Simplifying Signature Engineering by Reuse

Sebastian Schmerl[1], Hartmut Koenig[1], Ulrich Flegel[2], and Michael Meier[2]

[1] Brandenburg University of Technology Cottbus
03013 Cottbus, Germany
sbs@informatik.tu-cottbus.de
koenig@informatik.tu-cottbus.de
[2] University of Dortmund
44221 Dortmund, Germany
ulrich.flegel@udo.edu
michael.meier@udo.edu

Abstract. Most intrusion detection systems deployed today apply misuse detection as detection procedure. Misuse detection compares the recorded audit data with predefined patterns, i.e. signatures. A signature is usually empirically developed based on experience and expert knowledge. Methods for a systematic development are scarcely reported yet. Automated approaches to reusing design and modeling decisions of available signatures also do not exist. This induces relatively long development times for signatures causing inappropriate vulnerability windows. In this paper we present an approach for systematic signature derivation. It is based on the reuse of existing signatures to exploit similarities with existing attacks for deriving a new signature. The approach is based on an iterative abstraction of signatures. Based on a weighted abstraction tree it selects those signatures or signature fragments, which are similar to the novel attack. Finally, we present a practical application of the approach using the signature description language EDL.

Keywords: Computer Security, Intrusion Detection, Misuse Detection, Attack Signatures.

1 Motivation

The growing dependencies of social processes on IT infrastructures as well as their increasing complexity provide a large potential of threats that jeopardizes these processes. To counter these threats intrusion detection systems (IDS) possess a prime importance as reactive measures. They provide means to automatically detect occurred security violations and to trigger appropriate countermeasures. IDSs apply two complementary approaches: 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. This is often difficult to provide so that anomaly detection has currently only a limited practical importance. Misuse detection focuses on the detection of attacks in audit trails described by patterns of known security violations, i.e. so-called signatures. The effectiveness of misuse detection strongly depends on the conciseness and the timeliness of the applied signatures. Imprecise signatures

G. Müller (Ed.): ETRICS 2006, LNCS 3995, pp. 436 – 450, 2006.
© Springer-Verlag Berlin Heidelberg 2006

heavily confine the detection capability of the intrusion detection systems and lead to false positives. The reasons of this detection inaccuracy can only be in part imputed to qualitative restrictions of the audit functions. Rather they must be sought in the signature derivation process itself. In particular, the derivation of signatures starting from given exploits often appears as weak point. An attack represents a sequence of actions that exploits a vulnerability in a program, operating system, or network. The derivation of a signature to detect the attack is mostly based on experience and expert knowledge. Methods for a systematic derivation have scarcely been reported yet. Automated approaches to reusing design and modeling decisions of available signatures also do not exist. This results in relatively long development times for signatures, causing an inappropriate window of vulnerability [7].

The development time of signatures could be shortened and their conciseness improved, if - analogously to software technology - methods for the reuse of design and implementation decisions of available signatures are applied. Only a few approaches have been published which deal with this subject. Cheung et al. propose to simplify the signature design by applying attack models [1]. This approach corresponds to the design patterns of software engineering [2]. It allows the reuse of architectural design decisions. The reuse of specified signatures or signature fragments is not supported. Rubin et al. describe how mutants can be generated for a given attack [3]. Mutants exploit the same vulnerabilities as the basic attack without, however, performing the same security relevant actions. If a signature for an attack mutant is supposed to be developed, the signature of the basic attack could be reused, if available. The approach of Rubin et al. could be reused for this purpose by deriving an abstracted attack. The required transformations though (except simple transformations like IP fragmentations) strongly depend on the specific attack. A universally valid procedure for all kinds of attacks is not implementable with this approach. Rubin et al. further describe a refinement of signatures based on formal languages [4]. This approach may help the signature developer to remove triggers for false positives caused by imprecise signatures. The procedure, however, assumes an almost error-free reference signature. Larson et al. present a tool for extracting the significant events of an attack from the audit trail [8]. It executes the attack and records the respective audit data. Then the differences between this audit data and attack free audit data are determined. The problem of deriving a signature from the differences, however, remains unsolved.

In this paper we present an approach for systematic derivation of signatures from given exploits and signatures. It is based on the reuse of existing signatures or signature fragments. The approach can be automated. It selects signatures from a set of existing signatures that are similar to the new attack. These signatures help the signature developer to orient itself and, if possible, to reuse former design decisions. The paper is organized as follows. In Section 2 we describe the general principle of the approach. Next in Section 3 we adapt the procedure to a concrete signature description language. Section 4 describes the practical application of the approach. The final remarks contain some conclusions and give an outlook on future research.

2 Principle of the Approach

The development of a signature for a novel attack can be divided in the following steps: (1) execution of the attack on a dedicated system to record its traces in an audit

trail. The traces are security relevant events (basic events) which are observed by sensors. (2) The signature developer investigates the basic events, identifies the relevant ones, and (3) step-by-step derives the new signature w.r.t. the approach of the attack to exploit the vulnerability of the attacked system. An important aspect in this context is the detection of patterns allowing the intrusion detection system to find attack traces. (4) After specifying the signature it must be validated to prove its correctness and conciseness. If needed, corrections or changes have to be introduced. The signature development process is time-consuming, in particular for phases (3) and (4). If knowledge of former signature designs may be reused, the time exposure for the signature development could be significantly reduced and their quality could be improved. Therefore the reuse of signatures is of great importance. Beside savings during design, the reuse of approved, i.e. validated, signature fragments may also considerably shorten the expensive validation phase.

In the following, we show how signatures can be automatically selected from a set of existing signatures which are qualified to be reused for a novel attack. This is based on the assumption that the signatures of related attacks are alike. After selecting the relevant signatures w.r.t. the new attack the signature developer can look for similarities and adapt the selected signatures. The identification of relevant signatures can be accomplished by an iterative abstraction of existing signatures until traces of the novel attack are covered. Abstraction means a generalization of signatures. Whether an abstracted signature detects the traces of the new attack can be easily decided by matching the signature to the traces. Abstractions are accomplished by iteratively applying transformations to the basic signatures. The abstraction procedure results in an abstraction tree. Each kind of transformation is weighted by a metric which defines a similarity measure related to the original signature. In order to identify reusable signature fragments the signature developer should focus on the least abstracted signatures. In the following sections we discuss transformations, abstraction trees and similarity measures.

2.1 Signature Transformations

A signature defines the set of identifiable manifestations of an attack. An attack manifestation is characterized by the events occurring during attack execution. The events form traces of the attack in the audit trail. In order to develop new signatures based on the reuse of existing ones, signatures that detect similar attack manifestations need to be identified. Although signatures may be similar, each signature is specialized to detect a specific attack. Signature transformations strongly depend on the signature model used in the applied signature description language. To identify reusable signatures it is necessary to abstract from the attack-specific elements. This is done by iteratively removing the semantic features w.r.t. the applied signature description language. The semantic features determine the set of attack manifestations a signature can detect. New signature abstractions can be obtained by removing restricting features from a signature or by weakening them. Transformations enlarge the set of attack manifestations M_{AS} that can be detected by an abstracted signature *AS*. A transformation *T* only abstracts a signature *S* to *AS* if $M_S \subseteq M_{AS}$ holds for the related sets of manifestation M_A and M_{AS}.

We apply the following rules for selecting appropriate transformations: (a) No transformations should be defined which produce the same abstraction. This rule prevents redundant transformations. (b) Transformations should always be selected such that they semantically weaken the signature only slightly. Such transformations lead to fine-granular abstractions of signatures. (c) A transformation generating an abstraction AS from a signature S should preferably meet the condition that the respective manifestation sets M_S and M_{AS} are disjoint.

2.2 Signature Abstraction Tree

There exists a relation between the derived signatures concerning their abstraction. Two signatures A and B are related, if B is abstracted from signature A. The relations between a basic signature S and all its abstractions can be represented by a signature abstraction tree in which S represents the root node. Direct children of S are those abstractions that can be generated by applying a single transformation to S. The signature abstraction tree and the related abstracted signatures can be generated successively. The tree structure determines the abstraction degree of S.

An abstracted signature associated to a node of the abstraction tree may detect all signature manifestations of the signature of the parent node, i.e. an abstraction tree defines a *contained in* relation over all sets of attack manifestations. Figure 1 depicts an example abstraction tree of S. Abstraction AS_1 is, for instance, directly generated from S by transformation 1, whilst signature AS_7 is derived from AS_1 using transformation 2. Their manifestation sets M_S, M_{AS1}, M_{AS7} must fulfill the condition $M_S \subseteq M_{AS1} \subseteq M_{AS7}$.

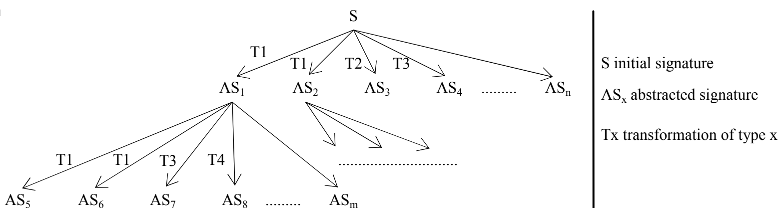

Fig. 1. Example of a signature abstraction tree

The signature abstraction tree may contain identical nodes. Two nodes are identical if the associated signatures are identical. The sub trees of these nodes are equal. Therefore one of the identical nodes and its sub tree can be removed.

After deriving the abstraction tree the abstracted signatures have to be tested to prove to what extend they identify the traces T of the new attack. These tests can be supported by the use of an intrusion detection system. The test applies a breadth-first search. It stops, when an abstracted signature is found which detects the traces T and if there is no other untested signature which has a larger similarity to the root signature. In practice, the generation and evaluation of abstracted signatures should be performed in an interleaved manner. This ensures that only the signatures required for the test are generated and that in case of identical nodes the test is limited to the respective tree fragment.

2.3 Similarity Measures of Signatures

The similarity of an abstracted signature compared to the basic signature decreases with each applied transformation. In order to estimate the similarities of different abstracted signatures the edges of the abstraction tree are weighted with a metric δ. This metric maps the type of the transformation onto real numbers. If a signature AS is generated from signature S by transformation X then the related edge between nodes P and P' is weighted with the metric value of the transformation X. The metric rates the degree of the semantic abstraction of the transformation. After weighting the edges of the tree the similarities of different abstracted signatures related to the basic signature S can be assessed. This is done by cumulating the edge weights on the path from the abstracted signature to the root node, i.e. the abstraction degree of a signature is determined by the sum of the edge weights.

2.4 Selection Procedure

Now we describe how signatures are selected from a set K of known signatures for a new attack. First the traces T of the new attack, which are received by executing the exploit on a system, are logged. Next the abstraction degree of each signature S in K is determined. It summarizes the abstractions which have to be applied to S to recognize the traces T. The procedure comprises five steps: (1) Successive application of transformations to S to derive all possible signature abstractions. (2) Generation of the abstraction tree. (3) Weighting the edges in the tree using a metric δ. (4) Applying all abstracted signatures of S to the traces T by using an IDS and indicating all signatures which identify T. (5) Selecting the abstracted signature with the smallest edge weight to root S from this subset. This abstraction degree is assigned to the signature S. After accomplishing this procedure for all signatures S in K, the abstraction degree of each signature is given. The signatures in K with the lowest abstraction degrees are suggested to the signature developer for reuse. The selection steps may be optimized and executed in parallel as indicated in Section 2.2.

3 Application to EDL

Signatures are specified using various languages. Therefore the selection procedure has to be adapted to the given signature description language or semantic model. We now demonstrate this for EDL (*Event Description Language*) [5], [6]. EDL is a signature description language which is based on a Petri-net like modeling approach. It supports the specification of complex multi-step attacks and possesses a high expressiveness and nevertheless allows for efficient analysis (cf. [6]). Before we describe the possible transformations we outline the essential features of EDL. A detailed description of EDL can be found in [5].

3.1 Modeling Signatures with EDL

The descriptions of signatures in EDL consist of places and transitions which are connected by directed edges. *Places* represent states of the system which are traversed by the related attack. *Transitions* represent the state changes. They describe the specific

events which cause the state change, e.g. security relevant actions. These events are contained in the audit data stream recorded during the attack. The signature execution is represented by tokens which flow from state to state. *Tokens* represent concrete signature instances. They can be labeled with values as in colored Petri-nets.

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 of the tokens which are located in a place. 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. They are marked with an initial token at the start of analysis. Each signature has exactly one *exit place* which describes the final place of signature. If a token reaches this place, then the signature has identified a manifestation of an attack in the audit data stream. *Escape places* indicate an analysis stop of an attack instance. They are reached if events occur which make the completion of the attack instance impossible. Tokens which reach these places are discarded. All other places are *interior places*. Figure 2 shows a simple signature with places P_1 to P_4 for illustration.

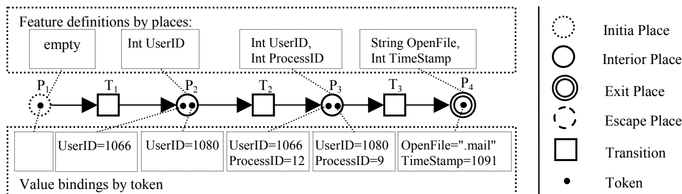

Fig. 2. Features and places

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 system states requires a security relevant event. Therefore each transition is associated with an event type. Further, a system change can require additional conditions which specify that certain features of the event (e.g. user name) are assigned with particular values (e.g. root). Conditions can require distinct relationships between event and token features on input places (e.g. same values).

If a transition fires, tokens are created on the transition's output places. These tokens 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. [5]) 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. Only in the consuming case the tokens which activate the transition are deleted on the input places.

Figure 3 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*.

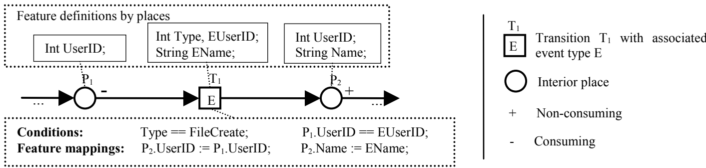

Fig. 3. Transition properties

3.2 Signature Transformations for EDL

In the following we present seven transformations for abstracting EDL signatures. We distinguish between transformations that retain the structure of the input signature (transformations 1 to 3), i.e. which modify only the properties of the transitions, and transformations that change the signature structure (transformations 4 to 7), i.e. which modify the topology of the places, transitions and their connecting edges. For each transformation, we describe the objective, the procedure, and the pre-conditions. The transformations abstract a given signature *S* with a given set of manifestations M_S, if and only if for the resulting signature *AS* and the corresponding set of manifestations M_{AS} holds $M_S \subseteq M_{AS}$. Note that this is not necessarily the case for arbitrary transformations. We provide appropriate pre-conditions for transformations that ensure that the transformations indeed abstract the signatures and the resulting abstract signatures are syntactically correct w.r.t. the modeling framework. If a transformation *Tx* may violate its pre-condition but not when applying another transformation *Ty*, *y*!=*x*, then *Tx* may only be used after *Ty*. Such dependencies are indicated where appropriate.

Transformation 1 (*broadening event correlation*): The consumption property of a given transition determines which events in a given trace should be correlated. The consuming mode restricts the set of events that are considered for correlation (see T_1 in Figure 3). The consumption of a token removes collected information about observed attack steps. Therefore, by transforming a consuming edge into a non-consuming one we broaden the set of events considered for correlation. As result, the number of manifestations detected by the signature may increase.

Procedure: If there is a consuming edge from an input place to a transition, the edge is transformed into a non-consuming edge.

Pre-conditions: The transition connected to the edge must not have an escape output place. Otherwise the effect of the transformation equals the effect of transformation 7.

Remarks: Transforming a consuming edge into a non-consuming one may significantly increase the number of tokens to be considered simultaneously. This negatively affects the performance of the IDS.

Transformation 2 *(relaxing static conditions)*: A given transition can restrict the events that may activate the transition by constraining the event features to constant values (*static conditions*), e.g. the first condition of T_1 in Figure 3 (*Type==FileCreate*) restricts the set of events that may activate T_1 to events where the feature *Type* is valued *FileCreate*.

Procedure: If there is a transition with a static condition, remove the static condition of the transition.

Pre-condition: The modified transition must not have consuming incoming edges. Otherwise tokens may be consumed and evolved, which may never reach an exit place, due to conditions of subsequent transitions. Consequently the number of detected manifestations is effectively reduced. Transformation 1 may be used to ensure the pre-condition.

Remarks: Instead of completely removing a condition, we could (a) remove only subterms of the condition or (b) relax restrictive test operations. We believe that such a refinement does not result in a relevant degree of abstraction.

Transformation 3 (*relaxing dynamic conditions*): A given transition can restrict the events that may activate the transition by constraining the event features to values binded to input tokens (*dynamic conditions*), e.g. the second condition of T_1 in Figure 3 (P_1*.UserID==EUserID*). Evaluating dynamic conditions means correlating token features and event features, i.e. restricting the set of events that may activate the transition by enforcing relations between these events and events that have previously fired some transition(s). Such restrictions can be revoked by removing dynamic conditions, resulting in an increased number of events that may activate the corresponding transition.

Procedure: If there is a transition with a dynamic condition, remove the dynamic condition of the transition.

Pre-condition: See the pre-condition of transformation 2.

Remarks: By removing dynamic conditions the corresponding feature definitions of the input places and the token bindings of preceding transitions may become obsolete and should be removed. Moreover, we could relax dynamic conditions in a finer-grained way, as described for static conditions for transformation 2.

Transformation 4 (*removing pre-conditions*): The topology of places and transitions implies a temporal constraint on events for transition activation. If we intend to modify temporal pre-conditions, we can modify signature elements that are connected to initial places, ignoring the event expected first in the temporal order. More specifically, if we intend to ignore temporal pre-conditions of the signature, we can remove transitions connected to initial places.

Procedure: If there exists a transition *T*, where all connected input places are initial places, remove *T* as well as all of its input places that are not connected to other transitions. The output places of *T* are transformed into initial places of the resulting signature, if not already removed due to a loop (input place == output place). Applying transformation 4 to transition T_1 of the example signature in Figure 4 results in the signature depicted in Figure 5.

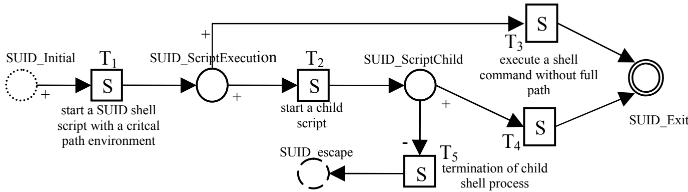

Fig. 4. Input signature to be transformed

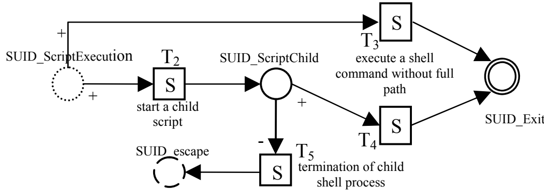

Fig. 5. Abstracted signature by means of transformation 4

Pre-condition: Transformation 4 may only be applied, if the following conditions are met. (1) The resulting signature would contain at least one path from some initial place to some exit place. Note that applying transformation 4 may imply to transform interior places with feature bindings into initial places without feature bindings. Thus, (2) we may only remove transitions, where the output places do not bind features to tokens that are referenced by subsequent transitions. Note that removing feature bindings affects the transitive closure of feature bindings. Moreover, (3) the newly transformed initial places must not be connected to consuming edges. Transformations 1 and 3 may be applied to ensure the aforementioned conditions.

Transformation 5 (*removing post-conditions*): If we intend to ignore temporal post-conditions of a signature, we can remove transitions connected to exit places, ignoring the event expected last in the temporal order.

Procedure: If there exists a transition *T*, where all connected output places are exit places, remove *T*, as well as all of its output places that are not connected to other transitions. The input places of *T* are transformed into exit places of the resulting signature. Applying transformation 5 to transition T_4 of the example signature in Figure 4 results in the signature depicted in Figure 6.

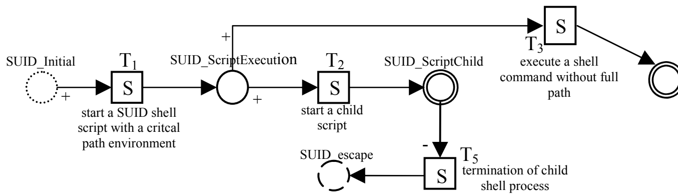

Fig. 6. Abstracted signature by means of transformation 5

Pre-condition: As for transformation 4, the resulting signature would have to contain at least one path from some initial place to some exit place.

Remarks: As a result of transformation 5 there may be more than one exit place. This is okay as long as none of the hierarchical concepts of EDL are used. The abstracted signature may contain transitions that have exit places as input places, e.g. T_5 in Figure 6. Since these transitions will never be activated, this is semantically correct. However, in order to avoid confusion of the signature engineer, such transitions should be pruned. When removing a transition, the feature definitions of preceding places and token bindings of preceding transitions may become obsolete and should be removed.

Transformation 6 (*removing intermediary conditions*): If we intend to ignore the intermediary temporal conditions of a signature, we can remove transitions not connected to initial and exit places, ignoring an event expected to occur after the first and before the last event in the temporal order.

Procedure: If there is a transition *T* that is connected to interior places only, then remove *T* as well as all of its output places. The former input places of *T* are transformed into input places of the transitions connected to the former output places of *T*. Applying transformation 6 to transition T_2 of the example signature in Figure 4 results in the signature depicted in Figure 7.

Pre-conditions: The transition selected for removal must not bind features to tokens that are referenced by subsequent transition conditions. As for transformation 4 the transitive closure of feature bindings needs to be considered for this criterion. Transformation 3 may be used to ensure the pre-condition.

Remarks: As for transformation 5, spurious feature definitions of places and feature bindings of transitions should be removed.

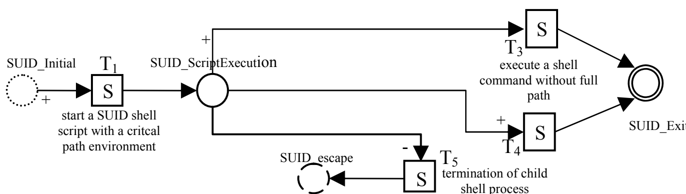

Fig. 7. Abstracted signature by means of transformation 6

Transformation 7 (*removing escape conditions*): Ignoring escape conditions may abstract a signature, because the removal of such a condition may result in further consideration of tokens that otherwise would have been removed. Escape conditions are modeled by transitions where the output place is an escape place.

Procedure: If there is a transition *T* that is connected to an escape place, then remove *T* as well as its output place, unless it is not connected to further transitions. Applying transformation 7 to transition T_5 of the example signature in Figure 4 results in the signature depicted in Figure 8.

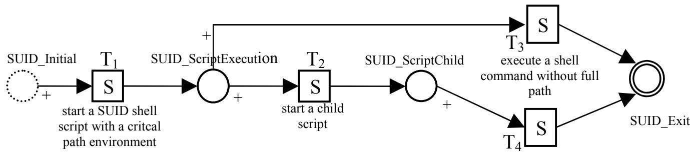

Fig. 8. Abstracted signature by means of transformation 7

Pre-conditions: None.

Remarks: As with transformation 1, the number of tokens to be considered for the abstract signature may significantly increase, thereby impairing the performance of the IDS. Moreover, as for transformation 5, spurious feature definitions of places and feature bindings of transitions should be removed.

The given transformations meet the rules (a) and (b) from Section 2.1. For transformations 1, 2, 3 and 7, rule (c) ($M_S \subset M_{AS}$) is only met, if additional signature instances in existence due to the transformation can be correlated such that the corresponding tokens reach an exit place. If this is impossible due to certain transition constraints, the transformations result in an increased number of tokens, but the number of detected manifestations is not increased.

The proposed transformations cover the whole range of syntactical elements of the signature language EDL. The transformations allow removing or relaxing arbitrary elements provided by EDL for restricting the flow of tokens from an initial to an exit place, effectively restricting the number of signature instances. Transformations are restricted to removing or relaxing constraining elements, thereby generalizing the original signature. We ignore the possibility of aggregating and (de-)composing signatures here, because - in contrast to more general signatures - we consider the resulting signature as semantically different from the original signature. Our approach is based on the idea to suggest the original signature to the signature engineer for further consideration in combination with its abstracted signatures which detect the given manifestation. We believe it would be rather irritating for the engineer, if the original signature detects something semantically different than the abstracted signature.

3.3 Complexity

We demonstrate that our approach can be used under real-world resource limitations by determining the space requirements for the abstraction tree (see Section 2.2) w.r.t.

the EDL signature transformations given in Section 3.2. Since the abstraction tree is constructed during run-time, its size also provides an intuitive measure for the computational complexity of our approach.

Suppose that for a given signature *S* the pre-conditions for all transformations are met. The number of nodes in the abstraction tree for a given signature *S* is the number of signatures that can be abstracted from *S*. The number of abstract signatures for *S* depends on the following parameters: *N* the set of transitions with no escape output places, *E* the set of transitions with escape output places, where $N \cap E = \emptyset$, *b* the number of conditions used by the transitions in *N*, and *c* the number of transitions in *N* that are connected to consuming edges.

Given the transformations from Section 3.2 $m=c+b+|E|+|N|$ elements of *S* can be transformed. Along a path from the root node of the abstraction tree of *S*, in each level a transformable element is removed. Hence, the tree depth is at most *m*. Moreover, in level *k* of the abstraction tree we have chosen to transform *k* from *m* transformable elements of *S*. Thus, there are $h_k = \binom{m}{k} = \frac{m!}{(m-k)!*k!}$ nodes in level *k*, representing all distinct abstract signatures that can be derived from *S* by *k* transformations. Consequently, the number of nodes from the root node to level *k* is $g_k = \sum_{i=0}^{k} \binom{m}{i}$. Further, there are at most $g = 2^m$ combinations how the transformations can be applied to *S*. Thus, *g* is the lowest upper bound of the total number of nodes of the abstraction tree.

The further investigations consider the set of signatures that was described in more detail in [6]. Table 1 gives the parameters for the signatures with the smallest, the largest, and an average number *m* of transformable elements. The average signature is a virtual signature, where the average parameter values of all signatures are assumed. For these signatures Table 1 also gives the depth *m* of the abstraction tree and the number g_m of abstracted signatures. The signature with the largest *m* describes a pretty complex shell link attack which can be considered as a special case. Note that while g_m considers identical nodes, all pre-conditions of transformations were ignored. However, Table 1 demonstrates that deriving all abstract signatures is feasible for average signatures, but infeasible for complex signatures. This result is further illustrated in Table 2. Execution times s_k were measured for the IDS SAM [7] on a Pentium III 800 MHz w.r.t. an attack manifestation with 10 events. s_k is the aggregate execution time of SAM in seconds (if not noted otherwise) for matching g_k signatures against the given attack manifestation.

Table 1. Signature parameters and size of corresponding abstraction trees

Parameter		*min*	*max*	*average*
number of non-escape transitions	$\|N\|$	3	7	4,25
number of conditions	b	7	18	8
number of consuming edges	c	0	4	1,25
number of escape transitions	$\|E\|$	2	2	2,5
max. depth of tree	m	12	31	16
number of abstracted signatures	g_m	4.096	2.147.483.648	65.536

The results suggest that we can generate and test all abstract signatures for average signatures in a reasonable time. Complex signatures obviously pose a problem. We propose to reduce their complexity by testing their static conditions before applying transformations. Abstracted signatures can be excluded from generation and the test, if each path from an initial place to an exit place contains some static condition that cannot be met by any event in the given attack manifestation. Additionally, it is not necessary to generate and test all abstract signatures of a given set K of signatures. It is sufficient to select t signatures that had to be abstracted the least. Suppose we found abstract signatures for the first t signatures S_i, $i=1..t$ in K with minimum similarity measures of a_i. Then it is sufficient for signature S_{t+1} to generate and test abstract signatures only as long as their similarity measure is lower or equal $max(a_i)$.

Table 2. Execution time for abstraction tree testing

	h_k, g_k and s_k for testing k tree levels							
signature	***k=***	***2***	***4***	***6***	***8***	***10***	***12***	***m***
min	h_k	*66*	*495*	*924*	*495*	*66*	*1*	*1*
	g_k	*79*	*794*	*2.510*	*3.797*	*4.083*	*4.096*	*4.096*
	s_k	*0.1*	*1.2*	*3.8*	*5.7*	*6.1*	*6.2*	*6.2*
max	h_k	*465*	*31.465*	*736.281*	*7.888.725*	*44.352.165*	*141.120.525*	*1*
	g_k	*497*	*36.457*	*942.649*	*11.460.949*	*75.973.189*	*301.766.029*	*2.147.483.649*
	s_k	*0.7*	*55.2*	*23.8m*	*4.8h*	*31.9h*	*5.2d*	*37.6d*
average	h_k	*120*	*1.820*	*8.008*	*12.870*	*8.008*	*1.820*	*1*
	g_k	*137*	*2.517*	*14.893*	*39.203*	*58.651*	*64.839*	*65.536*
	s_k	*0.2*	*3.8*	*22.5*	*59.3*	*88.8*	*98.2*	*99.2*

4 Example

In order to prove the suitability of the approach, an analysis of different signature engineering cycles of several signature developments has to be performed. Thereby traditional development processes as well as processes incorporating the proposed procedure have to be considered. By comparing the relevant parameters of the processes, e.g. the development times and the quality of the developed signatures, the suitability of the approach can be evaluated. Such an evaluation requires large financial efforts as well as a lot of human resources. To get an impression of the suitability of the proposed procedure we applied the procedure several times exemplarily. Thereby we made consistently positive experiences. In the following we describe an application example. The modus operandi, the results as well as possible improvements are explained.

The set of known signatures contains (amongst others) the signatures of a *Suid-Script*-, a *Link*- and a *Failed-Login*-Attack which are described in [7]. As new attack we used a candidate that has substantial similarity to the *Suid-Script*-Attack. Both attacks exploit the environment variable path and the suid mechanism. By using the variable path, a user specifies a list of directories to be searched for executable files. As a precondition of the *Suid-Script*-Attack a directory (*Dir*) must exist, that can be written by the attacker. Further a *Suid-Root*-Script must exist that calls a command (*Cmd*) without using its complete path. In this case an attacker can place a script in directory *Dir* that is equally named with *Cmd*. When the *Suid-Root*-Script is executed, the attacker script is called and executed with the privileges of the root user. Figure 4

sketches the corresponding EDL signature. The mechanisms used by the new attack and the traces generated by its execution are much the same. But, instead of using a suid script a binary suid application is used. The execution of applications is logged by different events and values than the execution of scripts.

Using the audit trail documenting the new attack and the set of known signatures the proposed procedure is applied. We use a metric δ that associates each transformation with value 1. Table 3 summarizes the results. It shows the applied transformations, their frequency, the level of abstraction, as well as the number of places and transitions of the four least abstracted known signatures.

Table 3. Signature ranking

Signature name	*Level of abstraction (concerning Δ(x)=1)*	*Applied transformations*	*M= c+b+|E|+|N|*
Suid-Script	*1*	*1*Transform. 2*	*12*
JoinMailFile	*3*	*3*Transform. 2*	*12*
Link-Shell	*4*	*3*Transform. 2 + 1*Transform. 3*	*31*
Failed-Login	*6*	*6*Transform. 2*	*13*

The *Suid-Script*-Signature is suggested as signature for the new attack. Merely one condition of transition T_1 needs to be adapted. Even if the attack is modified in a way such that instead of a script an application that is equally named to *Cmd* is placed in *Dir*, the *Suid-Script*-Signature is selected from the set of known signatures and suggested as most suitable for reuse. In order to detect the modified attack, the conditions of the transitions T_2, T_3 and T_4 need to be adapted additionally. But the intrinsic signature characteristic, which realizes the tracking of child processes, persists.

5 Final Remarks

In this paper we have presented an approach to reusing patterns of existing signatures for the development of new signatures. The approach is geared to systematic development of signatures and exploits the fact that similar attacks produce similar traces, such that existing signatures may provide a substantial basis for developing new signatures. The reuse of approved structures may not only reduce the effort of the signature engineering process, but can also considerably shorten the costly test and correction phase. Moreover, the proposed procedure allows the signature engineer to revert to experience with existing signatures.

A signature base *K* typically contains a number of signatures that are specialized to a certain attack. The proposed approach selects the signatures of *K* that are most similar to the new attack by systematically relaxing the specializations of signatures in *K*. The procedure can be automated. For the selection process, the following preconditions have to be fulfilled: (1) The quality of the signatures in the set *K* is good. (2) The transformations must be chosen carefully and follow the rules mentioned in Section 2.1. (3) The metric used to measure signature similarities rates the semantic abstractions of the transformations appropriately.

There are several future research directions. In this paper we focus on suggesting reusable signatures for a new attack. An alternative way to support the signature engi-

neer is a catalogue of design patterns for signatures. So we intend to examine larger sets of signatures for recurring patterns to derive and generalize these to design patterns. We envision that signature design patterns provide the same advantages like design patterns in object oriented software (cf. [2]). Another direction is the automatic derivation of test scenarios from a signature, in order to improve the testing of signature correctness and conciseness.

References

[1] Cheung S.; Lindqvist U.; Fong M.: Modeling Multistep Cyber Attacks for Scenario Recognition. In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition, Washington, USA, IEEE Computer Society Press, pp. 284-292, 2003.

[2] Gamma E., Helm R., Johnson E. R., "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, 1997, ISBN: 0201633612.

[3] Rubin S., Jha S., Miller B.: Automatic Generation and Analysis of NIDS Attacks. In Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, AZ, USA, IEEE Computer Society Press, pp. 28-38, 2004.

[4] Rubin S.; Jha S.; Miller P. B.: Language-based generation and evaluation of NIDS signatures. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, IEEE Computer Society Press, pp. 3-17, 2005.

[5] Schmerl S.: Entwurf und Entwicklung einer effizienten Analyseeinheit für Intrusion-Detection-Systeme (in German). Master Thesis, Group Communication Systems, Brandenburg University of Technology Cottbus, 2004.

[6] 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, Lecture Notes in Computer Science, Vol. 3548, pp. 188-205, 2005.

[7] Lippmann, R.; Webster, S.; Stetson, D.: The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection. In Proceedings of the Symposium on Recent Advances in Intrusion Detection, Zurich, Switzerland, Lecture Notes in Computer Science, Vol. 2516, pp. 307-326, 2002.

[8] Larson U., Lundin Barse E., Jonsson E.: METAL - A Tool for Extracting Attack Manifestations. In Proceedings of the 2nd Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Vienna, Austria, Lecture Notes in Computer Science, Vol. 3548, pp. 85-102, 2005.