Efficiency Issues of Rete-based Expert Systems for Misuse Detection

Michael Meier, Ulrich Flegel
University of Dortmund
Department of Computer Science
D-44221 Dortmund, Germany
michael.meier, ulrich.flegel@udo.edu

Sebastian Schmerl
Brandenburg University of Technology Cottbus
Computer Science Department
D-03013 Cottbus, Germany
sbs@informatik.tu-cottbus.de

Abstract

This paper provides a general and comprehensive approach to implementing misuse detection on expert systems and an in-depth analysis of the effectiveness of the optimization strategies of the Rete algorithm wrt. the general implementation approach. General efficiency limits of Rete-based expert systems in the domain of misuse detection are determined analytically and validated experimentally. We conclude that expert systems may still have their merit in rapid prototyping of misuse detection IDSs, but they should not be considered for modern production systems.

1. Introduction

Within a quarter century of research the community has come up with a plethora of novel approaches to the problem of intrusion detection. In many areas of the field a diverse arsenal of new techniques and technologies has been developed, but it seems that one favorite standard approach to misuse detection are expert systems. Since the first IDSs using expert systems (Haystack [25], MIDAS [24]) there has been proposed in close succession of a few years yet another expert system IDS: IDES [16], AudES [30], NADIR [11], DIDS [26], Hyperview [6], NIDES [4], CMDS [23], SECURENET [12], AID [27], Emerald eXpert [15]. Even recently we continue to see proposals to employ expert systems or the Rete algorithm for misuse detection [5]: SNIDJ [1], Hybrid IDS [7].

We feel that it is time to clarify the limits of expert systems in the domain of misuse detection, such that we can move on and use existing state-of-the-art solutions that are more suitable to the domain.

The novel contribution of this paper is threefold: a general and comprehensive approach to implementing misuse detection on expert systems, based on a comprehensive semantics model; an in-depth analysis of the effectiveness of the optimization strategies of the Rete algorithm wrt. the general implementation approach; analytically determined and experimentally validated general efficiency limits of Rete-based expert systems in the application domain of misuse detection.

We summarize and define the concepts of misuse detection in Sect. 2 as well as of expert systems in Sect. 3 to be able to describe the implementation of misuse detection using expert systems. In Sect. 4 we analyze the limits of expert systems in the application domain of misuse detection with respect to efficiency. We conclude with a survey of related work in Sect. 5 and a summary in Sect. 6.

2 Misuse Detection

Attacks or *misuse scenarios* are defined as *activities* considered to violate the security policy of a given organization. We distinguish the type and the instance of an activity. An *activity type* is specified by a descriptive name plus a set of feature types. A *feature type* describes possible instances of a certain feature of an activity type using a pair of a descriptive name and a range/domain of values. A *feature instance* is a specific pair of a descriptive name and a value within the range of the corresponding feature type. Consequently, an *activity instance* is determined by a descriptive name and a set of feature instances.

The *audit component* of a given system observes the system activity. The *observation/manifestation of an activity type* is denoted as *event type*, which is also specified by a descriptive name and a set of feature types. Likewise, the *observation/manifestation of an activity instance* is an *event instance*, given by a descriptive name and a set of feature instances. The mapping between activity types and event types is specific for the given audit component. Note that the activity names mapped to event names are not necessarily the same, as well as the selection of feature sets and the feature names. The system-specific syntax of an event type is specified by the corresponding *audit record type*, and a given event instance is represented in the system-specific syntax by an *audit record instance*. We denote a partially

In P. Samarati; Ch. Payne (eds.): Proceedings of the 23rd International Annual Computer Security Applications Conference (ACSAC 2007), pp. 151-160, Miami Beach, Florida, USA, December 2007, IEEE Computer Society.

ordered set of audit record instances and event instances as an *audit (record instance) trail* and *event (instance) trail*, respectively.

Misuse detection then is the process of detecting manifestations of misuse scenarios in the audit trail based on models of manifestations of misuse scenarios (*signatures*). This is done under the assumptions that manifestations are actually observed, when a misuse scenario takes place (*completeness*), and that a misuse scenario actually takes place, when respective manifestations are observed (*correctness*). In the following, we assume that we have audit components that can observe misuse scenarios completely and correctly. This assumption simplifies the terms used for describing the concepts of misuse detection, because we do not need to strictly distinguish activities and observations/manifestations of activities. We can then talk about events, as if they are actual activities, instead of merely observations/manifestations of activities. Also, it is then possible to talk about models of misuse scenarios that describe (misuse) activities, if we actually mean models that describe misuse scenario manifestations.

2.1 Semantic Requirements of Misuse Detection

We briefly summarize the semantic requirements for modeling misuse scenarios for misuse detection [18]. In the following, we denote a model of a given misuse scenario as a complex event type. A *complex event type* consists of inter-related *step types*, where each step type specifies conditions in order to match an event type that is part of the corresponding manifestation of the misuse scenario. The conditions comprise constraints on event type names, intra-event feature instances, inter-event feature instances and the order of step types. Additionally a step type defines, which event instance is selected for the step instance, whether preceding step instances are consumed and what *action* is executed as a response to the detected activity.

For a *complex event instance* to occur, matching event instances must have occurred and must be *bound* to the step types required by the complex event type. The binding of an event instance to a step type, is denoted as the *step instance*, and the step instances are considered to be the (partial) *complex event instance*. If all required step types of the complex event type are instantiated, the (complete) complex event instance is said to occur.

Summarizing, a complex event type models an attack or misuse scenario by specifying how to identify the event instances that can be observed while the misuse scenario activities take place. Since several misuse scenarios of the same type may be executed simultaneously, we need to model manifestation-specific state (associate feature values with step instances), to be able to distinguish distinct (partial) instances of a given (type of) complex event.

The semantics of complex events can be partitioned in three dimensions [18]: *Event Pattern*, *Step Instance Selection*, and *Step Instance Consumption*.

2.2 Event Pattern

An event pattern defines the complex event type to look for. The frame of a complex event type is formed by the step types and their *order*. Many complex event types describe simply consecutive step types (*sequence*, e.g. *(A; B; C)*). Alternative step types can be modeled disjunctively, allowing to represent variants of misuse scenarios in a compact way (*disjunction*, e.g. *(A OR B OR C)*[1]). Concurrent threads of step types can be modeled in a conjunctive way, such that all interleavings of the corresponding event instance sequences are accepted (*conjunction*, e.g. *(A AND B AND C)*). Simultaneous step types may occur in parallel systems and can be correlated using the time stamps of the manifestations (*simultaneous*, e.g. $(A||B||C)$). In the context of a complex event type, certain event types prohibit completing the complex event instance. Such event types can be modeled to be not allowed to occur within parts of the manifestation of a misuse scenario (*negation*).

A very important aspect of the semantics is the ability to specify constraints on the context in which the step types of a complex event type are instantiated. Constraints that can be evaluated by merely inspecting the feature instances of the current event instance are denoted as *intra-event conditions* and can be used for example to select event instances that affect a certain user, host or file. *Inter-event conditions* can only be evaluated by inspecting at least two event instances, which implies to create state. For example, inter-event conditions can be used to correlate event instances that affect the same user, host or file.

2.3 Step Instance Selection

While a given event pattern specifies *when* a complex event instance occurs, the *step instance selection* defines, *which* of the possibly more than one matching event instances is bound to each step type of the complex event instance. This is an important decision, if we are not only interested in the fact that a complex event instance occurred, but if we also need to document the event instances that lead to the complex event instance for further correlation and response. We use three instance selection modes [18]: selecting the *first* or the *last* event instance or *all* event instances that match the given step type. These modes can be used for example to detect when a performance parameter exceeds a threshold, capturing the parameter value when the threshold was exceeded the first time, the current (last) value right

[1] Refer to Meier [18] for details on the semantics of the operators.

before the next step type occurred, or all values for further statistics.

2.4 Step Instance Consumption

The current system state that is relevant for a given complex event instance is reconstructed by binding event instances to the step types of the complex event type, such that the partial instance of the complex event represents the reconstructed relevant system state. After an event instance has been bound to a given step type of a complex event type, the resulting partial complex event instance represents the occurrence of the event instance as well as the modified relevant system state. Some event types describe activities that change features of system state that are relevant in the context of the considered complex event type, for example the destruction of system objects, e.g. process termination and file deletion, or the change of object features, e.g. renaming a file and changing access privileges. Such activities are said to *consume* the relevant system state, which has been created by previous activities, and which is represented by the partial instance of the complex event type (e.g. *(A; consuming B; . . .)*. Other (*non-consuming*) activities do not change relevant features of system state, e.g. reading from a file (e.g. *(A; non-consuming C; . . .)*). Step instance consumption defines, whether a given partial instance of a complex event type can evolve into one or more partial instances by binding an event instance to a given consuming or non-consuming step type, respectively. Since an instance of a consuming step type modifies the system state represented by the partial complex event instance, the partial complex event instance is evolved by binding the event instance to the step type, effectively consuming the old partial complex event instance. However, the partial complex event instance is evolved for each occurrence of an instance of a given non-consuming step type, creating new partial complex event instances representing each occurrence while retaining the old partial complex event instance.

3 Expert Systems

A key characteristic of expert systems is the separation of application knowledge and general problem solving strategies. Application knowledge is specified as facts and rules, while problem solving is realized by the inference mechanisms of the given expert system shell. Because of this separation expert systems can be applied in a flexible way. A significant number of IDSs has been implemented using expert systems (see Sect. 5), based on the following justification: facilitate easy and rapid implementation, and leverage general and optimized inference mechanisms [15, 5].

3.1 General Architecture

Rule-based expert systems automate the evaluation of statements on facts in the fact base and the inference of conclusions, which may derive new facts. For that purpose statement evaluation and conclusion inference are represented as condition-action-rules (IF-THEN rules). An expert system typically comprises the following components (see Fig. 1): a fact base containing valid facts, rules to evaluate facts and to derive new facts, and a rule interpreter to control the inference process.

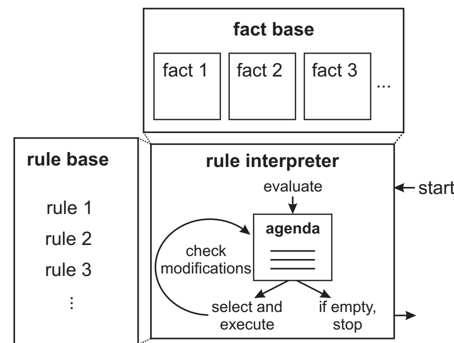

Figure 1. Components of an expert system.

The inference process is composed of two phases. During the *evaluation phase* all applicable rules are determined by evaluating rule conditions. Applicable rules are collected in the *agenda*. During the subsequent *selection phase* exactly one rule is selected from the agenda and is executed. Rule execution may modify the fact base and thereby affect the applicability of rules. Thus, the agenda is updated subsequently and it is checked whether rules become applicable or non-applicable due to fact base modifications. Next, another rule is selected from the agenda and is executed. This procedure is repeated until the agenda is empty and the rule interpreter terminates.

If there are multiple rules on the agenda, distinct alternative execution orders of these rules are possible. Since rule execution may affect the fact base and consequently the agenda, distinct execution orders of rules may result in different fact bases. In this context a set of rules is called *confluent*, if distinct execution orders of these rules result in the same fact base. During the selection phase, which is sometimes called *conflict resolution*, an execution order is determined. This process can be partially controlled by choosing one of the given conflict resolution strategies. Typically rules can be prioritized manually or based on the specificity of their conditions. In general, rule selection is a non-deterministic process.

3.2 The Rete Algorithm

Misuse detection systems based on expert systems implement audit trail analysis using the rule interpreter. A naive rule interpreter iterates over all rules and facts in order to determine applicable rules. The optimization of this procedure has been one main focus of expert system research. Experiments have demonstrated that about 90 percent of the execution time of rule interpreters were required to evaluate the conditions of rules. Consequently optimizations focus on the evaluation phase. Several optimized match algorithms based on similar ideas were developed [8, 20, 21], where the Rete algorithm [8] is the most popular and most used one. Due to the fact that most of the available production quality expert system shells employ the Rete algorithm, to the best of our knowledge all expert systems applied to misuse detection employ Rete (see Sect. 5).

The Rete algorithm is based on two ideas: avoid iteration over rules and avoid iteration over facts. The first idea exploits the observation that the major fraction of execution time is required for condition evaluation. For optimization the rule conditions are organized as nodes in a data flow graph, the so-called *Rete network*. Conditions that appear in multiple rules are pooled together in the Rete network, such that redundant evaluation of conditions is avoided. This technique is comparable to approaches for *Common Subexpression Elimination* used by compilers [2].

The second idea is based on the assumption that the fact base rarely changes. This is exploited by associating facts with the condition nodes in the Rete network, whose conditions they match. When a fact enters the fact base, the rule interpreter finds all conditions that match the fact and associates it with the corresponding nodes in the Rete network. When a fact is removed from the fact base the interpreter again finds all conditions that match the fact and removes the association from the respective nodes in the Rete network. As a result, Rete-based rule interpreters never iterate over the fact base. Conditions are evaluated only when inserting or removing facts.

Table 1. Example facts.

fact no.	object		
	name	type	prop
1	shell	file	executable
2	gcc	file	executable
3	audit	device	readable

fact no.	action	
	type	target
4	write	mail
5	write	shell
6	read	tmp

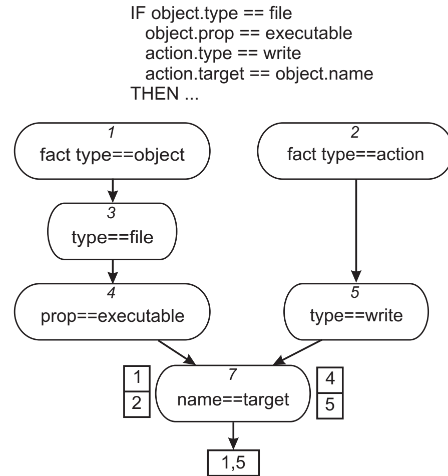

Figure 2. Example rule and Rete network.

Rete is illustrated using the example rule depicted in Fig. 2 and the facts given in Tab. 1. The example rule contains conditions regarding two fact types: *object* and *action*. We use *(fact type).(feature name)* as notation to refer to a feature of a fact type. The rule conditions contain three intra-fact conditions. They restrict the features named *type* and *prop* of the fact type *object* to the values *file* and *executable* respectively, and the feature named *type* of *action* facts to the value *write*. The inter-fact condition requires that the feature named *target* of *action* facts equals the feature named *name* of *object* facts. Figure 2 also depicts the Rete network that corresponds to the example rule. The Rete network starts at the top with nodes that represent the intra-fact conditions. These nodes are called *alpha nodes*. The root nodes at the top of a network always are alpha nodes that test the type of facts. Further intra-fact conditions regarding facts of a particular type are successor nodes of the respective root node. Inter-fact conditions are represented in the Rete network as beta nodes below all alpha nodes. The example contains one beta node for the condition that correlates *object* and *action* facts. While alpha nodes (except for root nodes) contain exactly one (downwards directed) input edge beta nodes contain exactly two input edges. For each input edge a beta node holds a memory, i.e. a *left* and a *right* memory. The left (right) memory of a beta node stores all facts that match the conditions of all nodes along the path from left (right) input edge of the beta node upwards to a root node. If a pair of facts from the left and the right memory of a beta node exists that matches the condition of the node, copies of both facts are merged and moved further downwards the Rete network. Facts or fact pairs that arrive at the bottom (any node without successor) of the Rete network represent facts that match the conditions of the rule represented by the path(s) through the Rete network. When a fact enters the fact base it is provided to all root nodes of

the Rete network. If a fact matches the condition of a node it is forwarded to the successor node. The example facts 1 and 2 in Tab. 1 match all the intra-fact conditions on *object* facts. Thus they are stored in the left memory of the beta node in Fig. 2. Facts 4 and 5 match the intra-fact conditions on *action* facts and are stored in the right memory of the beta node. Only when a new fact arrives at a beta node and is stored in the left (right) memory of the node, it is evaluated regarding the node condition with every fact from the right (left) memory of the node. The pair of facts 1 and 5 matches the inter-fact condition represented by the beta node and thus matches all conditions of the rule. Note that facts 3 and 6 neither match all intra-fact conditions on object facts nor action facts. Thus they do not reach a memory of a beta node and are not stored in the Rete network. When a fact is removed from the fact base it must be removed from all beta node memories. Hence either memory indexing is used (each fact keeps a list of nodes where it is stored) or the fact insertion (matching) procedure is repeated to identify the nodes where a fact is stored.

In a nutshell: Rete is used to avoid that all facts are evaluated in each cycle of the rule interpreter. For the further discussion two strategies of Rete regarding condition evaluation can be identified:

S1: Intra-fact conditions are evaluated only once for each fact entering the fact base. A fact (reaches the memory of a beta node and) is stored in the Rete network, if it meets the conditions on the path from that beta node to a root node.

S2: Inter-fact conditions are evaluated only once per pair of facts. Pairs of facts matching such conditions are stored in the Rete network.

3.3 Applying Expert Systems to Misuse Detection

In expert systems complex event instances are specified as facts and rules. For the purpose of misuse detection there are two types of facts. First there is an *event fact* that represents the current event instance to be analyzed. Second there are *instance facts* that are used to represent (partial) complex event instances. They document the existence and the state of ongoing misuse activity instances. A given step type of a complex event type is described as a (step) rule. The IF part of the rule evaluates the event fact and instance facts regarding the conditions of the step type.

The binding of a matching event instance to a step type is realized in the THEN part of a rule by copying relevant features from the event fact to respective instance facts. That is, an instance fact contains all relevant feature instances of all step instances of the corresponding (partial) complex event instance. Note that instance facts do not necessarily contain all feature instances of step instances. It is sufficient to store only the feature instances that are required for further correlation of event instances (denoted as *correlation features*) or to keep information (*information features*) that is required later on, e.g. to generate rich alerts.

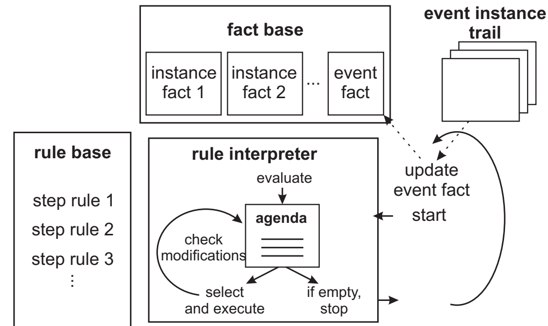

Figure 3. Misuse detection based on expert systems.

The following sketches the general mapping of complex event types to rules and facts. There are two types of facts:

Event fact: representing the current event instance. At all times during analysis there is exactly one event fact in the fact base of the expert system.

Instance facts: representing (partial) complex event instances, containing all correlation and information features of step instances. These facts also indicate which step instances of a complex event type already have occurred, which is required for checking the temporal order of step instance occurrences.

Each step type of a given complex event type is described by a rule. The IF part specifies the conditions of the step type (cf. Sect. 2.1), including constraints on the event type name; intra-event conditions, which are mapped to intra-fact conditions on the event fact; inter-event conditions, which are mapped to inter-fact conditions that either compare distinct instance facts or the event fact with instance facts; and conditions that check the temporal order of step instances by comparing respective features of instance facts.

The THEN part of a (step) rule describes step instance selection, i.e. the binding of event instances to step types by copying features of the event fact to instance facts; step instance consumption by modifying an existing instance fact or a newly created copy of an existing instance fact; and actions associated with step types by executing respective procedures.

The analysis cycle of a misuse detection expert system is as follows (cf. Fig. 3): an event instance is mapped to an event fact, which is then inserted in the fact base. Subsequently the rule interpreter is started and determines

all applicable rules by evaluating rule conditions. One of the applicable rules is selected and executed. Thereafter it is checked whether rules have become applicable or non-applicable due to modifications to the fact base. This procedure is repeated until the agenda is empty and the rule interpreter terminates. Then the event fact is removed from the fact base, a new event fact representing the next event instance is inserted in the fact base and the rule interpreter is started again.

There are two situations where the mapping of complex event types requires special attention. First, the pattern matching realized by misuse detection requires what we call *deterministic matching*, which is generally not implemented by rule interpreters, but can be emulated by adequately specifying rules. Problems occur in situations like the following. We assume a complex event type describing a disjunction e.g. *(A;(consuming B1 OR consuming B2))*. For each step type *B1* and *B2* a rule is specified and it is possible that an event of type *B* occurs that matches both step types. In this case during the evaluation phase both rules *B1* and *B2* are placed (with the same priority) on the agenda and one of them is non-deterministically chosen by the rule interpreter for execution. Execution of the selected rule consumes or modifies instance facts and thereby deactivates the other rule. However for complete misuse detection it is desirable that both rules are executed. Non-deterministic selection of rules out of a non-confluent agenda must be avoided by adequately specifying the rules. In the example the consumption of instance facts has to be delayed until both rules have fired, e.g. both rules do not consume instance facts but create a special fact that indicates that they have fired and an additional consumption rule with low priority consumes the instance fact and removes the special facts. Because of its low priority the consumption rule executes after all (step) rules were executed and only if at least one special fact exists.

The rule interpreter handles all facts in the same way, so its operation cannot be strictly clocked by event facts as is required for misuse detection. Consequently a second problem (*cascaded rule execution*) potentially occurs for sequences of step types matching the same event type. Such sequences can lead to the consecutive execution of several step rules for a given matching event fact. Suppose we have a complex event *(A;A;A)* and the first (step) rule executes when a matching event fact exists. Thereby the rule creates an instance fact representing the occurrence of the step as well as its feature bindings. As a result the second step rule becomes applicable during the analysis of the same event fact. Executing the second rule renders the third rule applicable, still for the same event fact. As a result the three rules are executed for the same event fact. To avoid this, event facts can be numbered and if a rule executes for this event fact its number is stored in the respective instance fact. Additionally, rules are restricted by an inter-event condition to only execute if the number of the event fact and the event fact number stored in the instance fact differ.[2] This solution avoids cascaded execution of rules for the same event fact.

4 Limits Regarding Efficiency

To discuss the efficiency of the Rete algorithm in the application domain of misuse detection we use a simplified but typical example for a complex event type that describes a sequence of activities executed by the same process under certain conditions *(. . . ; A; B; C; . . .)*. The rules and facts specifying the complex event type are given in Fig. 4 and Tab. 2. Rule 1 describes a step type named *A* and requires event instances that contain a feature instance with the name *resource* and the value *x*. Rule 2 represents a step type named *B* and selects event instances with value *42* for the feature instance named *uid*. An event instance of the type named *C* that contains value *2048* for the feature instance named *perm* is required by rule 3. All three rules require that the feature instance named *state* of the (partial) complex event instance contains a particular value (*2*, *3* or *4*) and that the feature instance named *pid* of the event instance contains the same value as the feature instance named *process* of the (partial) complex event instance. Table 2 contains one *event* fact and three facts of type *instance* that represent three partial complex event instances.

The root nodes (1 and 2) of the Rete network in Fig. 4 check the type of the facts to be processed. Alpha nodes 3 to 11 represent the intra-fact conditions. The alpha nodes 3 to 5 check the feature named *state* of partial complex event instances which is used to express restrictions regarding the order of step types. Complex event types do not contain any other *intra-instance conditions* that compare features of complex event instances with constant values[3]. The evaluations of the *state* feature of complex event instances are the sole intra-fact conditions for instance facts. The number of respective alpha nodes is determined by the number of states of the complex event type. Further, each path through the Rete network starting at the root node for instance facts contains exactly one alpha node (besides the root node). The intra-event conditions of the corresponding step types are mapped to intra-fact conditions on event facts. They are represented by the alpha nodes 6 to 11.

Beta nodes of the Rete network represent the inter-event conditions of the step types of the complex event type, i.e. comparisons between distinct complex event instance features or between event instance features and complex event

[2] EXPERT-BSM (cf. Sect. 5) uses a special feature of P-Best to achieve a similar effect. Executing (step) rules mark the event fact with a label, and execute only for unlabeled event facts [15].

[3] Such comparisons are only expressed by intra-event conditions of the step rule that copies the respective step instance feature.

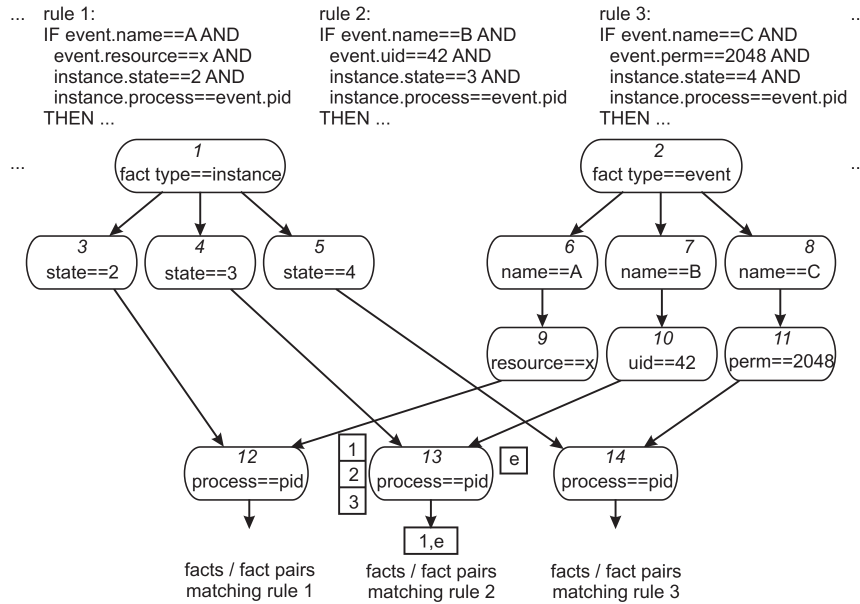

Figure 4. Example (step) rules and Rete network.

Table 2. Example facts.

fact no.	event					
	name	resource	pid	uid	perm	...
e	B	y	17	42	0	...

fact no.	instance		
	state	process	...
1	3	17	...
2	3	19	...
3	3	23	...

instance features. Accordingly the Rete network may contain beta nodes that either relate features of distinct instance facts (denoted as *II-beta nodes*, not shown in our example) or compare features of the event fact with features of instance facts (denoted as *EI-beta nodes*, see nodes 12 to 14 in Fig. 4). Beta nodes 12 to 14 check, if the value of the instance fact feature named *process* equals the value of event fact feature named *pid*.

Since the rules describe step types that are associated with event types, each rule contains at least one condition that refers to the event fact. Moreover holds that step types[4] usually contain an inter-event condition that determines if a complex event instance exists, that applies to the step type. More specifically, these conditions relate feature instances of event instances (*pid* in the example) to the corresponding feature instances of complex event instances (*process* in the example) in order to associate the current event instance with the corresponding complex event instance(s). Hence, a path from a root node to the bottom (any node without successor) of the Rete network usually contains at least one EI-beta node that relates a feature of an instance fact to a feature of an event fact. Furthermore holds that the event fact is continuously changed during analysis operation. As a result, an event fact is used at most once for processing each EI-beta node where it is cached, and is replaced afterwards.

Thus the strategies realized by the Rete algorithm (cf. Sect. 3.2) exhibit the following properties when applied to misuse detection:

S1: Intra-fact conditions are evaluated at most once for each fact entering the fact base.

Event fact: The event fact changes in each cycle and is evaluated at most once per alpha node. Hence, storing event facts at condition nodes to recall matching intra-fact conditions is counterproductive. It necessitates the removal of the event fact from the Rete network.

Instance fact: Instance facts may remain unchanged for a number of cycles. The only intra-fact conditions on instance facts are evaluations of the *state* feature. The number of these state conditions to be checked for a given instance fact depends on the number of step types of the given complex event type. Hence, storing instance facts at condition nodes represents an actual optimization.

S2: Inter-fact conditions are evaluated only once per pair of facts.

[4]Except for *initial step types*, which are the first step types of a complex event type.

II-beta nodes: This is an actual optimization for conditions between different instance facts (represented by II-beta nodes), since instance facts as well as the respective merged fact pairs may remain unchanged for multiple cycles.

EI-beta nodes: This is not an optimization for conditions between the event fact and instance facts (represented by EI-beta nodes), since the event fact changes in each cycle, such that the pair of event fact and instance fact is invalidated in each cycle. Storing the respective fact pair in the Rete network is counterproductive, since it later on necessitates the removal of the fact pair from the network.

Wrapping up, in the application domain of misuse detection the optimizations of the Rete algorithm are only partially effective and partially counterproductive. This raises the question whether the gain of the algorithm exceeds its cost. To our knowledge neither analytical nor experimental evaluations have yet considered this question.

In order to answer the question we conducted the following experiment. We instrumented two IDSs such that we can determine the cumulative number of evaluated conditions during operation. We use the cumulative number of evaluated conditions to measure the efficiency gain achieved by some optimization strategy, because the relevant optimization strategies are geared towards reducing the number of unnecessary condition evaluations.

The first instrumented IDS is CLIPS-IDS [13], which uses the expert system CLIPS for inferences. The second instrumented IDS is our *Signature Analysis Module (SAM)* [19], which uses a domain-specific inference mechanism. For SAM we can selectively enable various optimization strategies, such that we can get an idea how Rete performs in comparison to distinct domain-specific optimization strategies. We compare Rete to the following optimization strategies of SAM: SAM_1 employs no optimization strategy, SAM_3 uses Common Subexpression Elimination and SAM_5 uses the complete set of optimization strategies.

We have described and evaluated the optimization strategies of SAM elsewhere [19] and use the same experimental setting here. In a nutshell, we used a complex event type for the *shell link attack*, which is a classic example for a multi-step attack. The complex event type is composed of 9 step types that match the creation of a link to a *suid* script, its renaming, removal and execution under certain circumstances. It was chosen because of the variety of different conditions of its step types in order to avoid effects of specific implementation techniques of certain predicates, such as string comparison. The first 18.000 event instances of the test data used in [19] were used to conduct the experiment. This event instance trail represents several complex event instances for the shell link attack and two other at-

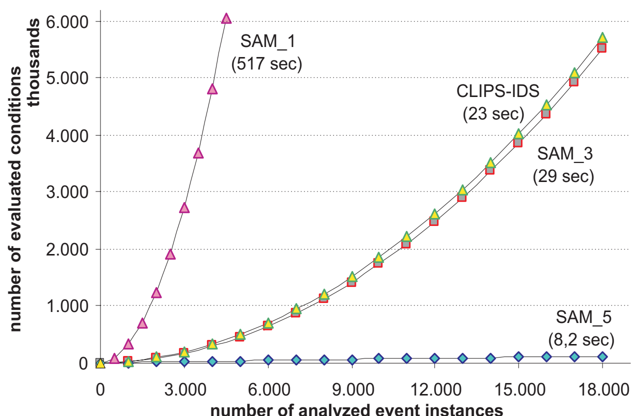

Figure 5. Measurements.

tacks. A detailed description of the attacks and the test data is given in [19]. The execution times required to analyze the 18.000 event instances of the test data are shown in Fig. 5. We however also measure the cumulative number of condition evaluations, because this measure is more conclusive regarding the reduction of condition evaluations achieved by an optimization strategy, independently from machine level effects. We count the number of evaluations on the conditions occurring in our test setting (equal, not equal, string comparison, file permission comparison).

The measurements depicted in Fig. 5 clearly demonstrate that the efficiency gain of Rete in CLIPS-IDS is very close to the one achieved by Common Subexpression Elimination in SAM_3. Intuitively this can be expected, since the Rete network is constructed in such a way that Common Subexpression Elimination is inherently achieved. Note that CLIPS does not need twice as much condition evaluations as SAM_3 does, since CLIPS uses memory indexing, and therefore does not require condition (re-)evaluation on fact retraction. However, the further optimization strategies of Rete discussed above are ineffective in the misuse detection domain, they are even slightly counterproductive when compared to SAM_3. It can also be seen from Fig. 5 that Rete still performs better than the inference engine of SAM without optimizations (SAM_1), but significantly worse than the same inference engine with various optimization strategies (SAM_5).

5 Related Work

This paper critiques the approach of applying expert systems to the problem of misuse detection, because it provides limited efficiency. Our discussion of related work is twofold: expert system based IDSs and efficiency issues.

Over the years the community came up with numerous misuse detection systems that use expert systems for rule in-

ference. All of these expert systems use the Rete algorithm [8] and thus are susceptible to the problems discussed in this paper.[5] P-Best (Production-based Expert System Toolset) [24] has a long history in the IDS domain and has been employed in an older version producing Lisp object code for MIDAS (Multics Intrusion Detection and Alerting System) [24] and IDES (Intrusion Detection Expert System) [16], as well as in a more efficient version producing C source code for NIDES (Next-generation IDES) [4] and Emerald eXpert (Event Monitoring Enabling Responses to Anomalous Live Disturbances) [15]. CLIPS (C Language Integrated Production System) [9] also is a popular choice of IDS developers and was used for Haystack [25], DIDS (Distributed Intrusion Detection System) [26], CMDS (Computer Misuse Detection System) [23], SECURENET [12] and CLIPS-IDS [13], which we developed basically to compare the performance of Rete-based inference engines with domain-specific inference engines. Previously, we employed the commercial RTworks expert system shell [29] for AID (Adaptive Intrusion Detection system) [27]. Hyperview leverages the commercial Ilog expert system [6]. More recently JESS (Java Expert System Shell) [10] was used for SNIDJ (Signature-based Network Intrusion Detection system using JESS) [1] and a hybrid IDS [7]. Rete has also been proposed as the inference algorithm for misuse detection independently from any particular expert system [5, 22]. Further expert system based misuse detection systems have been built, but it is not documented, whether these expert systems use Rete. AudES (Expert system Security Auditing) uses the commercial ESE/VM (Expert System Environment) [30] and NADIR (Network Anomaly Detection and Intrusion Reporter) [11] uses an expert system also developed at LNL, Los Alamos.

Facing ever-increasing audit data volumes and growing signature knowledge bases the efficiency of inference mechanisms employed for misuse detection is of key importance. Several approaches have been proposed to cope with this situation. Sommer et al. and McHugh advocate analyzing more compact, i.e. less detailed, network data [28, 17]. An orthogonal approach are optimizations geared to more efficient inference engines. Kruegel et al. transform signatures into a decision tree to reduce the number of redundant comparisons [14]. This approach is comparable to the efficiency gained by Common Subexpression Elimination as used by Rete. Anagnostakis et al. propose optimized string matching algorithms [3]. These optimizations are restricted to the detection of single-step attacks. We proposed, implemented and evaluated various optimization strategies for the efficient detection of multi-step attacks [19]. These strategies also comprise Common Subexpression Elimination (see Sect. 4), but apart from that the optimization strategies are orthogonal to those employed by Rete.

[5]For RTworks the documentation does not explicitly state that Rete is used, but the description indicates that the algorithm used is very similar to Rete.

6 Conclusion

We have introduced the necessary concepts to understand how misuse detection can be implemented on pertinent expert systems, as well as the necessary concepts to understand the optimization strategies of the Rete algorithm used by such expert systems. Based on a general approach for implementing misuse detection on expert systems we have analyzed the effect of these optimization strategies.

The results indicate that apart from Common Subexpression Elimination the strategies must be ineffective, because they are based on an assumption that does not hold in the application domain of misuse detection: it is assumed that the fact base remains largely static, such that intermediate results can be cached and recalled. However, most of these results involve the current event instance, which continuously changes, such that most of the cached results are immediately invalidated.

Our measurements support that Rete can be used to achieve some optimization, which is limited to the effect of Common Subexpression Elimination. However, the further optimizations of Rete are ineffective and even slightly counterproductive. We have also shown that significant further optimization can be achieved, if we consider domain-specific assumptions [19].

Note that by modifying the Rete algorithm to treat the event fact in a special way significant performance gains can be expected. However, such application specific adaptations contradict the basic idea of employing expert systems (see Sect. 3) in the first place. The motivation of using expert systems is to employ an out-of-the-box ready-to-use inference engine as analysis component. Since to our best knowledge all misuse detection systems that follow this approach employ Rete-based expert systems our results are broadly applicable and relevant.

Expert systems have been used and probably will be used to build IDSs for misuse detection, largely because the approach of taking an out-of-the-box inference mechanism is very appealing for rapid prototyping. Our main conclusion however is, that expert systems based on the Rete algorithm are not useful for IDSs in production environments due to efficiency and usability issues. Alternative algorithms such as TREAT [20] and LEAPS [21] have been developed which may provide additional performance improvements. However there are currently only very few expert systems that support these algorithms. The issue of efficiency of these algorithms in the domain of misuse detection requires further investigation.

References

[1] A. Ahmed and M. Garcia. Signature-based network intrusion detection system using JESS (SNIDJ). In *Proc. of the Int. Conf. on Internet and Multimedia Systems and Applications (EUROIMSA)*, Grindelwald, Switzerland, Feb. 2005.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1988.

[3] K. G. Anagnostakis, S. Antonatos, P. Markatos, and M. Polychronakis. E2xb: A domain-specific string matching algorithm for intrusion detection. In *Proc. of the IFIP TC11 18th Int. Conf. on Information Security (SEC03)*, pages 217–228, Athens, Greece, May 2003. Kluwer.

[4] D. Anderson, T. Frivold, A. Tamaru, and A. Valdes. Next-generation intrusion-detection expert system (NIDES). Technical Report SRI-CSL-95-07, Computer Science Lab, SRI International, Menlo Park, CA, USA, Dec. 1994.

[5] M. Chmielewski, A. Gowdiak, N. Meyer, T. Ostwald, and M. Stroinski. An approach to automate a process of detecting unauthorized accesses. In *TERENA-NORDUnet Networking Conf.*, Lund, Sweden, June 1999. TERENA.

[6] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 240–250, Oakland, CA, USA, May 1992. IEEE Press.

[7] A. El-Semary, J. Edmonds, J. González, and M. Papa. Implementation of a hybrid intrusion detection system using Fuzzyjess. In *Proc. of the 7th Int. Conf. on Enterprise Information Systems*, pages 390–393, Miami, FL, USA, May 2005.

[8] C. L. Forgy. Rete: A fast algorithm for the many pattern/ many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[9] J. C. Giarratano and G. D. Riley. *Expert Systems: Principles and Programming*. PWS Publishing, 3rd edition, 1998.

[10] E. F. Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications, 2003.

[11] K. A. Jackson, D. H. DuBois, and C. A. Stallings. An expert system application for network intrusion detection. In *Proc. of the 14th Nat. Computer Security Conf.*, pages 215–225, Washington, D.C., USA, Oct. 1991. NIST/NCSC.

[12] S. K. Katsikas and N. Theodoropoulos. Defending networks: The expert system component of SECURENET. In *Proc. of the IFIP TC6/TC11 Int. Conf. on Communications and Multimedia Security*, pages 291–302, Essen, Germany, Sept. 1996. Chapman & Hall.

[13] R. Krauz. Implementierung eines auf dem Expertensystem-Tool CLIPS basierenden Intrusion Detection Systems (in German). Technical report, Brandenburg University of Technology Cottbus, Dept. Computer Science, 2004.

[14] C. Kruegel and T. Toth. using decision trees to improve signature-based intrusion detection. In *Proc. of the 6th Int. Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, number 2820 in LNCS, pages 173–191, Pittsburgh, PA, USA, Oct. 2003. Springer.

[15] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 146–161, Los Alamitos, CA, USA, May 1999. IEEE Press.

[16] T. F. Lunt, R. Jagannathan, R. Lee, A. Whitehurst, and S. Listgarten. Knowledge based intrusion detection. In *Proc. of the Annual AI Systems in Government Conf.*, pages 102–107, Washington, DC, Mar. 1989. IEEE CS Press.

[17] J. McHugh. Sets, bags, and rock and roll: Analyzing large data sets of network data. In *Proc. of the 10th European Symposium on Research in Computer Security (ESORICS 2004)*, number 3193 in LNCS, pages 401–422, Sophia Antipolis, France, Sept. 2004. Springer.

[18] M. Meier. A model for the semantics of attack signatures in misuse detection systems. In *Proc. of the 7th Int. Information Security Conf. (ISC 2004)*, number 3225 in LNCS, pages 158–169, Palo Alto, CA, USA, Sept. 2004. Springer.

[19] M. Meier, S. Schmerl, and H. Koenig. Improving the efficiency of misuse detection. In *Proc. of the 2nd GI Conf. on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*, number 3548 in LNCS, pages 188–205, Vienna, Austria, July 2005. Springer.

[20] D. P. Miranker. Treat: A better match algorithm for AI production systems. In *Proc. of the 6th National Conf. on Artificial Intelligence*, pages 42–47. AAAI Press, 1987.

[21] D. P. Miranker, V. Brant, B. J. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proc. of the 8th National Conf. on Artificial Intelligence*, pages 685–692. AAAI Press, 1990.

[22] J.-P. Pouzol and M. Ducass. From declarative signatures to misuse IDS. In *Proc. of the 4th Int. Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, number 2212 in LNCS, pages 1–21, Davis, CA, USA, Oct. 2001. Springer.

[23] P. E. Proctor. Audit reduction and misuse detection in heterogeneous environments: Framework and application. In *Proc. of the 10th Annual Computer Security Applications Conf.*, pages 117–125, Orlando, FL, USA, Dec. 1994.

[24] M. A. Sebring, E. Shellhouse, M. E. Hanna, and A. R. Whitehurst. Expert systems in intrusion detection: A case study. In *Proc. of the 11th Nat. Computer Security Conf.*, pages 74–81, Baltimore, MD, Oct. 1988. NIST/NCSC.

[25] S. Smaha. Haystack: An intrusion detection system. In *Proc. of the 4th Aerospace Computer Security Applications Conf.*, pages 37–44, Dec. 1988.

[26] S. R. Snapp, S. E. Smaha, D. M. Teal, and T. Grance. The dids (distributed intrusion detection system) prototype. In *Proc. of the Summer USENIX Conf.*, pages 227–233, San Antonio, TX, USA, June 1992. USENIX Association.

[27] M. Sobirey, B. Richter, and H. Koenig. The intrusion detection system AID – Architecture and experiences in automated audit trail analysis. In *Proc. of the IFIP TC6/TC11 Int. Conf. on Communications and Multimedia Security*, pages 278–290, Essen, Germany, Sept. 1996. Chapman & Hall.

[28] R. Sommer and A. Feldmann. Netflow: Information loss or win? In *Proc. of the 2nd ACM SIG-COMM and USENIX Internet Measurement Workshop (IMW 2002)*, Marseille, France, Nov. 2002.

[29] Talarian Corporation. RTie inference engine. Technical report, Talarian Corp., Moutain View, CA, USA, 1995.

[30] G. Tsudik and R. Summers. Audes – an expert system for security auditing. In *Proc. of the Conf. on Innovative Applications of Artificial Intelligence*, pages 89–93. AAAI, 1990.