the transition can fire. *Output places* of transition *T* are places with an incoming edge from the transition *T*. They characterize the system state after the transition has fired. A change between two system states requires a security relevant event. Therefore each transition is associated with an event type. The firing of a transition can further depend on additional conditions which specify relations over certain features of the event (e.g. user name) and their assigned values (e.g. root). Conditions can require distinct relationships between events and token features on input places (e.g. same values).

If a transition fires, tokens are created on the output places of the transition. They describe the new system state. To bind values to the features of the new tokens, the transitions contain *feature mappings*. These are bindings which can be parameterized with constants, references to event features, or references to input place features. The *consumption mode* (cf. [1]) of a transition controls, whether tokens that activate the transition remain on the input places after the transition fired. This mode can be individually defined for each input place. The consumption mode can be considered as a property of a connecting edge between input

place and transition (indicated by “–“ or “+”). Only in the consuming case the tokens which activate the transition are deleted on the input places.

Figure 2 illustrates the properties of a transition. The transition T_1 contains two conditions. The first condition requires that feature *Type* of event *E* contains the value *FileCreate*. The second condition compares feature *UserID* of input place P_1, referenced by *“P1.UserID”*, and feature *EUserID* of event type *E,* referenced by *“EUserID”*. This condition demands that the value of feature *UserID* of tokens on input place P_1 is equal to the value of event feature *EUserID*. Transition T_1 contains two feature mappings. The first one binds the feature *UserID* of the new token on the output place P_2 with the value of the homonymous feature of the transition activating token on place P_1. The second one maps the feature *Name* from the new token on place P_2 to event feature *EName* of the transition triggering event of type *E*.

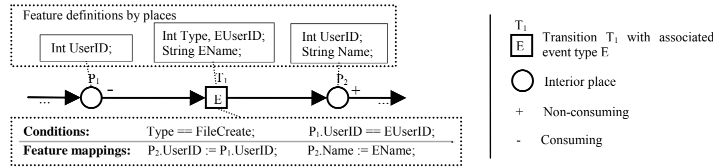

Figure 2: Transition properties

Firing Rule. In contrast to Petri nets, in EDL all transitions are triggered in a deterministic conflict free manner. First, all transitions are evaluated to determine active transitions for which all conditions for firing are fulfilled. The active transitions are triggered at the same time. So there is no token conflict. Figure 3 illustrates the triggering rule with two examples signatures. The left side of the figure shows the marking before an event of type *E* occurs and the right side shows the signature state after firing. None of the depicted transitions have an additional transition condition.

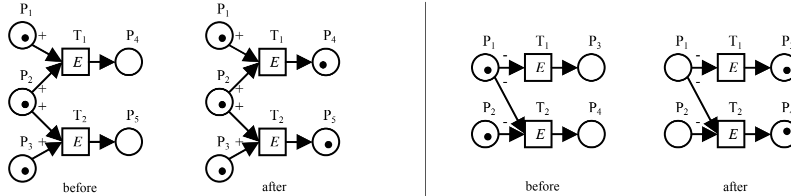

Figure 3: Conflict situations

Even though EDL is a very intuitive signature language new modeled signature frequently contain specification errors, such as unreachable places or transition conditions, which can never be fulfilled (mutually exclusive conditions). Other possible errors are a non continuous token flow from the initial place to the exit place or unreachable escape places to stop an attack tracking due to faulty feature mappings. Such specification errors lead to inaccurate signatures causing false positives or negatives, respectively. Many of these inaccuracies can hardly or not all be identified by static signature analyses. Therefore we decided to apply model checking to detect these types of errors.

3 Transformation of EDL signatures into PROMELA specifications

Before verifying EDL signatures with SPIN, we have to transform them into semantic equivalent PROMELA specifications. The transformation rules needed for it are described in this section. The challenge of this transformation consists in an appropriate abstraction of the EDL signatures to achieve a limited state space. An one-to-one transformation would result in an infinite state space, since the number of tokens in EDL is not limited because each ongoing attack has to be pursued. In addition, the value range of features is not limited as well. Therefore the EDL signatures have to be abstracted, but the essential semantics of the signatures must be preserved. Only thus, it is possible to detect errors in their specification and to have a limited state space.

3.1 Overview

The basic elements of PROMELA are processes, message channels, and variables. *Processes* specify concurrent instances in the PROMELA language. The execution of a process can be interrupted by another process at any time, if the process execution mode is not marked as atomic. Processes allow the specification of *sequentially executed* statements, the *iterative execution* of a block of instructions (do-loop) and *conditional execution* of instructions. Processes communicate among each other via global variables or message channels. *Messages* can be read from and written to *channels*. A channel operates like a queue. A message comprises a tuple of basic PROMELA data types.

The transformation of EDL signatures into PROMELA models is transition driven. All EDL transitions are mapped into PROMELA processes which model the semantic behaviour of the transitions. EDL places are mapped to message channels and tokens to messages correspondingly. A process *P* of transition *T* evaluates messages from those message channels, the input places of *T* are mapped onto. If all required conditions for firing a transition *T* are fulfilled, process *P* generates new messages and writes them into those message channels the output places of *T* are mapped onto.

In the following, the transformation of EDL signatures to PROMELA models will be described in detail. At first the transformation of places and transitions will be described then the realisation of the triggering rule to PROMELA will be explained.

3.2 Transformation of EDL places

The conversion to a PROMELA model starts with the definition of channels for all EDL places. The channel stores the tokens of the corresponding EDL place as messages. In EDL the tokens on a place describe the state of an attack on the observed system. Therefore, a place defines a set of features that describe the system state. The definition of a feature consists of (a) the features type (*bool*, *string*, *number*, *float*…), i.e. the values range the tokens may have on this place. Further the feature definition defines (b) the feature identifier for referring features in conditions of transitions. Thus the set of feature definitions of a place can be written as a tuple. This tuple can be directly adopted by the PROMELA model by defining the message type of the corresponding channel. Only slightly changes are needed to feature type definitions, except for EDL strings. They are mapped to fixed sized byte ar-

rays. The size of these arrays (MAXSTRING) and the maximum message capacity of a channel (CHANNELSIZE) will be set accordingly to the EDL signature (see Section 4.1).

As an example, Table 1 shows the conversion of two EDL places "*link_no_prefix*" and "*exit_place*" to PROMELA. The different place types (*initial*, *interior*, *escape*, and *exit places*) remain unconsidered during initial transformation to PROMELA because the different semantics of place types will be considered by the implementation of the firing rule.

<table>
<tr><th>EDL</th><th>PROMELA</th></tr>
<tr><td><pre>
link_no_prefix
  {
  TYPE INTERIOR
  FEATURES
      STRING mLinkName,
      STRING mScriptName,
      INTmScriptOwner
  }

exit_place
{
  TYPE EXIT
  FEATURES
    STRING mScriptName,
    INTmExecutorID
  }
</pre></td><td><pre>
typedef F_LinkNoPrefix
{
  byte mLinkName[MAXSTRING];
  byte mScriptName[MAXSTRING];
  int mScriptOwner;
};
chan LinkNoPrefix = [CHANNELSIZE] of { F_LinkNoPrefix };

typedef F_ExitPlace
{
  byte mScriptName[MAXSTRING];
  int mExecutorID;
};
chan ExitPlace = [CHANNELSIZE] of { F_ExitPlace };
</pre></td></tr>
</table>

Table 1: Transformation of EDL places into PROMELA channels

3.3 Transformation of EDL transitions

The topology of the transitions and places in an EDL signature defines the temporal order of the occurrence of events during an attack. A single transition specifies the type of the event that triggers the transition, additional conditions of this event, and the tokens (values of token features) on the input places of the transition. (see Table 2a). The evaluation of a transition T begins with the examination whether the type of an incoming event X corresponds to the event type associated with the transition. Furthermore, the transition condition has to be evaluated in relation to event X for all combinations of tokens from input places of the transition. Here, a token combination is an n-tuple, with n= number of input places of T and every element t_i of the tuple represents a token of place P_i. If all transition conditions are fulfilled for a token combination the transition will be fired creating new tokens with new feature values on the output places.

The transformation of EDL transitions into the PROMELA model starts with the definition of separate process types for all transitions. A process of a specific type assumes that the incoming event is stored in the global variable *gEvent* and its type in the variable *gEventType*. The process interprets the messages on channels representing the input places of the transition (in the following denoted as *input place channels*) as tokens. The process starts with the event evaluation and checks the type of the current event. In the example of Table 2a the EDL transition "*LinkWithPrefix(+) ExitPlace*" requires the occurrence of a "*SolarisAudiEvent*" (line 3). Therefore the respective PROMELA process "*LinkWithPrefix_ ExitPlace*" in Table 2b checks the event type in the *if*-condition in line 4. If the condition is fulfilled the process iterates over all messages on all its input place channels. In the example this is implemented by the *do*-loop in lines 9-39. Thereafter one message from the input place channel is read out (line 12), evaluated (lines 14-33), and finally written back to the channel (line 36). The channels implement a FIFO buffer behavior. If messages from more

than one input place channel have to be evaluated the iterations on all message combinations are performed by additional nested *do*-loops. The evaluation of the transition conditions are mapped directly onto PROMELA *if*-conditions (lines 14-18). Here, the currently considered combination of messages from the input place channels (only *lF_LWP* in the example because it is a single input place channel) is checked in relation to the current event (*gEvent*), whether all conditions are fulfilled. In this case and if the EDL transition is consuming those messages which fulfill the conditions are placed (line 27) for later removal in an auxiliary channel (*LinkWithPrefixDelete*) corresponding to the input place channel. Further new messages are created for all output place channels (line 20), the feature values of the messages are set (line 22-23), and the messages are written (line 29) in an additional auxiliary channel (*ExitPlaceInsert*) for a later insertion into the output place channel. The process terminates after all messages have been evaluated in relation with current event (*gEvent*). Finally there are tokens in the auxiliary channel (*…Insert*) to output place channels of the transition and tokens to be removed in the auxiliary channels (*…Delete*) of the input place channels of the transition.

a) EDL syntax

```
01 LinkWithPrefix(-) ExitPlace
02 {
03   TYPE SolarisAuditEvent
04   CONDITIONS
05     (eventnr==7) OR (eventnr==23),
06     LinkWithPrefix.mLinkName==RNAME,
07     euid != audit_id
08
09 MAPPINGS
10     [ExitPlace].mScriptName =
LinkWithPrefix.mScriptName,
11     [ExitPlace].mExcecutorID = euid
12   ACTIONS
13 ...
14 }
```

b) PROMELA syntax

```
01 proctype LinkWithPrefix_ExitPlace (){
02 atomic{
03 if ::
04 (gEventType==SolarisAuditEvent)
05 ->
06 F_LinkWithPrefix lF_LWP;
07
08 int lNrOfToken = len(LinkWithPrefix)
09 do
10 :: (lNrOfToken > 0) ->
12   LinkWithPrefix?lF_LWP;
13   /* checking Conditions */
14   if
15   :: ((gEvent.eventnr == 7) ||
16        (gEvent.eventnr == 23))
17   && (STRCMP(lF_LWP.mLinkName,gEvent.rname))
18   && (gEvent.euid != gEvent.audit_id) ->
19     /* Create a new Message */
20     F_ExitPlace lF_EP;
21     /* setting values of the new message */
22     STRCPY(lF_EP.mScriptName, lF_LWP.mScriptName);
23     lF_EP. mExcecutorID = gEvent.euid;
24
25     /* Transition is Consuming, */
26     /* so mark the message for delete*/
27     LinkWithPrefixDelete!lF_LWP;
28     /*save new message to insert it later */
29     ExitPlaceInsert!lF_LNP_new;
31     lNrOfToken--;
32   :: else /* do nothing */
33   fi;
34   /*put back current message to Channel */
36   LinkWithPrefix!lF_LWP;
37 :: else ->
38   break;
39 od;
40 :: else -> skip;
41 fi;
42 }
43 }
```

Table 2a/b: EDL transition and the related PROMELA process

3.4 Implementation of the deterministic triggering rule of EDL

EDL applies a deterministic firing rule which is conflict free as described in Section 2.1. The implementation of this rule is implicitly given in PROMELA. To guarantee a conflict free event evaluation every event is evaluated by applying the following four steps. (1) Reading out the current event. Depending on the event its values are written to the variable structure *gEvent* and the event type is stored in *gEventType*. After that (2) a process instance is created for all EDL transitions which corresponds to the process type of the respective transition as described in Section 3.3. These processes check sequentially the conditions of the EDL transitions in relation to the messages in the input place channels and the current event. The execution order of the processes is non-deterministic, but the processes are executed atomically (see atomic stmt. line 2, Table 2b), i.e. the processes cannot interrupt each other. This fact as well as the implementation principle of the processes, to store newly created messages and messages to be deleted in auxiliary channels first, ensure a conflict free firing rule. Thus, every process leaves the input data (messages) unchanged after analyzing them. When all transition processes have terminated (3) all messages to be removed are deleted from the channels and finally (4) all new messages are inserted. This is done by iteration on the messages in the auxiliary channels (*…_Delete* and *…_Insert*).

Figure 4(a) shows an example of an EDL signature fragment. None of the depicted transitions possess additional transition conditions. The depicted marking triggers the transitions T_1, T_2 and T_3, when the associated event E occurs. Figure 4(b) shows the corresponding PROMELA model after all processes have evaluated event E (completed step 2). The arrows indicate message readings and writings. To accomplish the evaluation of all processes, first process T_1 (for simplicity reasons we assume T_1 is the process evaluated first) reads and evaluates all messages from input channel P_1 message by message. A *read* message is always written back to the channel (dotted lines) to leave the messages "unattended" for the processes T_2 and T_3. Process *T1* inserts for each message from p_1 a new message into the p_2 insert channel (solid lines) and a copy of the *read* message from p_1 to *p1_delete* (solid lines) because of the consuming mode. After evaluating all other processes in the same manner the messages in the *delete* channels (*p1_delete*, *p2_delete*, *p3_delete*) are removed from the channels p_1, p_2 and p_3 in step 3. Finally in step 4 the new messages from the channels *p1_insert*, *p2_insert* and *p3_insert* are transferred to the respective channels p_1, p_2, and p_3. Figure 4(c) shows the situation after the evaluation of event E. The use of the auxiliary insert channels prevents in this example that T_2 evaluates the new messages generated by T_1 for the same event E. The *..._delete* channels are responsible for the conflict free message processing of T_1 and T_3.

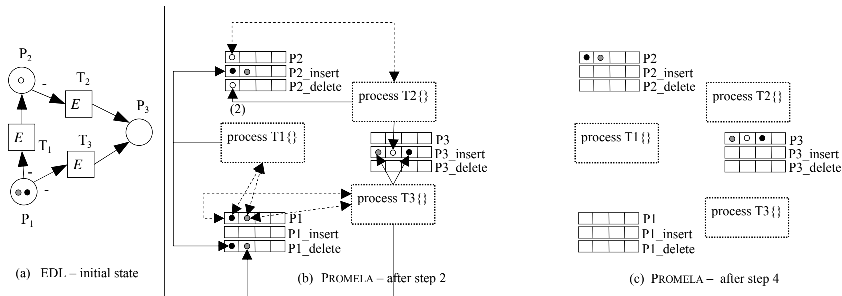

Figure 4: Transformation of an EDL signature into PROMELA

4 Signature verification

In this section we show how SPIN is used to identify typical specification errors in signatures. First we describe how we deal with events triggering transitions. We describe the decomposition of events in equivalence classes, the generation of new events, and the determination of the smallest possible channel size and string length to minimize the state space. Next we explain a number of typical specification errors. We formulate LTL conditions to prove their absence in the EDL modeling.

4.1 Setting up the verification

To verify signature properties we have to analyze the behavior of the signature in relation to the occurring events because they affect the firing of the transitions, the creation of new tokens, and their deletion. An exhausting verification of a signature specification considering all possible events is not possible because of the huge or even infinite number of potential events. Typically events possess about 15 different features, such as timestamp, user ID, group ID, process ID, resource names, and so forth. In order to limit the state space in this transformation all possible events are divided in equivalence classes. The partition is determined by the transitions conditions of the signature. We divide all transition conditions in atomic conditions. An equivalence class is set up for each combination of fulfilled and not fulfilled atomic conditions. A representative for each class is chosen which is used for the verification of the signature properties. We can limit the verification to the representatives without loss of generality, since all events of an equivalence class influence the signature in exact same way. The determination of the equivalence classes is accomplished by splitting up complex transition conditions into atomic conditions. For this purpose, we have to split the EDL conditions by means of AND- and OR-expressions. Then we use a constraint solver to determine a concrete event for each class. For this, all atomic conditions are entered into the constraint solver in negated and non-negated form each which then evaluates the constraints and calculates concrete values for all features of an event class. So an automatic generation of all representatives is feasible. If there is no solution for a combination of negated or non-negated conditions so this constraint represents features which mutually exclude each other. Such a class is deleted from verification.

In order to verify the demanded signature properties we analyze the signature with all generated event representatives. For this, (1) an equivalence class is randomly selected and the representative of the class is the new occurring event. Next (2) the corresponding PROMELA process is started for each EDL transition. These processes analyze the current selected representative event in relation to the messages on the input channels. If needed, new messages are created for a later insertion or deletion. After that (3) the message channels, the insert channels and delete channels are updated in the manner described in Section 3.4. SPIN generates the full state space by successively applying these three steps. The state space contains e.g. all produced messages, all executed instructions, all variable settings and thus the complete behavior of the signature. Based on this state space we can verify signature properties and identify signature specification errors.

The size of the state space is the crucial factor for the usability of the described approach. If the state space is too large the model checker needs too many resources (computing time, memory size) for verification. The size of the state space is not only determined by the number of equivalence classes, but also by the number of messages in the channels.

This is why the `CHANNELSIZE` (maximum number of messages in a channel, cf. Section 3.2) should be minimized. Without loss of generality, the unlimited number of tokens on a place P can be limited to the maximum number n of incoming edges of a transition T from same place P. In most cases n is one, except the signature has multiple edges (also called parallel edges) between an input place P and a transition T. Only in such cases the transition correlates several tokens on a single place. However, such topologies are very unusual for signatures. In most cases a transition process correlates only one message per channel. Since the complete state space generated by SPIN covers already all possible token combinations, we can limit the number of messages in the channels this way. More messages per channels only lead to states which represent combinations of existing states.

If strings are used in an EDL signature then the length `MAXSTRING` (cf. Section 3.2) of the corresponding PROMELA byte arrays must be specified. PROMELA does not allow dynamical memory allocation; therefore we must estimate the required array length beforehand. The defined byte array length does not affect the number of states calculated by SPIN, but it does influence the size of the state vectors in the state space. A state vector contains information on the global variables, contents of each channel, process counter, and local variables of each process. Consequently, the string size (`MAXSTRING`) should be specified as low as possible. It is first and foremost determined by the largest string (*`max_str_event`*) of all event class representatives. If the signature does not apply string concatenation we can automatically estimate `MAXSTRING` by *`max_str_event`* as upper bound. If string concatenation is used in an EDL signature without cycles then we can limit the `MAXSTRING` to *`max_str_event`*`*2`. In the rare case of a string concatenation in a cycle, `MAXSTRING` must be defined sufficiently by estimating the number of expected cycles.

4.2 Signature properties

Now we consider the properties which have to be fulfilled by each signature. If these properties are violated a specification error will be indicated. The properties to be fulfilled are specified as LTL formulas. These properties are verified by means of the model checker SPIN.

Tracking new attack instances: The signature always have to be able to track new starting attack instances. An attack instance denotes an independent and distinct attack. These new attack instances can start with any new event and have to be tracked simultaneously. With each new occurring event, a token must be located on each init place of the signature to ensure a simultaneous attack tracking. If a channel (C_I) represents an initial place (I) of a signature C_I must contain at least one message, each time the processes T_i representing the transition are started. This behavior can be expressed in LTL as follows: $\Diamond p \Rightarrow (a \mathcal{U} p)$ with $a = \text{len}(C_I)$, where $\text{len}(C_I)$ is the number of messages in channel C_I, and $p \equiv$true, iff a process P_i is running.

Unreachable system states: The places in a signature model represent relevant system states during an attack. If a token is never located on a certain place then this system state will be never reached. Accordingly the signature possesses a faulty linked topology of places and transitions or the modeled place is redundant. We specify the property that each place ($P_1, \ldots, P_n$) should contain at least once a token as a LTL condition over the corresponding channels ($c_{P1}, \ldots, c_{Pn}$): $\Diamond\, t_{CP1} \wedge \Diamond\, t_{CP2} \wedge \ldots \wedge \Diamond\, t_{CPn}$, with $t_{CPi} = (\text{len}(c_{Pi}) > 0)$ where c_{Pi} represents the place P_i.

Dead system state changes: In the same way system states are modeled by places. Changes in system states are specified by transitions. If a transition never fires this means that the system state change from the input to the output places of the transition is never accomplished. The reasons for never firing transitions are either wrongly specified transition conditions or the lack of tokens on the input places. Lacking tokens can be identified by the "*unreachable system states*" property. Transitions which will never fire because of wrongly specified transition conditions can be identified by unreachable code in the PROMELA process of a transition. If the state space is exhaustively generated and the statements for creating and mapping new messages (e.g. line 20 in Table2b) in a transition process are never reached then this transition will never fire. The determination of unreachable code is a standard verification option in SPIN.

Twice triggering token event combinations: If two transitions T_1 and T_2 have the same input place P and T_1 and T_2 are triggered by the same event as well as same token t on input place P then the signature allows for *twice triggering token event combinations*. This behavior means: it is possible that a single action/event transfers a single signature instance in two different system states. The reason for this is either the transitions conditions on T_1 and/or T_2 are underspecified, or T_1 and T_2 are modeling an implicit fork transition. If this behavior is intended an explicit fork transition T_F should be used instead of two transitions T_1 and T_2 (cf. Figure 5). Otherwise the transition conditions of T_1 and T_2 should be refined in such a way that T_1 and T_2 are not triggering for the same event and the same token t. The usage of implicit fork transitions should be avoided for following two reasons: (1) the fork behavior can not be seen directly in the signature topology of places and transitions and (2) implicit fork transitions need additional conditions for correct behavior. Both issues raise the signature complexity and increase the error-proneness.

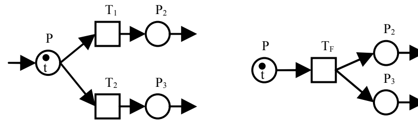

Figure 5: Implicit and explicit fork transitions

The behavior of such an implicit fork transition with `CHANNELSIZE=1` can be described for a pair of transitions T_1, T_2 with the same input place P by the following LTL formula: $\Diamond p$, with $p\equiv$true, iff T_1 and T_2 fires in the same event evaluation cycle. In a PROMELA model with more than one message per channel (`CHANNELSIZE>1`) the processes corresponding to the EDL transitions must be extended, so that each message from the input channels which fulfill all process conditions has to be copied to a further auxiliary channel. If this auxiliary channel contains a message twice after the termination of all processes then the signature possesses a *twice triggering token event combination*. The auxiliary channels must be erased before a new event occurs.

Non-completion of signature instances: A token on a place which cannot be transferred either to an exit or escape place denotes a *non-completion signature instance*. This corresponds to an attack instance whose abortion or successful completion is not recognizable. Accordingly, we check, whether the PROMELA model of a signature contains messages which cannot be transferred to an exit place or to escape place channels. This requires a modification of the PROMELA model in such a way that messages reaching the exit or escape place channels are deleted immediately.

A situation fulfills the *non-completion signature instances* property, when the PROMELA model reaches a state, where the initial state (only initial place channels contain a message) cannot be reached again. In this case, the PROMELA model is referred to be non reversible. Here reversibility is defined by the LTL formula: $\Box \neg q \vee \Diamond(q \wedge \Diamond p)$ with (1) $p \equiv$true, if all channels representing an initial place contain a single message and all remaining channels are empty, and (2) $q \equiv$true, if an arbitrary channel not representing an initial place contains a message.

The search for *non-completion* instances can be refined if the transfer of all messages to *escape* channels as well as the transfer of all messages to *exit* places are examined separately. The transfer of a signature instance (token) to *exit* and *escape* places always has to be possible because an attack can be aborted in each system state or continued until success. In order to check this, all *exit* place channels and all processes representing incoming transitions have to be removed, whereas for verifying the transfer to *exit* place channels, the escape place channels and processes representing incoming transitions have to be removed. Both cases can be verified with the aforementioned LTL formula.

Note that all described LTL properties can be adapted in such a way that an unambiguous identification of a faulty transition or problematic place/transition structures in the EDL signature is possible. For the sake of briefness, we cannot describe this in detail here.

5 Example

In order to prove the suitability of our verification approach we implemented the transformation rules introduced in Section 3 in a prototype transformer. This transformer reads an arbitrary EDL signature and generates the semantically equal PROMELA model. Besides, the transformer determines the equivalence classes for the occurring events by splitting the complex transition conditions into atomic conditions. After that, it generates the representatives for each equivalence class. For this, we use the finite domain solver [9] from the GNU PROLOG package [10]. We handle equal comparisons of two event features f_1, f_2 in an equivalence class by replacing each usage of f_2 (resp. f_1) with f_1 (resp. f_2). All other conditions are mapped to constraints between the features of the representative. Thereby the transformer automatically recognizes classes with mutually exclusive conditions. Merely the EDL regular string comparison condition must be handled manually. After determining representatives the prototype generates the LTL formulas of the signature properties to be hold whereby the properties described in Section 4.2 were adapted to concrete channel and process names. Finally SPIN is automatically started with the generated model and the LTL conditions to be verified.

In the following we give an example of our verification approach using a typical signature for detecting a shell-link-attack on a Unix system. The **shell-link-attack** exploits a special shell feature and the SUID (*Set-User-ID*) mechanism. If a link to a shell script is created and the link name starts with "-" then it is possible to create an interactive shell by calling the link. In old shell versions regular users could create an appropriate link which points to a SUID-shell-script and produce an interactive shell which runs with the privileges of the shell-script owner (maybe the root user). Figure 6 depicts the respective EDL signature consisting of 15 transitions with 3-6 conditions per transition. The full textual specification of the signature consists of 356 lines.

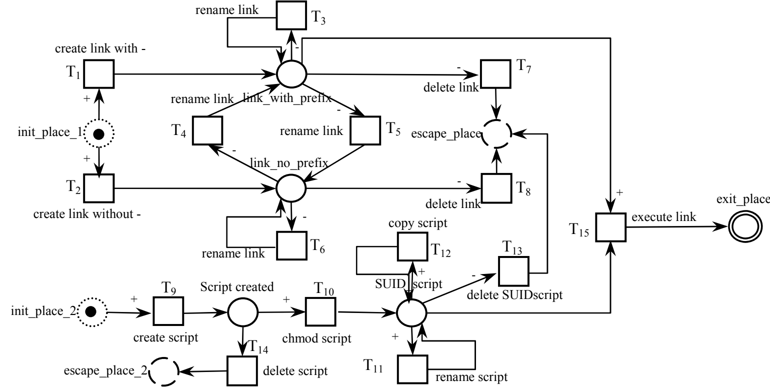

Figure 6: Simplified EDL-signature of the shell-link-attack

Our transformer identified 11 different atomic conditions for the shell-link-signature. Some of these conditions are mutually exclusive. 1920 representatives were generated for the equivalence classes. Further our tool automatically adapts the signature properties from Section 4.2 to the shell-link-attack signature and generated a set of LTL formulas that the signature should hold (see Table 3).

<table>
<tr><th>Signature property</th><th colspan="2">LTL formula in SPIN notation</th></tr>
<tr><td>Tracking new attack instances</td><td colspan="2"><> p -> (a U p);
a=(len($c_{init_place_1}$)>0 && len($c_{init_place_2}$)>0);
p=(isRunning(T1)||isRunning(T2)||...||isRunning(T14))</td></tr>
<tr><td>Unreachable system states:</td><td colspan="2"><>t_{CP1} && <>t_{CP2} && ... && <>t_{CPn}
t_{CP1} = (len($c_{init_place_1}$)>0);
t_{CP2} = (len($c_{init_place_2}$)>0);
t_{CP3} = (len($c_{link_with_prefix}$)>0);
...
t_{CPn} = (len(c_{exit_place})>0);</td></tr>
<tr><td>Dead system state changes:</td><td colspan="2">verified by unreached code</td></tr>
<tr><td>Twice triggering token event combinations</td><td colspan="2"><> p;
p=((wasTriggered(T1)&& wasTriggered (T2)) ||
(wasTriggered(T3)&& wasTriggered (T7)) ||
(wasTriggered(T3)&& wasTriggered (T5)) ||
(wasTriggered(T7)&& wasTriggered (T5)) ||
...
(wasTriggered(T15)&& wasTriggered (T11)) ||
(wasTriggered(T15)&& wasTriggered (T12)) ||
(wasTriggered(T15)&& wasTriggered (T13)) ||
(wasTriggered(T15)&& wasTriggered (T3)) ||
(wasTriggered(T15)&& wasTriggered (T5)))</td></tr>
<tr><td rowspan="2">Non-completion of signature instances</td><td colspan="2">[] !q || <>(q && <>p);</td></tr>
<tr><td>p=((len($c_{init_place_1}$)==1) &&
(len($c_{init_place_1}$)==1) &&
...
(len($c_{link_with_prefix}$)==0)&&
(len(c_{exit_place})==0))</td><td>q=((len($c_{script_created}$)>0)||
...
(len($c_{link_with_prefix}$)>0)||
(len(c_{exit_place})>0))</td></tr>
</table>

Table 3: LTL-Formulas to verifying the shell-link signature

The verification ensures that all properties are fulfilled by the shell-link signature beside the "*non complete-able signature instances*". This property does not hold for the place "*SUID_script*" in Figure 6. This place models a system state where the attacker has created a SUID-script. In the PROMELA model there are messages in the corresponding channel that cannot be transferred to an escape channel. Consequently, the signature does not model each possible case if the attacker cancels the attack after script generation (T_9) and script

mode change (T_{10}). This can be done, for instance, by deleting the created SUID script (T_{13}). A closer look on the transition T_{13} reveals that the transition does not distinguish how the script mode was changed to a SUID_script, either by a `chmod` syscall or by an `fchmod` syscall on transition T_{10}. In the first case, T_{13} must identify related tokens for each occurring deletion event by comparing filenames, in the second case by comparing file descriptors. But the condition for distinguish the two cases and the condition for the second case is missing in T_{13}, therefore delete events which base on file descriptors are not correctly handled by T_{13}. This issue is depicted in Figure 7. Here the relevant section around transition T_{13} of the shell-link-attack signature is shown. The transition T_{10} sets the feature *"FileDescIsSet"* on place *"SUID_script"* to false and feature *"FileName"* with the observed file name, if the T_{10} was triggered by a event for a `chmod` syscall. But if T_{10} is triggered by a `fchmod` syscall, then *"FileDescIsSet"* is set to true and the logged file descriptor is mapped to *"FileDescriptor"*. The problem ist, that the second condition on transition T_{13} only correlates the feature "*FileName*" of place *"SUID_script"* with the feature "Efilename" of the occurring event *F*. But, the case of matching file descriptors is not considered. To correct this error the signature developer has to add the distinction of the two cases and the missing equal condition for file descriptors as shown in Figure 7 in section "Conditions of T_{13} after correction".

Such errors are typical specification errors done by signature programmers. Further errors like e.g. mutually exclusive conditions, wrong transitions mappings, missing cases, or unreachable places can be also very well detected by our verification approach.

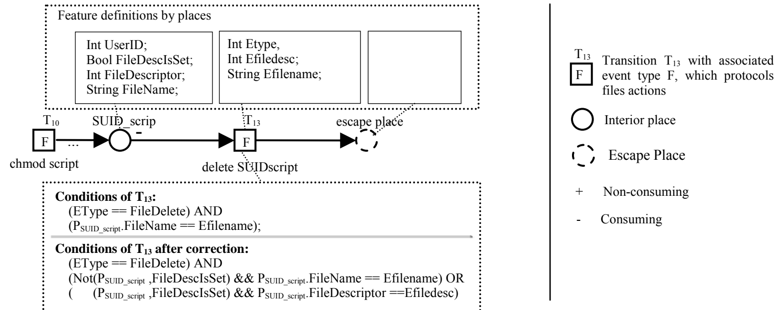

Figure 7: Detailed section of the shell-link-attack signature

Resource Requirements: In order to estimate the run-time and memory requirements of the SPIN tool we captured some performance figures. The following data refer to the verification of the generated PROMELA model of the shell-link-attack signature above. SPIN generated the complete state space for the PROMELA model on an AMD X2-64 (2 GHz) in 15 minutes and required nearly 900 MB for this. We used the SPIN options *partial order reduction*, *bit state hashing*, and *state vector compression*. In this configuration the complete state space contained 476,851 states with $2.2e^{+08}$ state changes. Our tool which performs the transformation from an EDL signature to the corresponding PROMELA model and the generation of the representatives of event classes required 25 seconds for the most complex signature.

Apart from these run-time characteristics, a further advantage of our approach is that unfulfilled LTL formulas, i.e. violated signature properties, can easily mapped onto concrete signature elements. Thus, fault detection and correction can be carried out easily.

6 Final Remarks

The derivation of signatures from new exploits is still a tedious process which requires much experience. Systematic approaches are still rare. Newly derived signatures often possess a significant detection inaccuracy which strongly limits the detection power of misuse detection systems as well as their acceptance in practice. A longer validation and correction phase is needed to derive qualitative and accurate signatures. This implicates a larger vulnerability window of the affected systems which is unacceptable from the security point of view. Verification methods can help to accelerate the signature development process and to reduce the vulnerability window.

In this paper we presented the first approach for identifying specification errors in signatures by verification. We have applied the SPIN model checker to detect common signature specification errors. The approach was implemented as tool for a concrete representative of a multi-step signature language, namely EDL. The tool maps a given EDL signature onto the corresponding PROMELA model and generates the signature properties which are then checked with the SPIN model checker. In addition, we developed an automated method for deriving a finite set of representative events, required for the verification. We have demonstrated and evaluated the approach exemplarily.

We are currently working on the identification of further properties which each signature should hold. Furthermore, we intend to include in our approach a feature that suggests possible solutions to the signature modeler to correct founded specification errors. As another working direction is the verification of single-step signatures as used in intrusion detection systems like *Snort*.

References

[1] Meier M.: A Model for the Semantics of Attack Signatures in Misuse Detection Systems. In: Proc. of 7th Information Security Conference (ISC 2004), Palo Alto, CA, USA, LNCS 3225, Springer, pp. 158–169, 2004.

[2] Meier M.; Schmerl S.: Improving the Efficiency of Misuse Detection. In: Proc. of the 2nd Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, Vienna, Austria, LNCS 3548, Springer, pp. 188-205, 2005.

[3] Vigna G., Eckmann S.T., Kemmerer R.A.: The STAT Tool Suite. In: Proc. of DARPA Information Survivability Conference and Exposition (DISCEX) 2000, Vol. 2, pp. 46-55, IEEE Press, Hilton Head, 2000.

[4] Schmerl, S.; König H.: Towards Systematic Signature Testing. In: Petrenko A., Veanes M., Tretmans J., Grieskamp W. (Eds.): Testing of Software and Communicating Systems, Proceedings of the 19th IFIP TC6/WG6.1 International Conference TestCom 2007, Tallinn, Estonia, June 2007, LNCS 4581, Springer, pp. 276-291, 2007.

[5] Eckmann S.T., Vigna G., Kemmerer R.A.: STATL: An Attack Language for State-based Intrusion Detection. Journal of Computer Security 10 (2002) 1/2: 71-104.

[6] Paxson V.: Bro - A System for Detecting Network Intruders in Real-Time. Computer Networks 31(1999) 23-24, pp. 2435-2463.

[7] Kumar S.: Classification and Detection of Computer Intrusions. PhD Thesis, Dept. of Computer Science, Purdue University, West Lafayette, IN, USA, August 1995.

[8] Ranum, M. J.: Challenges for the Future of Intrusion Detection. Invited Talk, 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), Zürich, 2002.

[9] Finite domain solver: http://www.gprolog.org/manual/html_node/index.html

[10] GNU PROLOG: http://www.gprolog.org/manual/html_node/index.html
[11] Holzmann J. G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003.