Fast sequential implementation of a lightweight, data stream driven, parallel language with application to intrusion detection

Xavier Martin

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

December 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Prof. Baudouin **Le Charlier** (Advisor)	UCL/INGI, Belgium
Prof. Marc **Dacier**	Eurecom, France
Prof. Mireille **Ducassé**	INSA/IRISA, France
Dr. Abdelaziz **Mounji**	Dexia Group, Belgium
Prof. Axel **Van Lamsweerde**	UCL/INGI, Belgium
Prof. Yves **Deville** (Chair)	UCL/INGI, Belgium

Abstract

The general problem we consider in this thesis is the following: we have to analyze a stream of data (records, packets, events...) by successively applying to each piece of data a set of "rules". Rules are best viewed as lightweight parallel processes synchronizing on each arrival of a new piece of data. In many applications, such as signature-based intrusion detection, only a few rules are concerned with each new piece of data. But all other rules have to be executed anyway just to conclude that they can ignore it. Our goal is to make it possible to avoid this useless work completely. To do so, we perform a static analysis of the code of each rule and we build a decision tree that we apply to each piece of data before executing the rule. The decision tree tells us whether executing the rule or not will change anything to the global analysis results. The decision trees are built at compile time, but their evaluation at each cycle (i.e., for each piece of data) entails an overhead. Thus we organize the set of all computed decision trees in a way that makes their evaluation as fast as possible.

The two main original contributions of this thesis are the following. Firstly, we propose a method to organize the set of decision trees and the set of active rules in such a way that deciding which rules to execute can be made optimally in $O(r_u)$, where r_u is the number of useful rules. This time complexity is thus independent of the actual (total) number of active rules. This method is based on the use of a global decision tree that integrates all individual decision trees built from the code of the rules. Secondly, as such a global tree may quickly become much too large if usual data structures are used, we introduce a novel kind of data structure called *sequential tree* that allows us to keep global decision trees much smaller in many situations where the individual trees share few common conditions. (When many conditions are shared by individual trees the global tree remains small.)

To assess our contribution, we first modify the implementation of ASAX, a generic system for data stream analysis based on the rule paradigm presented above. Then we compare the efficiency of the optimized system with respect to its original implementation, using the MIT Lincoln Laboratory Evaluation Dataset and a classical set of intrusion detection rules. Impressive speed-ups are obtained. Finally, our optimized implementation has been used by Nicolas Vanderavero, in his PhD thesis, for the design of stateful

honeytanks (i.e., low-interaction honeypots). It makes it possible to simulate tens of thousands hosts on a single computer, with a high level of realism.

Remerciements

Je voudrais remercier de tout mon cœur mon promoteur de thèse, le Professeur Baudouin Le Charlier, pour sa patience, ses conseils et ses encouragements constants pendant ces six dernières années. Il a investi un temps considérable ainsi que beaucoup d'efforts à m'aider à réaliser ce travail. Je suis tout simplement bouche bée face à autant de sacrifice personnel et ça m'inspire la plus grande admiration. Merci Baudouin, votre sagesse et votre dévouement ont permis à l'accomplissement de cette thèse d'être une réalité.

Un grand merci à mon jury de thèse qui s'est investi dans l'amélioration de ce travail lors de la dernière ligne droite. Leurs conseils ont été d'une très grande aide.

Merci aussi à tout le monde au département INGI de la faculté des sciences appliquées de l'UCL, et plus particulièrement à Isabelle que j'aimais faire sursauter par taquinerie, François avec qui j'ai passé de très bon moments dans le même bureau pendant plusieurs années, Raphaël et Renaud pour les discussions qu'on a eues sur des pistes de réflexion et Vivianne, Chantal et Fabienne pour les papotages.

J'aimerais également remercier toute l'équipe de l'INSA de Rennes qui m'a accueilli une semaine et plus particulièrement Mireille Ducassé et Jean-Philippe Pouzol avec qui j'ai passé de bons moments.

Un tout grand merci à ma famille qui m'a encouragé tout au long de mon doctorat et plus particulièrement lors de la rédaction. Je respecte la patience dont elle a fait preuve car je ne l'ai plus vue beaucoup à ce moment.

Pour terminer j'aimerais remercier chaleureusement tous mes amis qui m'ont accompagné et encouragé pendant ce travail. Merci à Nicolas, mon frère d'armes dans cette aventure, qui a fini sa thèse de doctorat en même temps que moi. Nous avons partagé tous nos sentiments de joie et de désespoir jusqu'à arriver à l'aboutissement heureux d'aujourd'hui. Un merci tout particulier à mon amie Sandy qui a été ma confidente pendant ces six années de travail et qui m'a aidé à corriger mon anglais. Son humour et son délire m'ont permis de tenir le coup et m'ont remonté le moral un grand nombre de fois. Encore merci Sandy car sans toi et ta "unks attitude", je ne crois pas que j'y serais arrivé. Merci à Yann, Claire et Stéphanie qui ont aussi été présents dans les moments difficiles. A la fois merci et pardon à Alexandra et Patrick à qui j'ai refusé bien des soirées lors de ma rédaction.

Mais ils ont été là pour me soutenir et je les en remercie.

Encore un grand merci à toutes les personnes que j'ai citées ci-dessus car, sans elles, cette thèse de doctorat n'aurait jamais pu voir le jour.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Context of this thesis

Designing specialized programming languages targeted to a particular class of applications may help programmers to solve problems more quickly and more correctly than by using general purpose programming languages such as C or Java. SQL is a well-known example of such an *application-specific language*, designed to access relational data bases. Application-specific programming languages enable the programmer to describe the problem and/or its solution in a more declarative and/or structured way. As a consequence, in such languages, the programmer has less direct control on the efficiency of its solution, which is more dependent of the language implementation.

Intrusion detection is an area of computer security where the notion of an application-specific language appears very relevant because the detection is usually based on the sequential analysis of a stream of "events", which can be audit records, network packets, or better structured objects built by abstracting and correlating raw data. In the signature-based approach of intrusion detection, the programmer has basically only to specify a set of patterns to detect. The underlying system should relieve him from the clerical tasks of reading and formatting the data, and of implementing the pattern matching.

There is a rich variety of programming languages and systems devoted to signature-based intrusion detection. Snort [53, 56] is a popular system, which follows the so-called stateless approach. In its simplest implementation, it applies a large set of rules to every packets of a network link. Each rule consists of a conjunction of simple conditions about the packet header and, usually, of a regular expression to be matched with the payload of the packet. Bro [49, 1] is a more sophisticated academic system, which follows the stateful approach. It is a two-layered system. The first layer filters and abstract packets, and maintains state information about current connexions. The second layer uses a tailor-made programming language to check security

policy constraints. STAT [68, 60] is another system, which uses a generic language to describe signatures, based on state diagrams. P-Best [36] is similar to STAT but it uses a different kind of programming language based on production rules. Finally, proposals for more specific and declarative signature description languages have been proposed in [51, 50, 44, 54].

All these approaches have in common the fact that they potentially have to apply a large set of "rules" (which may be very different things depending on the system and language) to every item of the data or event stream to be analyzed. Since different rules may perform similar tasks on the same item, any direct or naive implementation of such systems entails a lot of redundant work, which makes the system too inefficient. In this thesis, we address the issue of making them significantly more efficient.

In order to assess our contribution, we use –i.e., we modify the implementation of– ASAX [25, 46], a generic system that allows stateful analysis of any data stream through the use of its programming language RUSSEL. The reasons for using ASAX instead of another system are the following:

- ASAX is a very simple system with a clean architecture. Moreover, RUSSEL rules are compiled to an internal form very suitable for static analysis. We also had a good expertise of this system, making it easier to modify its language and its implementation.

- The original implementation of ASAX is efficient in the sense that is very simple and "light" but it is also very naive in the sense that all active rules are systematically applied to the current event. Thus the original implementation was a good reference to assess the benefit of the optimizations we propose.

1.2 Contributions of this thesis

The general problem we consider in this thesis is the following: we have to analyze a stream of data (records, packets, events...) by successively applying to each piece of data a set of "rules". We use the term "rule" to agree with the terminology used for signature-based intrusion detection. However, rules should be better viewed as parallel processes synchronizing on each arrival of a new piece of data or "event". Our goal is to schedule the processes at the beginning of each such cycle (i.e., each arrival) and, in particular, to decide which processes need not be executed at all. To do so, we perform a static analysis of the code of each "rule" (i.e., process) to build a decision tree that we apply to each piece of data before executing the rule (i.e., scheduling the process). The decision tree tells us whether executing the process or not will change anything to the global analysis results. More precisely, the decision tree computes a sufficient condition for the fact that executing the rule has no effect on the sequel of the analysis. Since decision

trees are built at compile time, their construction has no impact on the execution time but their evaluation at each cycle (i.e., for each piece of data) entails an overhead. Thus it is desirable to organize the set of all computed decision trees in a way that makes their evaluation as fast as possible. We consider that a most ambitious objective is to be able to take all decisions in $O(r_u)$ where r_u is the number of "useful" rules, i.e., the rules that must be executed. Indeed, if we reach this objective, evaluating the decision trees does not increase the computational complexity of evaluating the right rules.

Considering this general picture, the main original contributions of this thesis can be summarized as follows.

1. We propose a method to organize the set of decision trees and the set of active rules in such a way that deciding which rules to execute can be made optimally in the sense defined above. Note that since the complexity is $O(r_u)$, it is independent of the actual number of active rules. We will provide a case study where the number of active rules may become arbitrarily large while the number of useful rules remains very small. The techniques and algorithms we propose for this issue constitute Chapter 7.

2. The method discussed in the previous point is based on the use of a global decision tree that integrates all individual decision trees built from the code of the rules. It appears that such a global tree may quickly become much too large if usual data structures for decision trees are used. We introduce a novel kind of data structure called *sequential tree* that allows us to keep global decision trees much smaller in many situations where the individual trees share few common conditions. (When many conditions are shared by individual trees the global tree remains small.)

Before giving a summary of what is presented in this work, we stress a last important point. This doctoral research was done in parallel with the work of Nicolas Vanderavero. In his PhD thesis [65], Nicolas Vanderavero proposes a framework to build efficient low-interaction honeypots, called honeytanks, and, more generally, to build various kinds of "traffic handlers" to simulate large numbers of hosts on a single machine. Nicolas Vanderavero's framework is based on ASAX and the use of ASAX for building honeytanks and traffic generators has provided the most striking examples for the usefulness of our own contribution. Thus, we refer to his work to get significant programming examples and applications where RUSSEL is used as a real parallel programming language, very efficient to write honeytanks and other traffic handlers. During our collaboration, he was using ASAX when I was modifying and improving its implementation. We did not modify the original syntax and semantics of RUSSEL, except on one

point: we introduced together the new notion of *frozen variable*, which both facilitates parallel programming with RUSSEL and allows more powerful optimizations. This common contribution is described in both PhD thesis.

1.3 Content of this thesis

Apart from the introduction (this chapter) and the conclusion (Chapter 8), the thesis is divided into two main parts. The first part provides background material and related work. The second part describes the original work.

1.3.1 Background

We start the background part with a description of ASAX and its original implementation. We provide an efficiency evaluation of the original implementation using a benchmark adapted from STAT, which is completely described in Appendix A. We then analyse what are the main aspects of the original implementation that may cause inefficiencies. This presentation constitutes Chapter 2.

We continue with a presentation of the related work in Chapter 3. We first analyze the literature on decision trees and diagrams. Then, we present previous work on optimizing signature based-intrusion detection and production-based languages as well as a few related areas. We also discuss the various kinds of languages and formalisms that have been proposed for signature-based intrusion detection. To finish this chapter, we give a more detailed description of two interesting systems, i.e., STAT and Bro. Chapter 2 comes before Chapter 3 to allow, in Chapter 3, some discussions that would be difficult to follow otherwise.

1.3.2 Our work

We start the presentation of our own work in Chapter 4. This chapter presents a first simple improvement to the implementation of ASAX to lower the work done at each ASAX cycle. An experimental evaluation is provided on the STAT benchmark. It is continued in the next chapters as soon as new improvements are proposed.

Chapter 5 presents a static analysis of ASAX code that allows us to extract decision trees from the internal code of RUSSEL rules. We show that a sequential evaluation of these decision trees provides a significant speed-up on the benchmark.

Chapter 6 describes one of the two main contributions of this thesis. We introduce several kinds of data structures to represent global decision trees built by merging the local decision trees from Chapter 5. In particular, we introduce the new notion of *sequential tree* and we present and prove the correctness of a clever algorithm to merge sequential trees. We show

on the benchmark that it is not possible to get an acceptable compilation time without using these new data structures. We discuss various orders that are suitable to get the size of the global tree small without making useless evaluations of conditions. The experimental evaluation shows that the benchmark is evaluated two times faster with this improvement.

Chapter 7 contains our second main contribution. We address the problem of using the global decision tree of Chapter 6 to extract from the set of all active rules only the rule instances that need be executed. The problem is difficult when one has many instances of the same parametric rules of which only a small number needs be executed. We introduce several low-level implementation techniques, based on the use of "magic numbers", that make it possible to extract the relevant rules with an overhead proportional to the number of the rules that must actually be executed. Parametric conditions are evaluated only once in the global decision tree by introducing hash tables that are generated automatically at compile-time. The STAT benchmark is not adequate to assess the work presented in this chapter because most rules of the benchmark are stateless and non parametric. Thus, we borrow the honeytank of Nicolas Vanderavero [65] to demonstrate our claim that the execution time of an ASAX cycle can be made proportional to the number of actually useful instances when many (i.e., several thousands) rule instances are active.

Part II

Background

Chapter 2

ASAX: A system to perform many parallel analyses on a data stream

2.1 Introduction

ASAX was originally designed to carry out efficient parallel analyses on any kind of data stream. Its efficiency stems from a simple memory management, allowing the system to use neither explicit allocation and deallocation mechanisms nor garbage collectors. Though the efficiency of ASAX is proven in situations featuring simple analyses, it suffers a lack of performance in more complex situations involving many parallel analyses. In this chapter, we experiment these situations and we show that it is amenable to several optimizations. However, to understand the optimizations we develop further in this thesis, some important aspects about ASAX and its implementation need to be described in details. More precisely, we need to depict all mechanisms involved within the analysis process.

The chapter is organized as follows. In Section 2.2, we present the architecture and the general features of ASAX. Section 2.3 provides an overview of the language RUSSEL. Section 2.4 outlines general approaches to write analysis programs properly. In Section 2.5, we depict in details the implementation of RUSSEL, the analysis process and the memory management. This section also presents a high level representation of RUSSEL rules which we will use in the following chapters. In Section 2.6, we carry out an experimental evaluation of ASAX efficiency with the help of a benchmark featuring intrusion detection analyses. The benchmark involves signatures which illustrate the performance issues we want to underline. Finally, Section 2.7 draws the conclusion and proposes several optimizations to overcome the encountered performance issues.

2.2 Presentation of ASAX

In this section, we introduce ASAX and its architecture. We limit our presentation to a description of each feature having an important role in the analysis process.

2.2.1 Using the bottom-up rule-based language RUSSEL

ASAX uses the rule-based approach to implement detection mechanisms. This approach has already been used by many intrusion detection systems such as STAT [27, 19], P-Best [36] or GnG [44]. Analysis rules for ASAX are written in the RUSSEL language (RUle-baSed Sequence Evaluation Language) which was designed specifically to handle large audit trails.

RUSSEL offers guarded commands "à la Dijkstra" [15] of the form *Condition* → *Action*. They allow the programmer to design easily and efficiently analysis rules: should the analyzed audit event satisfies the *Condition*, the associated *Action* is performed. The later can be a filtering for further treatment, a specified action (like modifying variables used by the analyses), or an alarm report.

The language is capable of a high power of expression making it adaptable to any type of analysis (not only intrusion detection). RUSSEL offers an imperative style of programming, as opposed to the declarative one. It uses the classical statements of an imperative language such as assignment, routine call, block of actions, conditional action and repetitive action. In addition, RUSSEL exhibits a specific action designed for sequential analysis: *rule trigger*.

To perform several analyses in parallel on the same data stream, the adopted search strategy has to be bottom-up. Using a top-down approach such as Prolog is not efficient enough because it would imply backwards on the audit events. The bottom-up search process of RUSSEL starts with the audit records as basis facts and attempts to match event sequences thanks to the guarded commands mechanism.

The *rule trigger* is a concept built on the sequential approach. In ASAX, declarations and activations of rules are distinct. Rules need to be activated by an explicit trigger. Triggers are invoked by rules within action blocks. However, to ensure an effective execution of a RUSSEL program, a bootstrap rule has to be declared.

2.2.2 Handling stateful analyzes

Attacks may be composed of a single event or of a sequence of multiple events. Sometimes the order of the occurring events is important to identify the attacks. Such attacks are modeled by scenarios ASAX has to be capable

to detect to be exhaustive. Moreover, distinct instances of the same scenario should be detected individually.

Sequence of events can be described as finite state automata. Translating finite state automata is straightforward in RUSSEL. Information about the past states is kept by passing parameters to rules while triggering them. Transitions between states are implemented in the rules by guarded commands, which trigger new (instances of) rules for next state. Multiple instances of a single rule can be triggered whenever many instances of a signature are interleaved.

A deeper description of stateful rules is given in Section 2.4.2. In addition, Section 3.4.4 depicts a straightforward method to translate state transition diagrams defined in the STATL formalism [19] to RUSSEL.

2.2.3 Universality

One of the main features ASAX offers is universality. Analyses are performed on any type of audit trail. Actually the ASAX evaluator processes events in NADF format (Normalized Audit Data Format) only, which is its internal format. The rationale of defining this format is to allow existing audit trails to be translated to it in a straightforward way. Moreover, using the NADF format allows one to mix heterogeneous audit streams and to apply different kinds of analyses on them, independently to a specific application domain. The simplicity of the NADF format entails thus a simple design for the analysis process.

Other external formats can be translated to NADF by a so-called *format adapter*. The purpose of this component is to collect events recorded in the audit trails and to deliver them to the ASAX evaluator after being converted to NADF format. A joined file (called ADDF file) makes the mapping between fields in NADF records and identifiers used in RUSSEL programs.

Possibilities offered by the format adapter are not limited to a conversion task. Additional tasks may be performed such as event pre-filtering, multi-source mixing or multi-native-format mixing.

2.2.4 Runtime architecture

Figure 2.1 depicts the ASAX runtime architecture. It shows the different components and how they interact. The ASAX evaluator is a virtual machine which applies RUSSEL rules to NADF records produced by the format adapter. The compiler translates rules into abstract machine instructions for the virtual machine. Alerts are raised depending on the rule execution and audit contents.

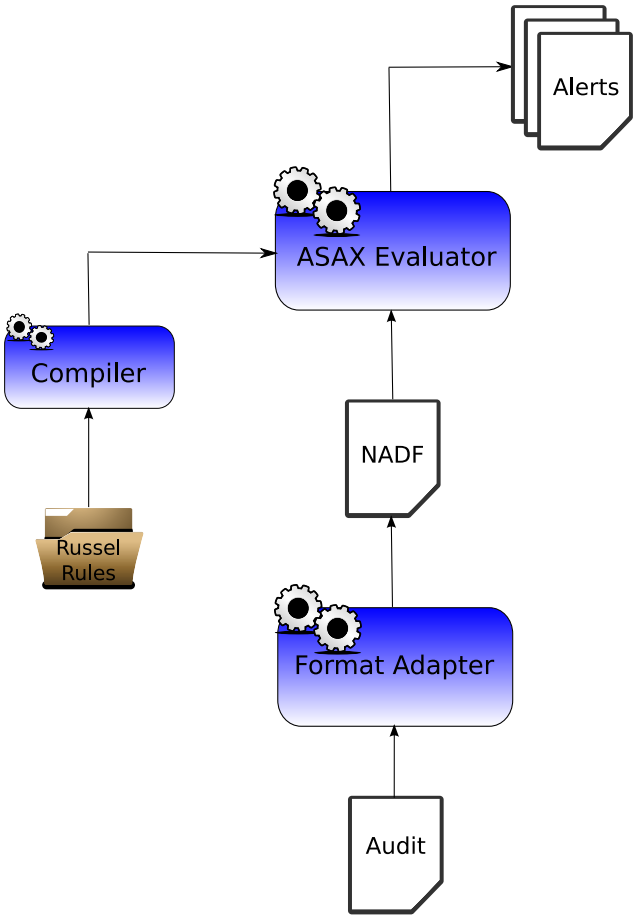


Figure 2.1: ASAX architecture

2.3 The RUSSEL language

Let us now describe RUSSEL. We first start with a walk-through of the rule evaluation process, followed by an informal presentation of the language.

2.3.1 Evaluation process

To perform several analyses in parallel, rules potentially have more than one instance running. Instances are created explicitly by *trigger* statements, except the bootstrap rule called `init_action` whose one instance is created automatically at the start of the analysis. Rules instances can be triggered either for the current analyzed record, for the next record or for a post processing purpose after the analysis. The ASAX evaluator thus maintains three distinct sets of instances that are called *current set*, *next set* and *completion set* in the following of the thesis. Also, whenever we use the

"active rule" term, we refer to all its instances in the current set.

The evaluation process is a simple loop. But first, the analysis is initialized by activating the bootstrap rule: `init_action`. This rule does not apply to a NADF record but it is intended to start the analysis by initializing global variables and triggering rules for the first NADF record. Afterwards, audit records are read and analyzed one by one. Figure 2.2 illustrates what we call an *ASAX cycle*. It consists of executing all instances contained in the *current set* with respect to the current record. These instances perform actions like computing, raising alerts or triggering rules. As shown in Figure 2.3, once the current set is entirely processed, instances in the next set switch to the current set and the process is iterated for the next record until there is no more record to read or there is no active rule to continue the analysis. Then, the completion set is finally executed as depicted in Figure 2.4.

It has to be noted that there is no specified order for the instances of rules to be applied to the current record. The programmer should thus not assume a particular execution order. Moreover, rule instances have to be re-triggered explicitly to stay alive for further analysis. Thus, there is no need for an explicit mechanism to kill instances of rules.

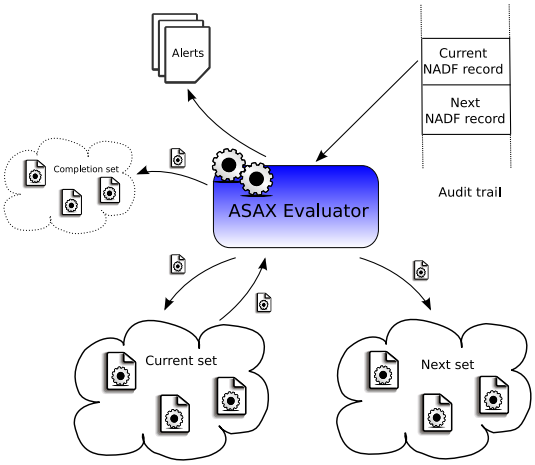


Figure 2.2: Rule evaluation: ASAX cycle

2.3.2 Overview of the RUSSEL language

A complete description of the syntax and the semantics of RUSSEL is given in Mounji's thesis [46]. As a clear understanding of the programming language is needed to understand its main features, we give a brief overview of the main features of the language.

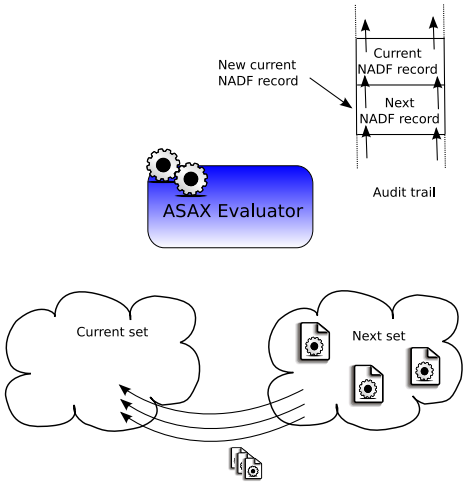


Figure 2.3: Rule evaluation: between two cycles

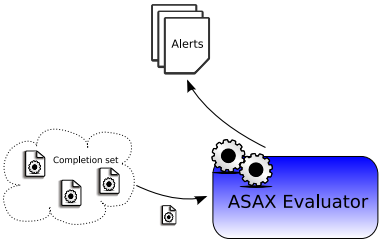


Figure 2.4: Rule evaluation: processing *completion* rules

Data types

RUSSEL supports two data types: *integers* and *strings of bytes*. Integers are signed and 32-bits encoded[1], while strings of bytes represent "raw data". Because of the potential heterogeneity of the data contained in analyzed trails, ASAX designers choose to adopt a generic approach. Strings of bytes are used to contain any type of treated data during analyses. Hence, all data in NADF records are encoded by strings of bytes. These data can then be interpreted with a specific type by using user-defined routines added to the language.

[1]In the original definition of RUSSEL, the size of integers was unspecified. But for many practical applications, it is useful to know the actual size of integers

Environment

The environment in RUSSEL is composed of variables (either global or local), rule parameters and NADF fields. The later are accessed thanks to identifiers provided in an external file. This file contains a mapping between the identifiers and the physical representation of the NADF fields. It usually comes along the used format adapter. Using an external file allows us to modify the identifiers any time without recompiling the system.

Syntactic constructs

The purpose of this section is not to show the precise syntax of RUSSEL but to give general ideas about the main constructs of the language:

- **Modules**: RUSSEL programs usually contain many rules grouped in several files called *modules*. Modules can import other modules with the `uses` keyword. For instance, to group all detection modules to run a complete IDS analysis, a file can be written as follows:

  ```
  uses module1,
       module2,
       ...
       module_n;.
  ```

  Each module may have its own bootstrap rule, i.e., `init_action`. All of them are executed at the beginning of the analysis. Rules can trigger other rules present in other modules.

- **Global variables**: Global variables are supported in RUSSEL. The variable scope can be limited either to the module containing its declaration, or extended to external modules. In the later case, the variable has to be declared (as external) in all modules intending to access it. The syntax to declare global variables is as follows:

  ```
  global <scope_1> g1: type_1;
  ...
  global <scope_n> gn: type_n;
  ```

  where `<scope_`$_i$`>` is either *internal* or *external*.

  During the development of this thesis, we extended the global variable features with the so-called *frozen* concept. Global variables declared as *frozen* have two values: a read-only value and a write-only value. Assignments modify the write-only value while expressions access the read-only value. The purpose of frozen variables is to make certain values constant for a whole ASAX cycle. Therefore, at the beginning

of each ASAX cycle, the write-only value is copied into the read-only one and takes an initial value back, i.e. 0. To declare a global frozen variable, the `frozen` keyword is added to the declaration as follows:

```
global <scope> frozen gf: type;
```

Section 2.4.2 fully describes, with an example, the relevance of using global frozen variables to match multi-event signatures.

- **Rule declarations**: Rule declarations are composed of a header and of a block of actions. The header contains the rule name, optional parameters and optional declarations of local variables. It is illustrated by the following:

```
rule foo(p1: type_1; p2:type_2; ... ; pn:type_n);
<local declarations>
<action block>
```

In case the rule has no parameters, parentheses are omitted. Local declarations take the following form:

```
var l1: type_1;
...
var lm: type_m;
```

For the specific case of rules accessible from external modules, the keyword `external` is added to the declaration:

```
external rule foo(p1: type_1; p2:type_2; ... ; pn:type_n);
```

- **Action blocks**: An action block is composed of a single action or of a sequences of actions. In the later case, the actions are grouped and surrounded with the `begin` and `end` keywords:

```
begin
   action_1;
   ...
   action_n
end
```

Also, semicolons are separators and not terminators as opposed to C or Java for example.

- **Conditional actions**: Conditional testing is based on the Dijkstra approach of guarded commands and has the following form:

if
```
    condition_1 --> action_1;
    ...
    condition_n --> action_n
```
fi

Each line is a guarded command. Conditions are evaluated in the appearance order. The evaluation stops at the first satisfied condition and the corresponding action is executed. The `if` instruction does not have an `else` clause. These can be easily simulated by adding a condition which is always evaluated to true. So, the test *if condition then action_1 else action_2* is written as follows:

if
```
    condition --> action_1;
    true      --> action_2
```
fi

- **Repetitive actions**: Repetitive actions are also based on the guarded commands from Dijkstra. However, all actions associated to satisfied conditions are executed and the process re-iters until no condition satisfies anymore. It presents a form similar to condition actions:

```
do
    condition_1 --> action_1;
    ...
    condition_n --> action_n
od
```

- **Rule Trigger**: To trigger rule instances, the `trigger off` keyword is used along another keyword referring to the targeted rule set, the name of the rule to trigger and optionally, actual parameters.

  trigger off `<scope> foo(p1, p2, ..., pn)`

  where `<scope>` is either `for_current`, `for_next` or `at_completion`, which respectively refers to the current set, the next set and the completion set. In case the triggered rule takes no parameter, the parentheses are omitted.

Testing the presence of a NADF field

Since NADF records have a various number of fields, it is possible, in RUSSEL, to test the presence of a given field with the help of the specific keyword `present`. A condition testing the presence of a field presents thus the following form:

```
if present nadf_field
```

Extending RUSSEL

Sometimes the programmer may need functionalities that are not provided by RUSSEL. For example, such capabilities may include regular expression matching, converting an IP address to a human readable form or manipulating complex data structures. ASAX offers a built-in mechanism to add extra functions to RUSSEL either as dynamic or static libraries. The later needs a recompilation of the ASAX binary.

2.4 Methodology for writing rules

Most ASAX power relies in RUSSEL. Though it allows to program a large number of functionalities in a straigthforward way, crafting analysis rules properly needs the programmer to get rid of old habit. This section aims at showing the difficulties the programmer might encounter. Usually, rules matching mono-event signatures, i.e., stateless rules, are easy to implement. The current record is matched if its content satisfies a list of criteria. Rules detecting multi-events signatures, i.e., stateful rules, are less simple to craft. Events must be correlated, implying rule parameters to keep the past states in memory. We provide generic patterns to write both stateless and stateful rules[2]. In addition, we present a method to detect non-overlapping signatures introduced in Vanderavero's PhD thesis [65].

2.4.1 Writing stateless rules

Stateless rules aim at matching mono-event signatures. They are rules that do no have parameters and that re-trigger during the whole analysis. Selecting events satisfying a condition is the usual used pattern. This approach can be implemented in RUSSEL by modules of the form presented in Figure 2.5. The `<selection condition>` is usually a conjunction of criteria the current analyzed event should match. It implies thus the use of compound conditions made of elementary ones. The records for which the `<selection_condition>` is false are ignored by the rule. It has to be noted that the rule systematically re-triggers itself at the end of its execution to remain alive during the whole analysis. This pattern is called *selection* or *filtration* in the following of this thesis.

A concrete example: detecting buffer overflows. The RUSSEL module presented by Figure 2.6 illustrates the selection approach. That module aims at detecting all buffer overflow attacks in an event flow. A buffer overflow attack consists in storing malicious code outside the bounds of a buffer. This code overwrites useful data and hence changes the behavior of

[2] A more exhaustive list of patterns is presented in the ASAX user's manual [47]

```
init_action;
 trigger off for_next process_record;

rule process_record;
begin
  if
    <selection_condition>
    --> <process the current record>
  fi;
  trigger off for_next process_record
end;
```

Figure 2.5: RUSSEL pattern for record *selection*

the targeted program, potentially leading the attacker to upper privileges. We define that an event has to satisfy the following conditions to be selected as a buffer overflow:

1. The event is a program execution.
2. This execution is successful.
3. The executed binary has its *setuid-bit* attributed.
4. The user targeted by the buffer overflow attack (usually `root`) is different than the attacker.
5. The length of the buffer containing the arguments of the execution is greater than a chosen value (128 in our example).
6. The same buffer contains non-ascii character (probably op-codes).

The RUSSEL code in Figure 2.6 presents some simplifications; some accessed NADF fields are integers but they are physically represented by strings of bytes. Normally, we should use a dedicated function to convert such field contents to integers. For clarity, we hide these functions. Moreover, instead of using integer literals to test fields, we use explicit constants (which are not present in the actual implementation) to make the rules easier to read.

The criteria stated above are directly applicable in the form of a conjunction of elementary RUSSEL conditions. The RUSSEL module involves a number of external user-made routines to perform its analysis: the function `is_suid(file_mode)` verifies the file mode `file_mode` includes the set-uid bit, the function `stringlen(s)` returns the length of the string `s` and the function `contains_non_ascii(buf)` checks if the buffer `buf` contains non-ascii characters.

```
global internal max_len: integer;

init_action;
begin
  max_len := 128;
  trigger off for_next bufov
end;

rule bufov;
begin
  if
    EVENT = exec and RETURN_VALUE != failed and
      is_suid(FILE_MODE) = 1 and USER_ID != AUDIT_ID and
      stringlen(EXEC_ARGS) > max_len and
      contains_non_ascii(EXEC_ARGS) = 1
      --> trigger off for_current alert
  fi;
  trigger off for_next bufov
end;

rule alert;
println('user: ', AUDIT_ID,
          ' - potential buffer overflow in ', PATH).
```

Figure 2.6: Stateless RUSSEL module detecting the buffer overflow attack

2.4.2 Writing stateful rules

Correlating events

Detecting stateful signatures, i.e., multi-steps scenarios, consists of matching sequences of events. The detection process can be modeled by non deterministic finite state automata. Let us assume the event sequence A, B, C. We want to detect all occurrences of this sequence of which all events belong to at most one occurrence. Taken individually, the events may not be related to each other and might not belong to the same instance of scenario. Events should be *correlated* before being selected. For instance, events fulfilling the conditions to be a `B` need to contain information related to the previously matched `A`. Also, records fulfilling the conditions to be identified as `C` should contain information related to the previously matched `A` and `B`. Figure 2.7 presents the corresponding detection automaton. The states represent the steps of the detection progression and the transitions are labelled with events. As the state $find\_A$ is the initial state, it has to remain alive

to ensure the detection of all occurrences of the signature. Consequently, the state presents a loop transition with no label, meaning that it is always fired (for all events). The transition labeled *else* means that it is fired if none of the other transition going from the same state is fired.

Non deterministic finite state automata are translated to RUSSEL in a straightforward way. States are rules and transitions are trigger actions. Figure 2.8 presents a generic pattern to apply correlation on selected records. As we can see, events are selected by distinct rules, each of the latter applying selection conditions to filter the records. Rules `find_B` and `find_C` are stateful rules, meaning that they include parameters representing information about the past states (in order to correlate the records) and so they are not systematically active. The rule `find_B` parameters `A_par_1, ..., A_par_n` contain values of some fields of a previously matched record A. Events B are selected only if they additionally fulfill the extra condition `<corresponds_to_A_par_1, ..., A_par_n>`, which means that B is related to A. The rule `find_C` behaves similarly. However its parameters `AB_par_1, ..., AB_par_m` may represent information from both events A, and B. Finally, the action executed when record C is detected may involve fields from the three records A, B and C.

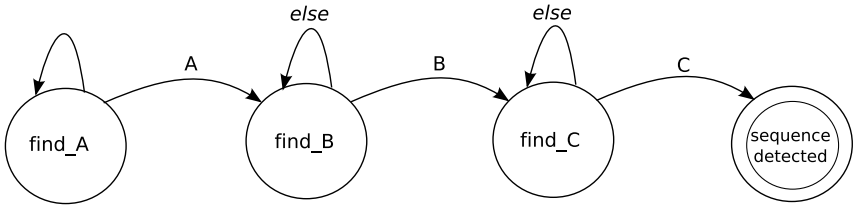


Figure 2.7: Detection automaton of event sequence A B C

Finding non-overlapping signature occurrences

The methodology presented in ASAX's manual [47] has recently been improved by Vanderavero [65] by developing a methodology to find all non-overlapping occurrences of a signature efficiently. We give a summary of this methodology. For a full description, we invite the reader to refer to the chapter 3 in [65].

Assume the signature `[A,A]`. We want to detect all occurrences of the event `A` followed by another event `A`. There can be other events in-between. Let the sequence of events `[A,B,A,A]` be the event flow. There are 3 overlapping occurrences of the signature: the first event with the third event, the first with the fourth one and the third with the fourth one.

For some analyses, like detecting a sequence of several failed logins from the same user, it is convenient to match non-overlapping occurrences of the

```
init_action;
trigger off for_next find_A

rule find_A;
begin
  if
    <is_a_A>
    --> trigger off for_next find_B(A_field_1, ..., A_field_n);
  fi;
  trigger off for_next find_A
end;

rule find_B(A_par_1, ..., A_par_n: string);
if
  <is_a_B> and <corresponds_to A_par_1, ..., A_par_m>
  --> trigger off for_next find_C(AB_field_1, ..., AB_field_m);
  true --> trigger off for_next find_B(A_par_1, ..., A_par_n)
fi;

rule find_C(AB_par_1, ..., AB_par_m: string);
if
  <is_a_C> and <corresponds_to AB_par_1, ..., AB_par_m>
  --> <sequence detected with ABC_field_1, ABC_field_p>;
  true --> trigger off for_next find_C(AB_par_1, ..., AB_par_m)
fi.
```

Figure 2.8: RUSSEL pattern for record correlation

signature only. That means that there cannot be other event `A` between the two `A` of the signature. In other words, each event `A` must belong to *at most one occurrence* of the signature. In that situation, only the first event with the third one should match in the example. Since the third event already belongs to an occurrence, it cannot start a new occurrence. But the fourth event starts a new one.

In general, situations are more complicated because events need to be correlated to belong to the same signature. Events contain attributes acting as an identifier for the signature occurrence they belong to.

The methodology developed by Vanderavero proposes the following pattern to detect all non-overlapping occurrences of a signature. For each of the n ($n > 1$) events $E_2, \ldots, E_n$ composing the signature, a rule `find_i` ($1 < i \leq n$) is defined. The task of rule `find_i` ($1 < i \leq n$) is to select an event E_i and to correlate it with the previous events of the same signature.

We also define a rule `find_1` that detects the first event, i.e., E_1, of all signature occurrences. In addition, it must also ensure that this event is not already belonging to an existing occurrence. The invariant of the pattern is defined as follows:
At each ASAX cycle,

- There is only one instance of rule `find_i` $(1 < i \leq n)$ for each signature occurrence partially matched at that moment.
- For a given signature occurrence, when an instance of rule `find_i` $(1 < i \leq n)$ exists, there is no instances of rule `find_j` $(i \neq j, 1 < i \leq n)$.
- There is exactly one instance of rule `find_1`. (It remains active to detect all occurrences of the signature.)

The difficulty of this problem lies in the fact that the rule `find_1` must know if the event it selects is already belonging to another signature occurrence, that is if the event is treated by another rule instance. An intuitive but incorrect approach would be as follows. When an instance of rule `find_i` selects an event E_i as part of the signature occurrence it follows, it sets a global boolean variable to true. Then, `rule_1` checks that variable. If it is true, it means that the current event has already been treated by another rule and, hence is not the start of a new occurrence. Otherwise, it means that the event does not belong to any current signature occurrences. This approach is not correct because it assumes that rule `find_1` is executed after all others. However, no assumption can be made on the execution order of rules in ASAX.

A correction to the approach consists in forcing the rule `find_1` to take a decision about an event after the execution of all other instances of the current cycle. A manner is to delay the decision of starting a new signature occurrence by one ASAX cycle. That is, the rule `find_1` waits for the next ASAX cycle to know if a rule instance treated the previous event during the previous cycle. In that case, it is sure that all other rule instances of the previous cycle have been executed.

Global frozen variables. To achieve that, rules `find_i` $(1 < i \leq n)$ must put the information that they have treated an event in a variable whose content is only accessible for the next cycle. The concept of *frozen global variables* has been introduced for that purpose. These variables are made of two parts: a read-only and a write-only area. All read-accesses to a global frozen variable return the value stored in the read-only area. All write-accesses modify the value stored in the write-only ares. At the end of each ASAX cycle, the value in the write-only are is copied to the read-only area and the value is the write-only area is reinitialized to zero. Using global frozen variables ensures hence two statements defined by the following invariant:

- The value returned by read-accesses is constant during an whole ASAX cycle.
- Any value assigned to a frozen global variable is only effective for the next ASAX cycle.

The final procedure is thus the following: rules `find_i` ($1 < i \leq n$) must use a global frozen variable `flag` to mark that the current event has been treated during the current cycle. In that case the value 1 is assigned to `flag`. When `find_1` finds an event E_1, it does not start systematically a new signature occurrence. It must wait for the execution of all other rule instances to take a decision. It triggers then a rule `delayed_find_1` for the next cycle and passes the attributes in E_1 identifying the occurrence as parameters. The rule `delayed_find_1` checks the value of the global frozen variable `flag`. If its value is 1, it means that the previous event has been treated during the previous cycle, i.e., it already belongs to an existing signature occurrence. If its value is 0, it means that the previous event is the first event of a new signature occurrence. In this situation, the rule triggers an instance of rule `find_2` with the identifier of the new occurrence as parameter. That rule will check if the current event is the next event in the signature. (This is thus triggered for the current cycle). If it is not the case, it re-triggers itself to find the next event in the following in the event flow. Figure 2.9 depicts the final pattern of the approach. It has to be noted that the variable `flag` is never reinitialized to 0 manually.

A concrete example: detecting a sequence of 3 failed logins by the same user. Figure 2.10 contains an example of a RUSSEL module using a global frozen variable. It aims at detecting all sequences of three failed attempts to log in by the same user and displays the number of matched sequences after the analysis. It is obvious that it must use the frozen variable concept, otherwise each *failed login* event is considered as the first event of a new sequence.

The rule `init_action` is executed first. It initiates the global variable `count` that counts the number of matched sequences as well as the frozen variable `check`. It also triggers the rule `first_failed_login` to start the analysis process and the rule `echo_result` to display the number of detected sequences at completion.

The rule `first_failed_login` searches for all failed attempts to log in by any user. It delays the decision of starting a new sequence by triggering an instance of the `analyze_failed_login` rule for the next ASAX cycle. The latter rule checks the value of the global frozen variable `check`. It its value is different than 0, it means that the previous event has been treated during the previous cycle, and hence it belongs to an existing sequence of failed logins. Otherwise, the previous event starts a new sequence. As a result, an

```
global internal frozen flag : integer;

init_action;
begin
   flag := 0;
   trigger off for_next find_1
end

rule find_1;
begin
   if <the current event is the first event of the signature>
     --> trigger off for_next delayed_find_1(ID)
   fi;
   trigger off for_next find_1
end;

rule delayed_find_1(ID);
if flag = 0
   --> trigger off for_current find_2(ID)
fi;

rule find_i (1 < i <= n) (ID)
if
   <The current event is a Ei> and
     <identifier of the event is ID>
     --> begin
       flag := 1;
       trigger off for_next find_i+1(ID)
     end
fi.
```

Figure 2.9: RUSSEL pattern for non-overlapping signatures

instance of rule `count_failed_logins` is triggered for the current cycle. Its task is to find the remainder of the sequence starting from the current event. The rule is instantiated with two parameters: the ID of the user attempting to log in and the number of remaining attempts before raising an alert. The rule `first_failed_login` also re-triggers itself for the next cycle to search for other failed attempts. It remains thus alive during the whole analysis.

The rule `count_failed_logins` finds all *failed login* events belonging to an existing sequence. It may have more than one instance running; actually one per monitored user. When an instance of the rule finds a failed login by the user it monitors, it assigns the value 1 to the global frozen variable `check` to mark the current event as treated and, then, an alarm is raised if there is no remaining attempt to detect. If some attempts are still needed to be matched, the instance re-triggers itself but with a decremented number of remaining failed logins.

2.5 Implementation of ASAX

In the previous sections, we have presented ASAX at a conceptual level. It rules out a number of details about its implementation. We need these details to understand the concepts on which we develop our optimizations in the following chapters. During analyses, rules are executed by a built-in virtual machine. We provide now a full description of the implementation of that machine. It consists of picturing the internal representation of respectively the data types, the NADF format, the current environment and the compiled form of rules.

2.5.1 Data types

Integers are represented by their respective values. Strings of bytes are modeled by pointers to memory areas having the following form: the data length followed by the data itself. The data length is encoded as an unsigned 16 bits integer. This representation is more general than the null-terminated string representation, hence allowing data containing null bytes to be represented. In the following, we denote the internal representation of objects with the notation:

$$\boxed{\text{ri}(object)}$$

where *object* is the object to represent.

2.5.2 The NADF format

As we said in Section 2.2.3, analyses are performed with respect to events encoded in NADF format. Figure 2.11 describes the NADF record structure. It consists of a header, containing the total length of the NADF record,

```
global internal count: integer;
global internal frozen check: integer;

init_action;
begin
  count := 0;
  trigger off for_next first_failed_login;
  trigger off at_completion echo_result
end;

rule echo_result;
println(count, 'failed attempt(s) to log in found.');

rule first_failed_login;
begin
  if
    EVENT = login and RESULT = failed
    --> trigger off for_next analyze_failed_login(USER_ID)
  fi;
  trigger off for_next first_failed_login
end;

rule analyze_failed_login(user_id: integer);
if check = 0
  --> trigger off for_current count_failed_logins(user_id, 2)
fi;

rule count_failed_logins(user_id: string; tofind: integer);
if
  EVENT = login and RESULT = failed and USER_ID = user_id
  --> begin
    check := 1;
    if
      tofind = 1
      --> begin
        println('3 failed attempts to log in for user ', user_id);
        count := count+1
      end;
      true --> trigger off for_next count_failed_logins(USER_ID,tofind-1)
    fi
  end;
  true --> trigger off for_next count_failed_logins(USER_ID, tofind)
fi.
```

Figure 2.10: RUSSEL module detecting sequences of *3 failed logins* with a *frozen* variable

followed by a sequence of variable-sized fields. Each field is made of an identifier followed by data. The former is an unsigned 16 bits long integer, while the latter is represented by a string of bytes. For optimization reason that we explain later, fields are ordered by identifiers.

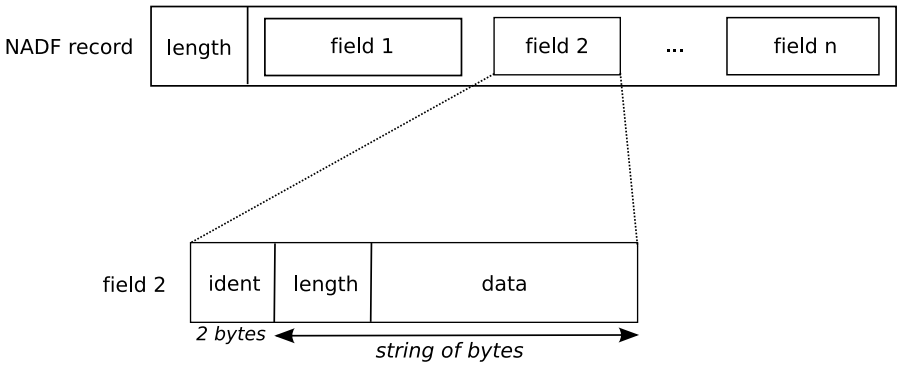


Figure 2.11: Structure of NADF record

2.5.3 Representation of the current environment

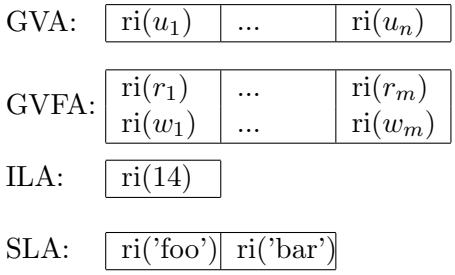
The local environment consists of local variables and actual parameters while global variables, literals and the current record compose the global environment.

Assuming a rule having $p_1, ..., p_n$ as actual parameters and $l_1, ..., l_m$ as local variables, the internal representation of its local environment are the two storage areas LVA (Local Variables Area) and APA (Actual Parameters Area):

APA:	ri(u_1)	...	ri(u_n)
LVA:	ri(v_1)	...	ri(v_m)

where u_i and v_j denote respectively the current values of p_i and l_j, i.e. for the current instance, with $1 \leq i \leq n$ and $1 \leq j \leq m$.

The internal representation of the global environment is similar, except for the current record. To avoid waste of memory allocation, literals are allocated only once and stored inside dedicated storage areas too. However, integer and string of bytes use distinct areas. Consider a RUSSEL module using the regular global variables $g_1, ..., g_n$, the frozen global variables $gf_1, ...gf_m$ and the literals 14, '`foo`' and '`bar`', the internal representation of the global environment are the following four storage areas GVA (Global Variable Area), GFGAVGlobal Frozen Variable Area), ILA (Integer Literal Area) and SLA (String Literal Area):



where u_i, r_j and w_j denote respectively the current value of the regular global variable g_i and the read and write value of the frozen global variable gf_j, with $1 \leq i \leq n$ and $1 \leq j \leq m$.

The input data trail is heavily accessed during the analyses. Hence, the internal representation of the current record must be designed carefully to ensure efficient accesses. Before being processed by the current rules, the current record is copied into a static buffer and pre-processed to update an indirection table whose purpose is to speed up accesses to it. Each entry of the indirection table is composed of two fields: a field identifier and a pointer to the corresponding field inside the current record buffer. At compile time, all rules are scanned for references to record fields. The corresponding field identifiers are stored in the indirection table in ascending order, just like the fields inside NADF records. It allows us to scan both the current record and the table in parallel to update the pointers. In case a field is not present in the current record, a special value `NOT_PRESENT` is assigned to its corresponding pointer in the indirection table. Consider we analyze records composed of four different fields: $field_1, ...field_4$ with the respective identifiers $1, ..., 4$, Figure 2.12 illustrates an example of the indirection table.

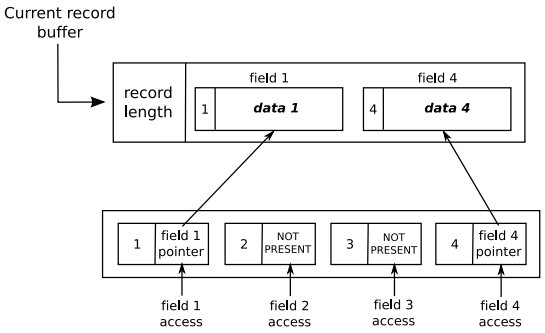


Figure 2.12: Example of indirection table

2.5.4 Internal code

RUSSEL rules are compiled into an internal code that is interpreted at runtime by a virtual machine. The internal code is made of *abstract machine*

instructions, also called *boxes*. The complete procedure to translate rules to boxes is described in [24]. The optimizations we develop in this work manipulate rules at the box level. Consequently, we describe now the box structures to provide all necessary details.

The compiled representation of a RUSSEL rule is a chained structure composed of boxes, actually forming a directed graph[3] where all paths are possible execution paths of the compiled rule. Boxes have either one or two successors. Boxes with one successor implement simple instructions, i.e. producing side effects. Assignment, rule trigger and arithmetic operations are such instructions. Boxes with two successors do not generate any side effects and stand for conditional instructions. One successor is the next box to be executed if the condition is evaluated to true while the other successor is the next box to be executed if the condition is evaluated to false. There are no boxes standing for logical operations. The way the boxes are connected depends on which logical operators (*and*/*or*) are used to build compound conditions. For the operator *not*, successors are simply swapped.

Each box presents the following content:

1. An operator which indicates the action to perform.
2. Addresses to a various number of operands, depending on the operator.
3. Branch addresses to the next box(es). Addresses containing the *nil* value mark the end of a path.

For example, we describe the structure of four distinct boxes:

- **Arithmetic box**: Evaluating arithmetic expressions needs to save intermediate values. As a result, a memory area, called *working area* is statically allocated to store these temporary values. Consider an arithmetic operation v_1 *op* v_2 where v_1 and v_2 are integers, it is represented by the following box:

op	γ_1	γ_2	δ	α

  where *op* is an arithmetic operator among {+,-,*, div, mod}, while γ_1 and γ_2 respectively denote the addresses of the integer values v_1 and v_2. The result of the operation is stored at address δ. α is the pointer to the next box to execute.

- **Assignment box:** Consider the assignment instruction $x := v$, this is represented by the following box:

assign	δ	γ	α

[3]Directed graphs are graphs of which all edges are directed.

where δ and γ respectively denote addresses of the variable x and the value v, and α is the pointer to the next box to execute.

- **Trigger box:** The three types of trigger have their own operators: $for\_current$, for_next and $at\_completion$. Boxes representing trigger actions present the following form:

op	rule_ptr	α

where op is one of the three operators above. The effect of this box consists of creating a new instance of the rule pointed by $rule\_ptr$ and insert it into the right rule set depending on the operator. α is the pointer to the next box to execute.

- **Relational box:** Relational conditions are represented as follows:

op	γ_1	γ_2	α_t	α_f

where op is a relational operator among $\{=, \neq, <, \leq, >, \geq\}$, while γ_1 and γ_2 denote the addresses of the values to compare. The pointer to the next box to execute is α_t in case the relation is true, α_f otherwise.

Figure 2.13 depicts an example of a full translation of a rule to boxes. The rule is a simplification of the rule `count_failed_login` introduced earlier. Instead of searching all sequences of 3 failed logins, it only counts all occurrences of failed logins appearing in the analyzed event flow. For clarity, variables and literals are indicated directly inside the boxes. Note that, as the trigger action is executed whatever the result of the condition is, all paths of the graph lead to the box representing the rule trigger. Moreover, the latter does not have successors, hence the value of its α pointer is nil. In addition, we can see that the assignment box follows the addition box and assigns the value previously stored by the later (at address $addr$) to the variable `count`.

2.5.5 Analysis process

So far, we have presented internal representations of all objects involved in the analysis process. The data stream analysis is ensured by two algorithms: the first algorithm aims at executing rule codes and the second algorithm uses the first one to apply the analysis to all records. We provide now the implementation details of both algorithms. In addition, we discuss the used memory management for the rule instance creations.

```
global internal count: integer;

rule count_failed_login;
begin
  if
    EVENT = login and RESULT = failed
    --> count := count + 1
  fi;
  trigger off for_next count_failed_login
end;
```

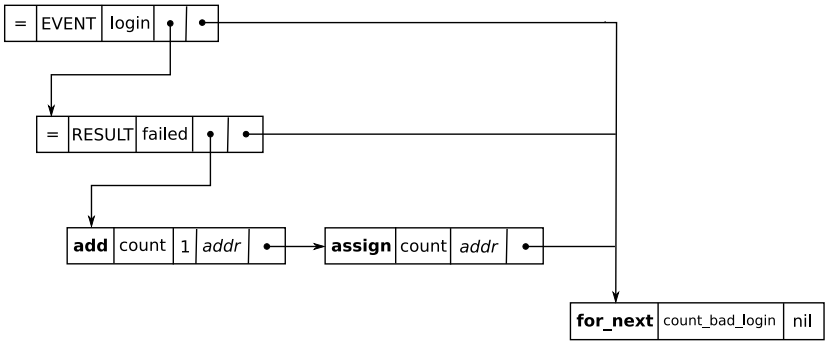


Figure 2.13: Box representation of a rule

Evaluating a rule

Since the internal code of each rule forms a directed graph, we use a graph representation to formalize RUSSEL rules. Rules are directed graphs of which all paths lead to a special node called *end*. The other nodes are either *conditional nodes* or *statement nodes*. Since there is no box for logical conditional instructions (see Section 2.5.4), a conditional node represents a relational or a presence condition. A statement node stands for a simple instruction (e.g., an assignment). Instructions that involve expressions are compiled into several boxes. For example, the internal code of a relational condition is a chain of boxes composed of a relational box (which is depicted in Section 2.5.4) preceded by boxes calculating the values of the tested expressions. In the graph representation, all boxes of a single instruction are regrouped inside a single node. Edges going from conditional nodes are labeled *true* or *false*. Edges going from other nodes are labeled *next*. Figure 2.14 depicts the graph of the rule `count_failed_logins` presented in Figure 2.13.

Algorithm 2.1 presents an abstract evaluation algorithm working on

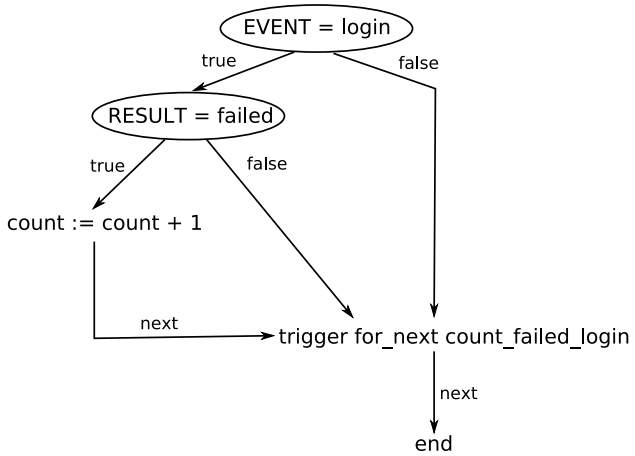


Figure 2.14: Example of a graph of rule

graphs. It uses the following functions: $type(N)$ gives the type of the node N (either conditional, statement or end), $succ(N, l)$ returns the successor of N corresponding to label l, $evaluate\_node(N)$ evaluates the node N with one successor and $evaluate\_condition(N)$ returns *true* if the condition corresponding to N satisfies.

Evaluating the event stream

At each ASAX cycle, the current rules, i.e. the instances of rules contained in the *current set* are executed. Algorithm 2.2 depicts the complete procedure to analyze streams. First, the rule sets are initialized (the current set is filled with all `init_action` rules). Second, the current rules are applied to the current record r until there is no more record left in the stream or there is no more active rule. At each iteration, Algorithm 2.3 is applied to the current record, which is first pre-processed to update the record indirection table. Before applying a rule instance, its actual parameters are copied inside the dedicated storage area APA. Finally, the completion rules are executed to finish the analysis. The function $get\_init\_actions()$ returns the set containing an instance of each declared `init_action` rule. The function $remove\_first(S)$ removes and returns the first element (record) of the stream S. The function $pre\_process(r)$ updates the indirection table with respect to the record r. The function $pick\_one\_instance(s)$ removes and returns an (arbitrary) instance from the set s. The function $param(inst)$ denotes the parameters of the instance $inst$. The function $code(inst)$ denotes the internal code of the rule whose $inst$ is an instance. Finally, the function $kill(inst)$ frees the memory allocated by the instance $inst$.

Algorithm 2.1 evaluate_rule(G, N) - Rule evaluation

Precondition: G is a graph of RUSSEL. N is a node of G.

```
begin
  if type(N) ≠ end then
    if type(N) = conditional then
      begin
        if evaluate_condition(N) = true then
          evaluate_rule(succ(N, true))
        else
          evaluate_rule(succ(N, false))
      end
    else
      begin
        evaluate_node(N)
        evaluate_rule(succ(N, next))
      end
end
```

Memory management of rule instances

Since there is no specified order to apply rules, rule sets can be indifferently modeled by stacks or queues without affecting the execution. However, for the simplicity of the implementation, a set of rules is modeled as a stack and implemented by a linked list. Each cell of the list is a descriptor for a rule instance. The structure of the descriptor is illustrated in Figure 2.15. It contains the following fields: a pointer referring to the internal code of the instantiated rule, a pointer to a variable sized memory area containing the actual parameters of the instance, the size of the parameters area and a pointer to the next instance descriptor within the rule set (rule stack). Notice that all instances of a rule share a unique instantiation of the internal code.

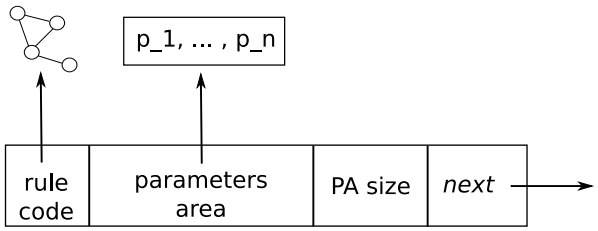


Figure 2.15: Structure of the rule instance descriptor

ASAX efficiency stems from the simplicity of its memory management.

Algorithm 2.2 analyze_stream(S) - Stream analysis

Precondition: S is a list of NADF records.

Postcondition: Let the rule sets: current, next and completion, APA the actual parameters storage area and the records in S: $r_1, ..., r_n$. Either all records have been analyzed or records r_1 to r_i have been analyzed where $1 \leq i \leq n$ and the current set is empty.

```
begin
  current ← get_init_actions()
  next ← {}
  completion ← {}
  while S is not empty and current is not empty do
    begin
      r ← remove_first(S)
      analyze_record(r, current)
      current ← next
      next ← {}
    end
  while completion is not empty do
    begin
      inst ← pick_one_instance(completion)
      evaluate_rule(code(inst))
      kill(inst)
    end
end
```

There is no explicit mechanism to allocate and deallocate rule instances in RUSSEL, and there is no need for a garbage collector either. Each instance descriptor is deallocated once it is executed. A new descriptor is allocated for each trigger action. The price to pay for this simplicity is the fact that new descriptors are also created for rules triggering themselves and, additionally, actual parameters must be copied in the newly created instance descriptor each time a rule is triggered.

2.6 Experimental evaluation of ASAX efficiency

In this section, we carry out experiments to measure ASAX efficiency. The used benchmark illustrates well situations involving many parallel occurrences of signatures. It appears that ASAX suffers a lack of performance of which we identify the sources. Consequently, the results we present now are used as reference point to evaluate our optimizations in the following chapters. Also the used benchmark is referred as the *Darpa benchmark* in the remainder of this thesis.

Algorithm 2.3 analyze_record(R, current) - Record analysis

Precondition:

- R is the current NADF record.
- current is the current rule set.

Postcondition:

- All instances in 'current' have been applied to the record R.

```
begin
  pre_process(R)
  while current is not empty do
    begin
      inst ← pick_one_instance(current)
      APA ← param(inst)
      evaluate_rule(code(inst))
      kill(inst)
    end
end
```

The test platform for our experiments is an AMD Sempron 3000+ with 1Gb ram running at 1.6Ghz. The running OS is Linux Ubuntu 6.10. To measure real analysis times, we first measure the time spent by ASAX to read the files without performing analyses. It includes the time needed to translate the data stream into NADF records as well as the time to realize the pre-processing of each record. The later depends on the number of field references present in the declared rules. We withdraw the reading time from all time measures to get the actual analysis time.

2.6.1 Measuring ASAX with an existing benchmark

To assess the efficiency of ASAX, we needed to choose a benchmark close enough to the reality to validate the generality of our results. This is the reason we have chosen to use a rewriting in RUSSEL of a set of intrusion detection rules of the USTAT IDS [27] and to use the Darpa/Lincoln Laboratory Evaluation Data [38, 37] for the audit data.

Description of the rules of the benchmark

USTAT original rules are described in the STATL language [19] (the formalism used by all components of the STAT tool suite). They can be found at [60]. The main reason to have chosen these rules is that they detect all sorts of classical attacks such as buffer overflows, ftp-write attacks, unau-

thorized accesses. Thus, there are amenable to be used in the context of real IDS deployments.

USTAT provides 18 detection rules based on BSM [61]. The method used to perform the RUSSEL translation is described in 3.4.4. There are 31 resulting rules among which 23 are stateless and always active. The remaining rules are part of stateful scenarios and are activated intermittently. Table 2.1 presents the rules of our benchmark, grouped by module. For each rule we indicate if it is stateful (Y) or not (N). The source code of all rules is provided in Appendix A.2.

The Darpa/Lincoln Laboratory Evaluation Data

For the test data, we use a sample of the Darpa/Lincoln Laboratory Evaluation Data [38, 37]. This data has been created by the Lincoln Laboratory from the Massachusetts Institute of Technology in 1998. This was carried out with the collaboration of Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory. The reason of creating and distributing this data set is that it defines a fixed reference to compare new IDSs. The data set is composed of synthesized audit trails in BSM format (for host-based IDSs) and network traces in `pcap` format (for network-based IDSs).

According to [42], the Darpa Evaluation Data may not reflect reality because both background and attack data have been synthesized and might not represent general behaviors in information systems. This is still a problem nowadays; it is actually impossible to forge data that are general enough to represent the general behavior of all (or at least most) information systems since they present a high heterogeneity of activities and security policy. However, the Darpa/Lincoln Laboratory Evaluation Data is the more standard and more documented existing benchmark to evaluate IDSs at the present day. Besides, novel research articles still use it to make their experiments.

The sample of the Darpa Evaluation Data we use to conduct our experiments is the BSM audit logs from the first week of the training data of 1998 [11]. Table 2.2 shows the size and the number of records for each file, as well as the number of raised alerts (which is identical for USTAT and ASAX).

2.6.2 Linearity in the number of instances

To assess the performance of ASAX, we first measure the total evaluation time for the Darpa benchmark, excluding the reading time. Table 2.3 illustrates that the analysis time is not necessarily proportional to number of analyzed records. Indeed, the *monday* file provides a higher time than the *friday* file while the later contains more records. Also, though the *wednesday* file holds only 1.5 times more records that the *monday* one, analyzing the

Rule	Module	Stateful
create	restricted_dir_write	N
mod_perm	mail_spool_hack	N
exec	execute_exit	N
exit		N
mod_perm	enable_suid_sgid	N
write	non_owner_write	N
write	restricted_write	N
read	restricted_read	N
access1	secret_unauth	N
access2		N
access3		N
write	system_program_write	N
exec	suid_shell_script	N
delete	unauth_delete	N
access1	secret_remote	N
access2		N
access3		N
exec1	buf_overflow	N
exec	lpd_delete	N
read		Y
delete		Y
hardlink	sh_minus_hack	N
exec		Y
exec1	buf_to_root_shell	N
exec2		Y
create_file	ftp_write	N
login		Y
read_rhosts		Y
exec1	suid_shell	N
read		Y
exec2		Y

Table 2.1: The rules composing the benchmark

Log	Size (Mb)	Number of records	Number of raised alerts
monday	88.5	744,087	579
thuesday	92	778,784	459
wednesday	130.9	1,094,938	1,230
thursday	91.5	767,929	1,084
friday	99	819,474	832

Table 2.2: Description of analyzed logs

first takes twice the time measured for the second. It is easily explained. The analysis time is not constant from one record to another. Depending on the content of the analyzed stream, a consequent number of instances of rules might be triggered, which potentially makes the evaluation process heavier. This number varies while the analysis goes through the records. It also varies from one file to another. To prove our statement, we made a second experiment which features refined measures.

Log	AT	#records	AT/#records
monday	55.765	744,087	0.075
thuesday	44.091	778,784	0.057
wednesday	109.313	1,094,938	0.099
thursday	55.477	767,929	0.072
friday	52.849	819,474	0.064

Table 2.3: Analysis times

For the second experiment, we measure evaluation times with a rule set in which we progressively add modules from the benchmark. The results are presented in Table 2.4. The column MODULES specifies the names of the modules, which are added in the order they are presented. The column NR depicts the number of declared rules so far, among which the number of stateful ones is provided by the column NSR. The column ANI presents the average number of running instances. Finally, the column ET gives the evaluation time. Note that this experiment is only realized on the *monday* file. As illustrated by Figures 2.16 and 2.17, the evaluation time is not linear neither in the number of modules nor in the number of rules. Actually it is, until we add the module `buf_to_root_shell` to the applied rule set. Though it only contains two rules, this module alone increases the analysis time by about 48 seconds. It has to be noticed that the average number of running instances grew a lot, i.e. from 21 to 88. Figure 2.18 depicts the evaluation time with respect to the average number of instances. We can see that the progression is linear. The little variations we can notice on the curve are

due to complexity differences from one rule to another. Adding the modules to the rule set in a different order should shift these variations.

MODULE	NR	NSF	ANI	ET
unix_restricted_dir_write	1	0	1	0.16
unix_mail_spool_hack	2	0	2	0.418
unix_execute_exit	4	0	4	0.963
unix_enable_suid_sgid	5	0	5	1.182
unix_non_owner_write	6	0	6	1.534
unix_restricted_write	7	0	7	1.725
unix_restricted_read	8	0	8	2.375
unix_secret_unauth	11	0	11	3.567
unix_system_program_write	12	0	12	3.922
unix_suid_shell_script	13	0	13	4.237
unix_unauth_delete	14	0	14	4.459
unix_secret_remote	17	0	17	6.288
unix_buf_overflow	18	0	18	6.316
unix_lpd_delete	21	2	20	7.825
unix_sh_minus_hack	23	3	21	7.96
unix_buf_to_root_shell	25	4	88	55.415
unix_ftp_write	28	6	89	55.719
unix_suid_shell	31	8	90	55.765

Table 2.4: Table of results

2.6.3 Sources of inefficiency

Our experiments allow us to identify two sources of inefficiency, which make the analysis not as efficient as it should be.

The first source of inefficiency is the fact that the trigger action is very costly. However, this is a low price to pay when the number of rule instances is small but efficiency problems arise when many rule instances are running. In fact, in such situations, it is often the case that most of the rules do nothing except re-triggering themselves for the next record because each rule instance searches for a specific record. Hence, very few rules really "do something". In that situation, it is reasonable to assume that the condition testing time is negligible in comparison to the time taken by the rule instance memory management. Consequently, the execution time is mostly equal to the time spent to deallocate memory for the rule instances that have been executed, to allocate memory for the new rule instances, and to copy the parameters. Moreover, It is also reasonable to consider that the time spent to trigger a rule is constant from one rule to another (if we assume that the number of parameters does not vary a lot). As a result, it explains the fact

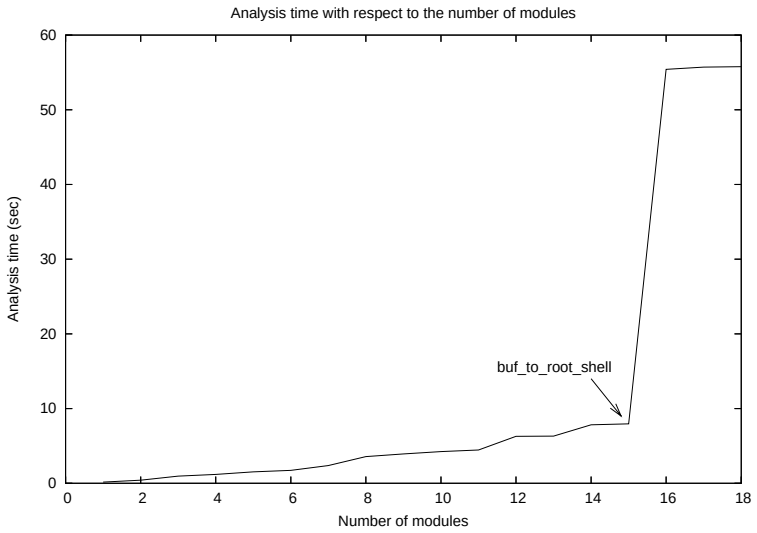


Figure 2.16: Analysis time with respect to the number of modules

that the execution time is linear in the number of rule instances.

The second source of inefficiency is the fact that results of condition evaluations are not cached in memory. Consequently, whenever a rule has more than one instance, it happens that the execution of the instances evaluates the same conditions several times. Moreover, the problem is extended to the rule level when rules contain identical conditions. Obviously, redundant evaluations result in a significant waste of resources.

2.7 Conclusion

In this chapter, we have presented a detailed overview of ASAX and its implementation. Thanks to its universality and its built-in complete language RUSSEL, the programmer is able to specify and craft very powerful analysis programs for any type of data source.

Experiments proved us that the analysis process is not as efficient as it should be when many instances of rules are involved. We identified two problems. First, most of the analysis time is spent to allocate and deallocate instance descriptors while it is not necessary for rules triggering themselves. Second, it occurs that conditions are evaluated many times because the results are not cached. The impact of both problems is sensible to the number of instances. Thus, it is very important to write carefully the rules to avoid any explosion of it.

In the second part of this work, we propose, design and implement optimization techniques for the language RUSSEL to overcome the inefficiency

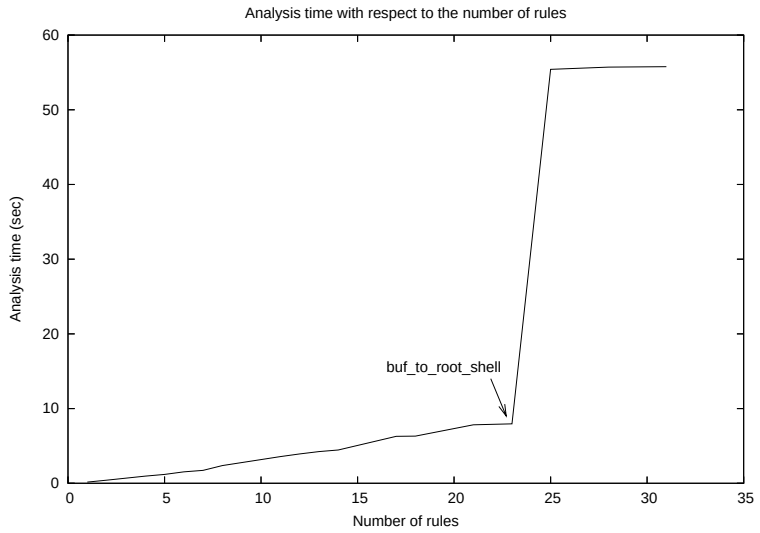


Figure 2.17: Analysis time with respect to the number of rules

issues. The key idea is to detect rule instances that are irrelevant to the analysis of the current event. Those rule instances do not have to be executed on the current event. They can be either discarded or re-triggered for the next event. Moreover, descriptors of re-triggered instances should be preserved and moved to the next set. We also achieve to factorize conditions to avoid redundant evaluations.

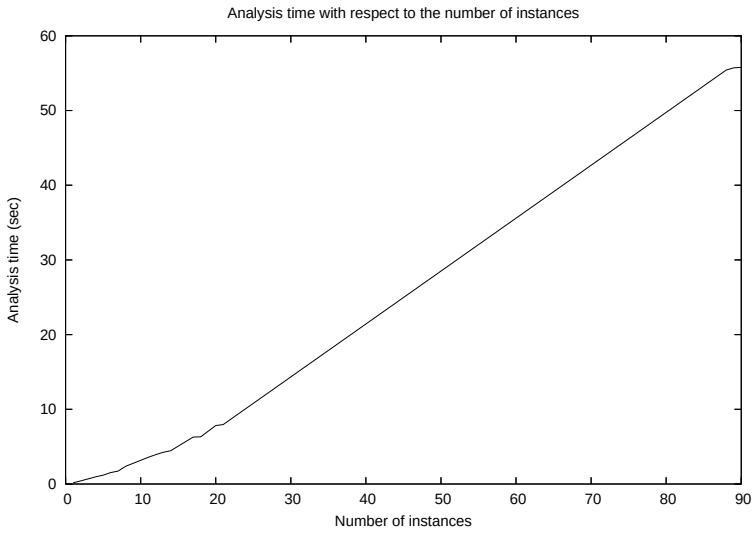


Figure 2.18: Analysis time with respect to the number of instances

Chapter 3

Related work

Before embarking in the presentation of our own work, we present, in this chapter, an account of several research areas that are, actually or seemingly, related to what we have explored and/or achieved. Although –and because– almost nothing has been said about our contribution at this point, we put forward some references to the sequel of this thesis, to motivate what will be presented in the core of the text.

Our presentation of the related work consists of five parts.

1. The optimization techniques developed in this thesis make use of several particular kinds of decision diagrams. We introduce the main kinds of decision diagrams known from the literature and we explain what they are used for, in model-checking, artificial intelligence, compilation, and intrusion detection. We discuss several issues related to the synthesis and the optimisation of such decision diagrams. As a preview to our own presentation, we explain why decision diagrams as described in the literature are not readily usable in our context.

2. In recent years, efficient intrusion detection has become an important research topic because of the growing number of attacks invading the Internet. We present existing work aiming at optimize intrusion detection systems and we discuss their generality to clarify the applicability of our own proposal.

3. Signature-based intrusion detection and, more generally, parallel multiple analysis of data (or event) streams is a specialized application area in which defining and using specialized programming languages is appealing. We present and compare several proposals of such languages, mainly to make it clear if they can benefit from our work, and to what extend.

4. The optimization techniques presented in this work have been implemented inside ASAX, a particular system for parallel multiple analysis

of data streams. To assess the techniques, we notably use a specific benchmark borrowed from STAT, a well-known signature-based intrusion detection system. Therefore, we give a more complete description of STAT and we explain how STAT signatures from the benchmark have been translated to RUSSEL, the ASAX language.

5. Finally, we provide a detailed overview of Bro, a major intrusion detection system, which is probably considered as the state-of-the-art for stateful intrusion detection by researchers in the field. Some design choices of Bro are significantly different from the choices made by STAT or ASAX. We discuss in the conclusion of this thesis whether our optimization techniques are applicable to Bro or, possibly, to a system providing the same high level functionalities but based on a different architecture.

3.1 Decision trees and diagrams

3.1.1 Decision trees and diagrams, in general

An early survey on (binary) decision trees and diagrams is proposed in [45]. This paper discusses various optimality questions related to the building of a decision diagram: building a smallest diagram, finding a fastest algorithm to build the decision diagram, minimizing memory usage,... We use some of the techniques described in this reference, mainly caching techniques, but not all of them. Caching techniques amounts to uniquely represent each subtree of a global decision tree in core memory. Isomorphic subtrees are thus represented only once so that subtree equality can be implemented as pointer equality. This allows one to reduce memory usage dramatically in many practical applications, as ours, without increasing the time complexity of algorithms.

However, classical binary decision diagrams are not directly usable in our context because, usually, each evaluation of a decision diagram, i.e., each evaluation corresponding to a specific data context, lead to a unique decision. The decision can be just "yes" or "no", or it can be the decision of executing a specific set of actions. In the first case, the decision diagrams have only two leaves (corresponding to "yes" or "no"). In the second case, there are as many leaves as possible sets of actions to execute. If the number of these sets is high, the size of the diagrams may become huge and the process of building the tree may spend too much time. In our application context, we build decision diagrams to choose one among three decisions for n different rules. The number of leaves is thus potentially equal to 3^n.

To minimize the size of diagrams and to accelerate the merging of diagrams, we have introduced a novel data structure called *sequential tree*. Sequential trees are quite different and more complex than usual decision

diagrams: on the one hand, independent subtrees, i.e., subtrees containing no common conditions or decisions are factorized; on the other hand, individual decisions are not always put at the bottom of trees, they can be put anywhere. Algorithms for handling sequential trees are more complex than for normal decision trees. Sequential trees are presented and evaluated in Chapter 6.

3.1.2 Boolean decision diagrams (BDDs)

In recent years, boolean decision diagrams, known as BDDs, have been extensively used in many research areas, notably for model-checking (see [8, 7, 5]). The kind of applications they are used for is rather different than ours: in model-checking, BDDs are used to perform set operations on large sets of states. They are not used to be actually evaluated and, if they would, they should only produce a binary decision not sets of decisions. The power of BDDs in model-checking stems from the fact that they can represent very large sets of states in a compact way. States are defined as boolean functions over a set of variables and BDDs are ordered on the variables. Ordering BDDs on the variables makes it possible to define a canonical form for BDDs, which facilitates the implementation of set operations on BDDs.

Our work reuses the idea of ordering decision trees but, in our case, the nodes of decision trees contain elementary conditions (i.e., relational conditions about integer or string values and variables), not single boolean variables as in BDDs. We thus have to define an order on elementary conditions. Although BDDs often are compact representations of large sets, their size is exponential in the number of boolean variables, in the worst case. Several authors have pointed out that the size of BDDs is very dependent of the particular choice of an order for the variables, and techniques and heuristics to choose a good order and to modify the order dynamically have been proposed [23, 55]. We also study the problem of choosing a good ordering in our application context in Chapter 6. The techniques developed for BDDs do not seem to be very useful in our context because the data structures we use (sequential trees) are rather differen t and more complex. In fact, with sequential trees the size of trees is reduced by other means, i.e., factorization of subtrees, so that choosing a good ordering is less important. Moreover it is unlikely that similar size reduction can be achieved only by changing the order on conditions.

3.1.3 Decision trees for classification

Non binary decision trees are used for the problem of classifying objects on the basis of a number of "features". Fast algorithms to synthesize decision trees that are as small as possible have been proposed (see, e.g., [52, 63, 59]).

A key point of these methods is that they postulate that the set of all objects must be divided to form a partition, i.e., two different leaves of the decision tree are disjoint sets. In our case, the same rule can be executed together with different other rules in different situations. Thus, the method must be adapted. Such an adaptation is proposed in [32, 33] for optimizing Snort. However, the authors note that with their adapted method the size of the decision diagram becomes quickly unmanageable. So they propose to decompose the global decision diagram into smaller ones that are evaluated sequentially. This is related to our notion of sequential trees, but our approach is much more general and innovative since sequentialization is applied at each level of the global tree. Moreover, we do not "induce" decision trees on the basis of training data as in the works cited above: We first extract a local decision tree from each possible rule by static analysis (see Chapter 5); then, we reorder the local trees according to some unique global order on conditions; finally, we merge the local trees to get a global sequential tree (see Chapter 6). Induction on the basis of training data is a particular case of merging local trees where local trees have a unique single branch leading to a positive conclusion and all conditions of the tree lead either to the next condition on the branch or to a negative conclusion. In [32, 33], each Snort rule defines a single training data because Snort only use conjunctions of elementary conditions. Thus, we solve a more general problems since we consider rules with arbitrary boolean combinations of elementary conditions (and side effects).

3.1.4 Decision trees in compilation

Decision trees are also used in compilers, especially for declarative languages such as Prolog and Constraint languages [13, 29, 30, 31]. These works are more elaborate than the methods proposed for Snort in [32, 33] but they also fundamentally reuse the ideas of [52, 63] (induction of decision trees) and they face the same difficulties, i.e., the dilemma between partitioning decisions or getting a size explosion of the decision tree.

3.1.5 Finding a good order for conditions in decision trees and diagrams

Finding a good order for conditions in decision trees and diagrams is a key issue. Depending on the application at hand, it may be important to save memory space, to build decision trees quickly, and to evaluate decision trees faster. BDDs use specific methods [23, 55]. Most other works [52, 63, 32, 33, 13] use an order based on an entropy function borrowed from information theory. The problem of finding a good order on conditions is also related to the problem of making probabilistic inferences in an inference net as in PROSPECTOR [26]. In that context, probabilities are dynamically

computed when traversing a network of inference rules where conditions are a priori given a "subjective" probability. Probabilities are then modified dynamically, depending on the path followed in the network, using Bayesian methods [18]. Our method to compute orders on conditions is related to this later approach: we attach subjective (or even arbitrary) probabilities to be true or false to elementary conditions; then, we use simple probability theory formulas to give a weight to every condition in the context of evaluating all rules of a given pre-defined set. We explore several heuristics corresponding to different goals such as minimizing the size of the global decision tree or minimizing the number of useless condition evaluations or making a compromise between the two (see Chapter 6).

3.2 Efficient algorithms for intrusion detection

The fundamental problem we face in this thesis is the problem of having a large number of "rules" to apply to every item of a sequential data stream (or event stream, or audit trail...). We want to speed up the process by selecting "relevant" rules only, at a cost as cheap as possible. This kind of problem has been considered by several authors in similar or rather different contexts. We now summarize some of these works and we relate them to our own work.

3.2.1 OPS5, P-Best and the Rete algorithm

OPS5[21] is an artificial intelligence language based on the production rule paradigm. Production rules are made of a set of conditions applicable to a data base of facts, which are similar to Prolog terms. If all conditions of a rule match facts of the data base, some actions (i.e., the "conclusion" of the rule) are taken. Actions mainly consist of adding or removing facts from the data base. The program stops when no more rules are applicable to the current data base. Otherwise production rules are repeatedly applied to the facts in the data base, in arbitrary order.

It is easily seen that a naive implementation of such a language is inefficient: since few facts are added or removed from the data base at once, all rules are matched very often to the same facts leading to many redundant computations. The main way to improve on the naive implementation is to try to perform every test only once and to avoid redundant tests. The RETE algorithm [22] has been specifically designed to avoid such redundant computations. It allows a rule to be computed incrementally: when the fact base is modified only the inferences using removed facts are remade; moreover partial, already made, inferences are further completed taking into account only the new facts. The RETE algorithm is used in systems for intrusion detection based on production rules such as P-Best [36]. P-Best in fact uses an adaptation of the Rete algorithm to sequential data stream

analysis: at each cycle, a new fact corresponding to the next data item to process is added to the fact data base; then, production rules are applied; when no more production rules are applicable the fact describing the data item is removed from the data base and a new cycle start.

The techniques developed in this thesis are not intended to improve on the general principle of the RETE algorithm, which is to build a data structure, called the RETE network, that describe all partial inferences and how to extend them. However, we propose techniques to detect rules that are relevant without "trying them". These techniques could be used inside P-Best to find more quickly which part of the RETE network must be considered when starting a new cycle. They could also possibly be extended to optimize the choice of the next part of the network to consider but this needs further investigation.

3.2.2 Snort optimizations

Snort [53] is a popular signature-based intrusion detection system. It is basically a stateless system, which means that each detection rule applies to the current event (i.e., network packet) only, without taking previous events into account. Optimisations for Snort, based on decision diagrams have been proposed in [32, 33]. These techniques are less general than what is proposed in this thesis because Snort rules only use conjunctions of simple conditions and have no parameters. Hence techniques based on induction of decision trees are applicable. They must be adapted however since non disjoint sets of rules may be applicable to different incoming packets. We address situations where rules contain arbitrary boolean combinations of elementary conditions. Moreover, the set of active rules may evolve dynamically.

The optimizations proposed in [32, 33] also address a problem rather specific to stateless signature-based intrusion detection, i.e., optimizing the matching of a large number of regular expressions against the same input string. In our framework, matching regular expressions to a string are just "normal" elementary conditions. We do not focus on their global optimization since it is an orthogonal problem. It is not difficult to integrate such techniques and ours in the same implemented system.

3.2.3 Optimizations of intrusion detection languages based on Petri nets

Optimization techniques addressing problems specific to stateful intrusion detection are proposed in [43]. The authors use a simple language based on Petri nets to define stateful signatures. Their optimizations are quite related to what we do. In particular, they avoid multiple evaluations of conditions, notably by using indexing for (their equivalent to our) parametric conditions. They propose an additional technique, which has currently no equiv-

alent in our implementation: unique evaluation of common subexpressions. This technique can be easily added to our system. An important difference however between ASAX and their language is the fact that RUSSEL is a complete language with side-effects. Thus not all common sub-expressions can be evaluated only once since the same variable can have different values at different program points. Static analyses techniques can be used to overcome the problem whenever possible.

3.2.4 Conclusion

To summarize on these aspects, our work proposes techniques that compare well with recent work proposed in the literature. Moreover, we show in the sequel that our techniques are applicable to programming languages more general than those that are usually used for signature-based intrusion detection. For example, RUSSEL, the language that we use as a test-bed for implementing our techniques, can be used to program powerful (stateful) honeytanks and, in that context, our optimization allows the honeytanks to be much more efficient since each ASAX cycle can be performed in bounded time for any number of active instances [65]. The systems discussed above are specialized for intrusion detection and we think they could hardly be used for programming honeytanks.

3.3 Application-specific formalisms for intrusion detection and related higher-level languages

In this section, we present some formalisms or programming languages that have been proposed specifically for making signature-based intrusion detection more convenient. We also compare them with ASAX and its programming language RUSSEL. Our goal is twofold: firstly, we want to make it clear to which extent the work proposed is this thesis is applicable to other approaches based on different programming paradigms. Secondly, we want to discuss the convenience of Russel as compared to these other proposals.

Production rule approach

Some intrusion detection languages such as P-Best [36] are designed according to the so-called *production rule* approach, i.e., they are related to the artificial intelligence language OPS5. At first glance, production rules look similar to `if fi` rules in RUSSEL but this is misleading. Conditions in a production rule may match several different facts of the fact base. It is possible to add facts to the fact base to keep track of events previously met. Thus a P-Best production rule may rather straightforwardly relate several events in the trace. In RUSSEL, a condition is just a boolean combination of simple tests on the fields of the current record, the parameters of the

rule, and local and global variables. In theory, rule parameters can be used to keep the relevant information about the past and hence to achieve the same effect as production rules. But it is done in a much less direct and declarative way. The RUSSEL paradigm can be seen as purely incremental and not declarative at all: each time a record is analyzed we extract from it the minimal information needed to continue the analysis and we pass it to future rules. Thus, one can see RUSSEL rules as the result of compiling production rules by hand to a RETE network[22]. This is in fact the seminal idea of ASAX. As a consequence, the optimizations presented is our work are useful for languages based on production rule only because they can improve the efficiency of the RETE algorithm itself.

Nevertheless, using a lower-level language may have benefits: production rules can relate events in a declarative way but very often one has to relate more complicated things than events, such as scenarios (i.e., sequences of events). In this case, a classical parallel programming approach may be more convenient. In a honeytank, for instance, one has to check if a TCP session is already open to avoid opening a new one with exactly the same parameters (see [65]). This can be done very efficiently with lower-level programming techniques in RUSSEL (see again [65]). Similar effects can be obtained with production rules by adding new facts representing scenario instances, but in a much less efficient way.

Declarative languages to specify attack scenarios

Higher-level application-specific languages for specifying attack scenarios have been proposed recently [51, 50, 44]. These formalisms use various operators to characterize a class of event sequences to be searched in a stream of events. This allows one to specify sets of event sequences at a high level, in a declarative way.

A main difficulty, when defining such higher level declarative languages, is to give them a semantics that is, at the same time, clear and natural, applicable to practical situations, and efficiently implementable. As an example, we can site the Sutekh language proposed in [51, 50]. The authors define (the semantics of) a signature as a pair of integers corresponding to the position of two elements in an event stream. Then, they give the semantics of the language based on this notion. This requires to make a few arbitrary choices. For instance, when defining the **if_not** operator, they state that the negative part of the signature must start strictly after the first element of the positive part and that it must stops strictly before the last element of it. One can easily imagine situations where another semantics would be more appropriate. Nevertheless, translating Sutekh efficiently to a more traditional programming language is not trivial as observed by the authors [50].

Another similar formalism is ADeLe [44]. Although similar to Sutekh at

first glance, it has in fact a much simpler semantics since only conditions on single events can be combined. There is no notion of signature as defined by a pair of positions. ADeLe is thus much easier to implement and it has actually been implemented (see [62]).

ASAX and RUSSEL have been used as target systems and languages to implement such higher-level languages [50, 9]. These experiments have shown that it is difficult to get the same efficiency with a translation to ASAX of a higher-level formalism than by writing RUSSEL rules right from the start. Nevertheless, the novel optimization techniques we propose in this thesis could be beneficial to an implementation of a higher-level language with ASAX.

Another approach to defining higher-level formalisms for stateful intrusion detection, based on temporal logic, has been proposed in [54]. The authors of this related work acknowledge that they have been directly inspired by the presentation of ASAX given in Abdelaziz Mounji's PhD thesis [46]. They use the same computation model as ASAX but they propose a higher level language to describe signatures. Notice that their recent implementation uses SWI-Prolog [71] (with cuts) as the target language. It would be interesting to achieve another implementation towards ASAX and to compare the efficiency of the two. An implementation to ASAX of a temporal logic based specification language, part of the KAOS [3] methodology for requirement engineering has already been proposed in [6] showing promising results for obstacle analysis of requirement specifications.

3.4 STAT: State Transition Analysis Technique

STAT [27, 68] is a method that models an attack scenario as a sequence of actions that brings a system from a healthy initial state to a final compromised state of a system. Scenarios are represented by a composition of states and transitions, which allows them to be graphically represented by *state transition diagrams*. A state is a snapshot of the environment of the system. A transition stands for an action specifying the event that makes the scenario move to a new state. The initial state of the diagram corresponds to the state of the system just before the beginning of the attack while the final state is the targeted compromised state of the system. The language STATL [19] proposes an operational semantics to represent the evolution of every instance of a scenario when several occurrences of the same attack occur at the same time.

The STAT model can be implemented in the form of a core, but this core alone is useless. It must be instantiated for a specific context by linking application-specific modules to the core. A number of modules have been implemented to form various IDSs [70]. One can cite for example USTAT, WinSTAT, NetSTAT, WebSTAT and LinSTAT. USTAT, WinSTAT

and LinSTAT are host-based intrusion detection systems. They respectively analyze Sun BSM audit events, Windows NT audit events and Linux syslog files. NetSTAT analyses network traffic while WebSTAT is an IDS designed specifically for web servers [69].

In the following of this section, we make a brief description of the STATL language and the architecture of the STAT runtime. We also make a comparison with ASAX. To conclude, we explain the method to apply to translate STATL scenarios to RUSSEL rules.

3.4.1 The STATL language

STATL is a text-based language and provides constructs to represent state transition diagrams. A STATL specification is a representation of a complete scenario. The execution of a STATL scenario generates a *scenario prototype* that contains data structures for the definition of the scenario and for the global environment, and a set of *scenario instances* where each instance stands for an attack in progress.

A STATL scenario is identified by a name. It can use global variables and local variables. Global variables are shared by all instances of the scenario while local variables are instantiated in each instance of the scenario (data structures for local variables are located inside *scenario instances*. A STATL scenario also defines the states and transitions representing the attack.

A state is identified by its name and has an assertion and a code block. Both are optional. A state assertion is a C-logical expression and it is tested before entry to the state. A state code block is a C compound statement composed of local declarations followed by statements and it is executed after entry into the state.

A transition is identified by its name and has a pair of state names, a type, an action, an assertion and a code block. The pair of state names identifies the source and the target states of the transition. The type specifies the evolution of a scenario instance when the transition is fired. There are three types of transition:

- Firing a *consuming transition* moves a particular scenario instance to another state. It makes the source state of the transition invalid, hence preventing any other occurrences of the attack to spawn from it.

- Firing a *non consuming transition* makes the source state remain valid. It spawns a new instance of the scenario and moves it to the target state. The original instance remains in the source state.

- Firing an *unwinding transition* makes the scenario instance roll-back to a previous state. It implies that all steps that led the unwinding instance from the target state to the source state are undoed. Thus, all instances that were created by the sequence of events composing these steps are discarded.

The action specifies the type of event that may cause the firing of the transition. The assertion and the code block are similar to their corresponding elements in states. In addition, assertions and code blocks of transitions have access to the attributes of the event defined in the action.

Example of a STATL scenario: the *ftp-write* attack

Figure 3.1 presents the state transition diagram specifying the well- known *ftp-write* attack. Solid arcs with a single arrowhead denote non consuming

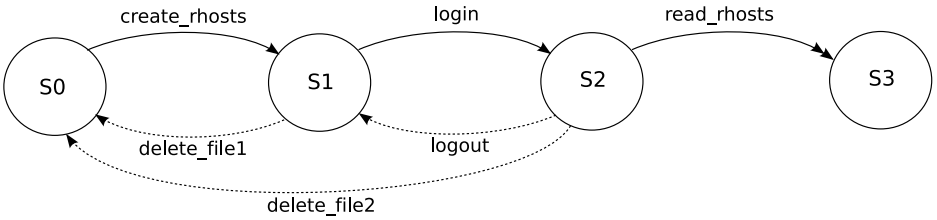


Figure 3.1: ftp-write state transition diagram

transitions while solid arcs with a double arrowhead stand for consuming transitions. Dashed arcs denote unwinding transitions.

The attack exploits the misconfiguration of some FTP service, allowing the attacker to remotely create a malicious `.rhosts` file in the FTP home directory. Afterwards, the attacker uses the file he created to open a remote session with the *rlogin* program without supplying a password. The attacker thus gain an access on a remote host with FTP privileges, which are usually superuser's if the service has been misconfigured. The event sequence to detect is the creation of the `.rhosts` file by a non-root user who do not own the directory where the file is written (transition *create_rhosts*), followed by a login procedure (transition *login*) reading the malicious file (transition *read*). In case the session terminates, it cannot read any file, hence rolling back the automaton to state S_1(transition *logout*). Similarly, whenever the file is deleted, there is no reason to continue tracking the attacker and the detection rolls back to the initial state S_0 (t ransition *delete*).

The corresponding STATL scenario is provided in Figure 3.2.

3.4.2 Runtime architecture

Intrusion detection systems based on the STAT model share a common runtime architecture (see Figure 3.3). The main component of this architecture is the *STAT core*. The core contains the domain-independent mechanisms used at runtime to match attack scenarios against the event stream. The core alone is useless to achieve the scenarios execution. It must be extended with *application-specific modules* to perform in a specific context.

```
scenario ftp_write
{
  int user;
  int pid;
  int inode;

  initial state init {}

  transition create_rhosts (init -> s1) nonconsuming
  {
    [WRITE w] : (w.euid != 0) && (w.owner != w.ruid)
      {
        inode = w.inode;
      }
  }

  state s1 {}

  transition login (s1 -> s2) nonconsuming
  {
    [EXEC e] : match_name(e.objname, "login")
      {
        user = e.ruid;
        pid = e.pid;
      }
  }

  state s2 {}

  transition read_rhosts (s2 -> s3) consuming
  {
    [READ r] :  (r.pid == pid) && (r.inode == inode)
  }

  state s3
  {
    {
      String username;
      userid2name(user, username);
      stat_log("ftp-write: remote user %s gained local access", username);
    }
  }

  transition delete_file1 (s1 -> s0) unwinding
  {
    [DELETE d]: d.inode == inode
  }

  transition delete_file2 (s2 -> s0) unwinding
  {
    [DELETE d]: d.inode == inode
  }

  transition logout (s2 -> s1) unwinding
  {
    [EXIT e]: e.pid == pid
  }
}
```

Figure 3.2: A STATL *ftp-write* scenario

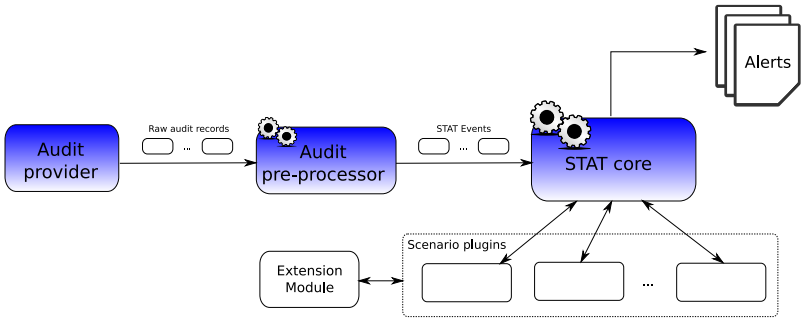


Figure 3.3: Runtime architecture of a STAT-based IDS

At one side, the core needs modules that deliver it events in an understandable format. This is achieved by an *audit provider* and an *audit pre-processor*. The audit provider supplies raw event records to the IDS. For example, it could be a module that accesses to the logging system of a host or a sniffer that dumps the network traffic. Raw event records are sent to the audit pre-processor which converts the supplied raw events into *STAT events*, the internal format handled by the *STAT core*. An implementation of the audit pre-processor converts events of a specific format, e.g., Sun BSM, linux syslog events or network packets. The audit pre-processor is thus chosen according to the used audit provider. The combination of the audit pre-processor with the audit provider is the equivalent of the *format adapter* in ASAX's architecture. Produced STAT events are sent in the form of a stream to the STAT core.

At the other side, the core needs attack scenario definitions to run analyses against the event stream. Attack scenarios are first defined in STATL. Then, they are compiled into C++ files and, finally, the C++ files are compiled into dynamic libraries objects called *scenario plugins*. The produced plugins are the executable representation of the scenarios and are linked to the core at runtime. In addition, the scenario plugins use types and functions provided by an application-specific *extension module*. These types and functions allows the scenarios to access and test the application-specific characteristics of the events. For example, in Figure 3.2, the function `userid2name` is a unix-specific function that returns the name of user corresponding to a given user ID.

We give now a deeper description of the two most important components: the audit pre-processor and the STAT core.

Audit pre-processor

The audit pre-processor filters out uninteresting audit records in the input audit stream[1]. Every record that passes through the filter is converted into a C++ object. The value of each field of the record is copied to a corresponding field in the C++ object[2]. Then, the produced C++ object is encapsulated inside a *STAT event*. A STAT event is defined by a C structure containing a reference to the event object and an integer identifying the type of the event.

Produced STAT events form a stream which is analyzed by the next component, i.e. the STAT core.

STAT core

The core contains the domain-independent mechanisms used at runtime to match attack scenarios against the event stream.

Initialization. At run-time, the core loads a number of scenario plugins whose list is defined in an XML file loaded at the startup of the IDS. When a scenario plugin is loaded into the core, a scenario prototype is created. A *scenario prototype* is a data structure that contains the representation of the corresponding scenario and the functions whose purpose is to verify assertions and to execute code blocks. The action types, i.e., the event types associated to the transitions of the scenario are collected and encapsulated into *event specs* and are inserted into an event spec database. When all plugins are loaded, the event spec database contains all event types that are relevant for the scenarios currently used.

During analyses, scenario prototypes are instantiated into distinct *scenario instances* to keep track of parallel occurrences of the same scenario. A scenario instance is characterized by its current state (in the scenario), a set of *event subscriptions* and its local environment. An event subscription defines the scenario instance interested in a certain type of event and the transition that has to be fired if the the associated assertion and the assertion of the destination state are verified. Subscriptions are stored inside event specs.

A *root instance* is initialized for each scenario at loading time. This instance is in the initial state of the scenario. Instances are organized with a parent-child relation. When a new instance spawns from an existing one (i.e., when a non-consuming transition is fired), the new instance becomes the child of originator instance. Linking instances with such a relation allows

[1] A list of relevant types of record is hardcoded in the pre-processor source code

[2] Fields in C++ objects are statically defined for all possible fields defined by the audit format

the core to easily retrieve the instances to discard in case of unwinding transitions.

Scenarios execution. Once all scenario plugins are loaded, the core starts to analyze the event stream. When a STAT event is treated by the core, the latter retrieves the transition which are interested by the event by consulting the event spec database. For each retrieved transition, the order of evaluation of assertions and execution of code blocks is defined as follows. First the assertion of the transition is evaluated. If it is true, the assertion of the target state is evaluated. If it also satisfies, then the transition is inserted in the set of transitions to fire.

Once all retrieved subscriptions have been treated, the collected transition are treated one by one in the following order: first the non-consuming transitions, then the consuming transitions and finally the unwinding transitions.

1. **Non-consuming transitions.** For each fired transitions, a new instance of the associated scenario is created and its current state is set to the destination state of the transition. The original instance becomes the parent of the newly created one and the environment of the parent is copied to its new child. Then, the code blocks of the transition and the destination state are executed in the context of the new instance. If the destination state is a final state, the instance is discarded. Otherwise, for each outgoing transition of the destination state, a subscription for the associated event type is inserted in the corresponding event spec in the database.

2. **Consuming transitions.** Then, the consuming transitions are fired. It is very similar to non-consuming transitions except that there is no creation of a new instance since the current instance changes state actually. Thus, previous subscriptions for the instance are cancelled and new subscriptions for transitions outgoing from the destination state are inserted in the event spec database. Then, as for non-consuming transitions, the code blocks of the transition and the destination state are executed.

3. **Unwinding transitions.** Firing unwinding transitions undoes the steps that lead the associated scenario instance to its current state. In other words, assuming an unwinding transition from state S_x to state S_y, all instances that were created between states S_x and S_y are discarded and their subscriptions are removed from the event spec database. These instances are retrieved by traversing the parent-child chain from the current until an instance in state S_y is found. Then the instance subtree rooted by the last visited instance is discarded.

The mechanisms stated above can be illustrated with the help of the STATL code in Figure 3.2. For example, the transition `login` can only be fired when a "EXEC" event occurs. To check if the transition has to be fired, the assertion `match_name(e.objname, "login")` is first evaluated. If it is true the assertion of state `s2` is evaluated. Since the later provides no assertion, it is true by default. Then the code block `user = e.ruid; pid = e.pid;` of the transition is executed and a new instance of the scenario is created (since it is a non consuming transition) and it is assigned to state `s2`. In addition, a subscription for the transition `read_rhosts` of the new instance is inserted in the event spec database.

3.4.3 Comparing STAT to ASAX

The STAT approach and ASAX are very similar in terms of generality but they actually differ on some points. We now provide a comparison between both systems along two axes: the architecture and the language used to define scenarios.

Architecture

STAT-based IDSs and ASAX share some similarities about their architectures. One of these similarities is that both systems convert raw event records into a normalized internal format. The approach of STAT allows a flexible use of types for STAT event fields. Since scenarios are not interpreted but translated to C++ code of which compiled objects are linked to the core and executed at runtime, STATL scenarios can directly manipulate types used to define STAT event fields (which are defined in C++). On the contrary, in ASAX, RUSSEL rules are interpreted. Thus, types must be defined in the specification of the language RUSSEL. Adding application-specific types to RUSSEL requires to modify its specification and its implementation. As a result, ASAX adopts the approach of defining all fields in NADF records as generic strings of bytes (type `string`).

We can also observe that scenario instances in STAT and rule instances in ASAX are very similar in the sense that the two data structures represent a code to execute on the current event. The execution of scenario instances in STAT are more optimized than the execution of rule instances however. The concept of event subscriptions allows the STAT core to treat the transitions interested in the current event only. Nevertheless, looking closer the source code of STAT reveals that the event spec database is actually a list, which is traversed to retrieve event specs (and thus associated subscriptions) matching the current event. This operation is done each time a new event is delivered to the core. This fact entails a waste of resources if more event specs are defined than registered subscriptions. In that case, retrieving subscriptions via the event spec database takes a longer time than testing the

event spec associated to each subscribed transition individually.

Language

The language STATL provides a layer over C++. This layer allows the programmer to structure C++ statements according to a state/transition architecture. Then, since STATL scenarios are translated into full C++ files and finally into executable objects, scenarios do not need to be interpreted by an evaluation engine as in ASAX. Thus, STAT is normally faster than ASAX if the latter is not optimized. Moreover, encapsulating C++ statements gives access to a large number of primitives available for C++. However, this approach also has drawbacks. STATL scenarios have to be compiled in two passes. The first pass only detects errors in the STATL layer while C++ errors are detected by the C++ compiler. The utilization for the programmer is thus not as flexible as for ASAX for example.

Modeling attack scenarios as a composition of states and transitions is very intuitive. However, in applications out of the intrusion detection domain, reasoning in terms of states and transitions can be less intuitive as using a more imperative approach such as proposed by RUSSEL. For example, assuming the simple program that displays the TCP header of every packet in a network trace, the RUSSEL implementation would only contain one rule re-triggering itself for all records. The STATL implementation would contain a state and a transition. The state has only a code block performing the display while the transition is a loop on the state. It has no assertion and must match all event types. This an artificial way to force the display for every event.

Scenario plugins to be used by the STAT core are defined in a XML file. Another XML file defines which states of the defined scenarios raise an alert. When a new scenario is added both files must be completed accordingly. When a scenario is removed, the references to this scenario must be removed from both files. This utilization is not as flexible as for ASAX. In ASAX, there is no configuration file to define the rules to execute. The definition is actually made by a specific RUSSEL module which include other modules with the keyword `uses` (see Section 2.3.2). The module is then given as input to ASAX and it is treated as a regular rule. It is thus easier to switch from one configuration to another when the applied scenarios vary a lot.

A last observation we can make is that scenarios in STAT cannot communicate to each other in any way. The scope of global variables is limited to all instances of a same scenario, as for static in JAVA classes for example. Global variables in RUSSEL allow rules to communicate to each other. We have illustrated that feature with an example involving rules notifying that they have treated an event to other rules (see Section 2.4.2).

3.4.4 Translating STATL scenarios to RUSSEL rules

The Darpa benchmark we have used to assess our optimization techniques are translations of STATL scenarios from USTAT into RUSSEL rules. The translation procedure we have used is direct and involves simple RUSSEL rules. However, it is not applicable to all STATL scenarios because it assumes the following hypotheses:

- At most one transition is fired from a state at each event.
- For each unwinded state (a target state of a unwinding transitions), there are not both consuming and non-consuming transitions starting from the state.
- There is no situation where an unwinding transition involves to discard instances that were created by the sequence of events that led the unwinding instance from the target state to the source state.

All USTAT scenarios satisfy these hypotheses. To translate other scenarios that are out of these hypotheses, we need to use more complex mechanisms that we do not describe.

We have adopted the following general rule to perform the translation:

- Each state is represented by a RUSSEL rule.
- Each transition starting from a state is represented by a guarded command inside the corresponding RUSSEL rule.

Assuming a state and the transitions starting from this state in Figure 3.4, transitions are labeled with a triplet e, a, c where e is the type of event (the action) to match to fire the transition, a is the assertion of the transition while c is its code block. In addition to their names, target states are labeled with a pair $a'; c'$ where a' is the assertion of the state and c' is its code block. The corresponding RUSSEL translation of the state is as follows:

```
rule s_0(<parameters>);
if
   EVENT = e1 and a1 and a1'
     --> <transition1 action>

   ...

   EVENT = en and an and an'
     --> <transition n action>

   true --> trigger off for_next s_0(<parameters>);
fi
```

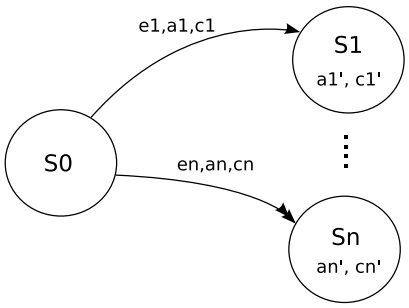


Figure 3.4: Example of a state transition diagram fragment to translate

where `<parameters>` is the parameters of the rule and `<transition i action>` is the RUSSEL translation of the code block $c_i; c_i'$ and $1 \leq i \leq n$. if no transition is fired for the current event, the state (rule) remains alive thanks to the default re-trigger achieved by the guarded command `true -->` **trigger off for_next** `s0(<parameters>);`.

In STATL, local variables of a scenario are used to correlate events. These variables are modified by assignment statements inside the code blocks of transitions and states. These assignments must not be translated to RUSSEL. Instead, the values affected to the local variables are used as actual parameters to trigger the rule representing the next state. These values either update the actual values of existing parameters in `<parameters>` or they complete the later with new parameters. Of course, the rule translating the target state is declared with the corresponding formal parameters. We translate a `<transition i action>` where $1 \leq i \leq n$ as follows:

begin
  `f(c_i);`
  `f(c_i');`
  trigger off for_next `s_i(<parameters updated by c_i and c_i'>)`
end

where f(c) is the RUSSEL translation of the code block c without the assignments to local variables.

Also, the rule must be re-triggered if the transition is non-consuming in order to keep the state valid. In that case, the translation of the transition action is thus completed as follows:

begin
  `f(c_i);`
  `f(c_i');`
  trigger off for_next `s_i(<parameters updated by c_i and c_i'>)`
  trigger off for_next `s_0(<parameters>)`
end

Unwinding transitions are considered as consuming transition if the target state is no more alive. If it is still alive, it is not needed to trigger the rule representing the target state. The statement `skip` is thus used instead of a trigger action, meaning no operation is performed.

Let us illustrate our procedure on the state `s1` from the STATL scenario in Figure 3.2. The RUSSEL translation of state `s1` follows:

```
rule s1(inode: integer);
if
   EVENT = exec and match('*/login', PATH) = 1
     --> begin
       trigger off for_next read_rhosts(inode, PID, RUID);
       trigger off for_next login(inode)
     end;

   EVENT = delete and FILE_INODE_ID = inode
     --> skip;

   true --> trigger off for_next login(inode)
fi;
```

The rule has one parameter memorizing the inode ID of the file of the current instance of the scenario. The first guarded command is the translation of the transition `login`. As it is a non-consuming transition, the rule re-triggers. The STATL instructions `user = e.ruid; pid = e.pid;` in the code block of the transition are assignments to local variables. The assigned values are thus passed as parameters to the triggered rule as achieved by the trigger action **trigger off for_next** `read_rhosts(inode, PID, RUID)`. The second guarded command is the translation of the unwinding transition `delete_file1`. As the target state is still alive, there is no need to trigger its rule. The last guarded command is the systematic re-trigger when no transition is fired.

3.5 Bro

Bro [49] is a stateful NIDS that monitors network traffic to detect intrusion activity. Its mechanism consists of extracting high-level events from a stream of network packets. Events are then analyzed by user-defined scripts reflecting the security policy of a site. The scripts are written in a specialized high-level language –called the Bro language– and describe the actions Bro should take regarding the monitored activities. For example, such actions can be updating structures containing information about the activities, raising alerts or closing ports as a mean of reaction against an intrusion threat.

In the following of this section, we make a description of the architecture of Bro. We also make a comparison with ASAX.

3.5.1 Architecture of the Bro system

The architecture of Bro is based on three layers performing analyses at different semantic level: packet filtering, event generation and policy script application (see Figure 3.5). Captured network packets are first filtered by using a BPF expression [41]. The filtered packets are then analyzed by a so-called *event engine* generating high level events which reflect specific activities on the network. The event engine is divided into a *protocol analyzer* that follows the communication between two endpoints, and a *signature engine* that matches the content of incoming filtered packets against patterns. Finally, security policy user-defined scripts are executed on the events by a *policy script interpreter*. We now describe each layer of the architecture of Bro in details.

Packet filtering

The network traffic is captured by the `libpcap` library [2]. Using this library isolates Bro from the hardware layer of the network. Besides, the `libpcap` library has been ported to many platforms and thus it eases the portability of Bro. Bro offers the possibility to filter packets by defining filters in the Bro language at the policy layer. These filters are compiled to a BPF expression and the latter is used by `libpcap` library to filter the captured packets. For example, packets can be filtered on the basis of their port numbers or control flags (if the packet is a TCP segment). Bro is configured to filter out all packets belonging to a protocol it does not know. In addition, all TCP packets with the SYN, FIN or RST flag are accepted.

Event engine

The purpose of the *event engine* is to give an abstract "view" of the network traffic which is further exploited by the policy layer. This "view" consists of a sequence of *high-level events*, giving a model for activities in the network traffic. Each event has an identifier denoting the corresponding activity such as connection establishments or specific command requests from a host to a server for example. Events have also arguments holding information related to the event, for example, the command of a FTP request.

In the implementation of Bro, an event is represented by a C++ object. Arguments of the event are hold in the object. The object also contains a pointer to an *event handler* that treats the event in the policy layer. When generated, events are placed in a queue.

During the analysis, the event engine first performs integrity checks on every filtered packet. It checks if the packet header is well-formed including

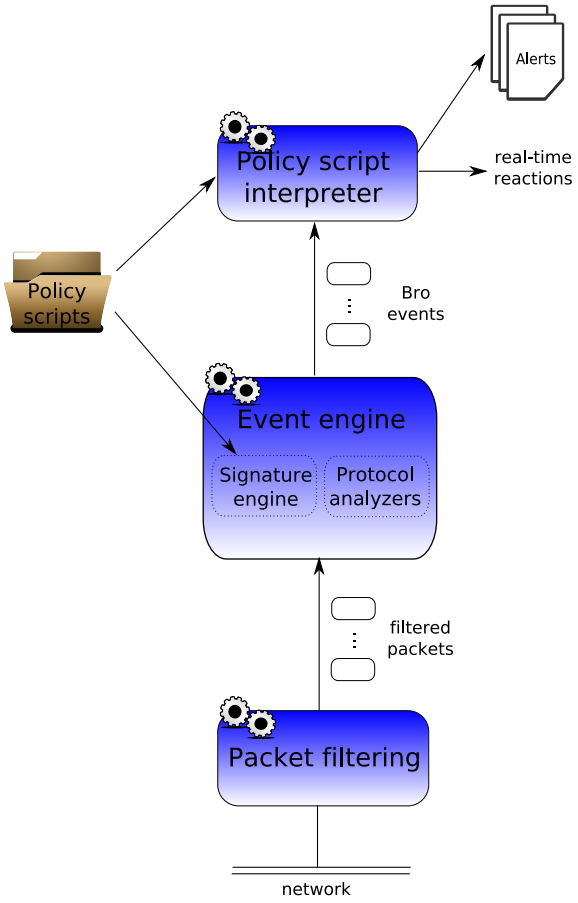


Figure 3.5: Runtime architecture of Bro

the verification of IP header checksum. If the packet is not valid, it is discarded and an event indicating the failed check is generated. If the packet is valid, it is dispatched to the *protocol analyzer* and to the *signature engine*.

Protocol analyzer Every packet is first treated by a *connection handler* that analyzes the packet at the transport layer level. Bro defines a higher-level concept of "connection" for TCP, UDP and ICMP protocols though only TCP is connection-oriented. TCP connections are defined by the protocol itself. For UDP and ICMP, the protocol analyzer considers a "pseudo-connection" is established when a host A sends a packet to a host B, meaning that A initiates a request to B. Bro instantiates a C++ object for each connection. The object contains information about the two endpoints (IP addresses and port numbers[3]) and the current state of the connection.

[3]For ICMP "connections", port numbers are fixed to 0

For each packet, the event engine looks up the "connection" associated to the tuple of the two IP addresses and the two port numbers inside a hash table. If the packet is the first of a novel connection, a new "connection" object is created and inserted into the hash table. The packet is then treated by the connection handler associated to the "connection".

The connection handler for TCP packets first checks if the TCP header of the packet is valid. If it is, it analyzes the content of the header. If the header has the SYN, FIN or RST flag, the connection state is adjusted accordingly. State changes may generate specific events; when a TCP connection is established, the `connection_established` event is generated. If a connection is not established when the time-out initiated at the first SYN packet expires, the `connection_attempt` event is generated. Similarly, if the endpoint replies a RST packet, the connection is rejected and the `connection_rejected` is generated. Connection handlers for UDP and ICMP are simpler; keeping a "connection" state for UDP communications allows the handler to know which endpoint makes a request and which endpoint makes the reply. `udp_request` and `udp_reply` events are generated respectively for UDP requests and replies. ICMP traffic is handled similarly.

When the processing of the connection handler ends, a *payload handler* analyzes the payload of the packet. For example, if the packet is part of a FTP session, a specialized handler inspect the content of the payload and generates FTP related events like `ftp_request`, `ftp_reply` or `file_transferred`. If the packet is part of an unknown application protocol to Bro, analysis stops.

It has to be noted that handlers (connection and payload) are hardcoded inside the C++ code. Adding new handlers for novel protocols requires thus to recompile the Bro binary.

Signature matching *Signature matching* was not present in the original version of Bro. This module provides the concept of *contextual signature matching*, which is an improvement of the string-based signature matching [57]. The concept augments the string-based matching process with high-level context provided by the protocol analyzer or by the policy layer. Signatures are defined in a specialized language by the user. A signature description is divided into two sections: conditions and actions. There are 4 types of condition:

- *Header conditions* match packet headers to limit the applicability of the signature to a subset of network traffic. For example, condition `ip-proto == tcp` matches TCP traffic only and condition `dst-port == 80` matches traffic destinated to a http server.

- *Content conditions* match a pattern against the payload of a packet. The pattern is defined by a regular expression. The match can be

tested the condition statement `payload`, wich applies the pattern on the raw payload data in the packet. Alternatively, the condition can be prefixed by a label corresponding to a protocol analyzer. Therefore, it matches the pattern against the data as extracted by the indicated analyzer. For example, condition `http /(etc\/(passwd|shadow)/` matches any request containing either `/etc/passwd` or `/etc/shadow`.

- *Dependency conditions* define dependencies between signatures. For example, condition `requires-signature` followed by a signature name defines the latter signature has to match on the same "connection" first. This type of condition allows Bro to define signature of multi-steps attacks, which is not possible with stateless NIDS like Snort for example.

- *Context conditions* uses other components of Bro to refer to information about the context of the analysis. For example, condition `tcp-state` defines restrictions on the TCP state of the TCP connection involving the analyzed packet. The condition of having the connection established is `tcp-state established`. Another example is condition `eval`, which calls a script function declared in Bro language in the policy layer. That function has to return a boolean and is typically used to test contextual information collected by scripts during the analysis.

Conditions are written in the order above and are not tested if all conditions declared before are not true. The only action a signature can take is the generation of a special `signature_matched` event with the `event` statement followed by a string describing the matched activity.

Figure 3.6 contains an example of a Bro signature from [57] detecting an attack performed in two steps. This attack is the intrusion of the Slapper worm by exploiting a potential vulnerability of a web server. The first step of the attack probes the server for its potential vulnerability by sending a badly formed http request and by inspecting the reply of the server. If the server is Apache, the second step of the attack tries to exploit an OpenSSL vulnerability on TCP port 443 to inject the worm in the system. Signature `slapper-probe` detects the first step of the attack while signature `slapper-exploit` detects the second step. The correlation between the two steps is done thanks the used context conditions; the condition `eval is_vulnerable_to_slapper` calls a function verifying whether the running web server is Apache and uses the vulnerable version of OpenSSL. The condition `eval has_slapper_probed` checks if the destination host (server) has already been probed, assuming the policy layer has conserved the information when it treated the event generated by the first signature. A dependency condition cannot be used here because packets performing the probe and accomplishing the exploit do not belong to the same connection. It is

```
signature slapper-probe {
  ip-proto == tcp
  dst-ip == x.y.0.0/16 # sent to local net
  dst-port == 80
  payload /.*GET \/ HTTP\/1\.1\x0d\x0a\x0d\x0a/
  eval is_vulnerable_to_slapper # call policy fct.
  event "Vulner. host possibly probed by Slapper"
}

signature slapper-exploit {
  ip-proto == tcp
  dst-ip == x.y.0.0/16
  dst-port == 443 # 443/tcp = SSL/TLS
  eval has_slapper_probed # test: already probed?
  event "Slapper tried to exploit vulnerable host"
}
```

Figure 3.6: Example of a contextual signature in Bro

also important to note that stateless NIDS such like Snort cannot perform such detection since it does not keep information about the past in memory. For more example of Bro signatures, we invite the reader to refer to [57].

Regular expressions are matched efficiently. This is done by compiling each expression to a DFA (Deterministic Finite-state Automata) of which final states indicate a matching. With that mechanism, a string of length n is matched in $O(n)$ operations where each operation is the firing of a transition performed by an efficient array lookup in constant time. To avoid matching expressions individually, and hence a very expensive processing time, regular expressions are unified into a single global expression. The global expression is compiled to a global DFA for all small expressions and again a string of length n is matched in $O(n)$ operations.

Policy script interpreter

The "view" of the traffic offered by the event engine is policy-neutral but defines the concepts (events) on which the security policy is modeled. The analysis of a packet by the event engine results in an event queue. The *policy script interpreter* treats events in the queue in FIFO order by applying scripts written in the specialized Bro language on them. Bro scripts define so-called *event handlers*. For each event passed to the interpreter, the corresponding handler for the event is retrieved via the pointer stored in the event object (see above). Arguments of the event are passed to the handler. Then the execution of the event handler analyzes these arguments and may

perform side effect actions as generating new events, logging alerts, executing external routines to react against the intrusion threat or recording data to disk. It also can modify the state of the "connection" corresponding to the event. When treated by an event handler, each event is consumed and removed from the queue. Once the queue is empty, the next packet in the input stream is analyzed.

The Bro language We only make a short description of the Bro language in the context of this thesis. A full description is provided in [49]. The Bro language is strongly typed and is network specific. In addition to the common primitive types such as `bool, int, double` and `string`, it proposes many built-in types well adapted for network analysis. One can cite, for instance, types `addr` and `port` that respectively are for IP addresses and TCP or UDP port numbers. Bro also supports a number of aggregate types:

- The `record` type is a regroupment of elements of arbitrary types like in C. For example, Bro provides the predefined `conn_id` type to stand for connection identifiers:

  ```
  type conn_id: record {
    orig_h: addr;
    orig_p: port;
    resp_h: addr;
    resp_p: port;
  };
  ```

  Fields in the record are accessed like in C but using the '`$`' instead of '`.`' (e.g., `c$orig_h`).

- The `table` type is a collection of elements of the same type. The table is indexed by indices of an atomic type or of an aggregate type. For example,

  ```
  table[conn_id] of ftp_session_info
  ```

  defines a table indexed by a `conn_id` record and yielding to records of type `ftp_session_info`.

- The `set` type is similar to the `table` type except that there is no yielded value. For example,

  ```
  set[conn_id]
  ```

  defines a set of elements of `conn_id` type.

The language offers numerous arithmetic, logical and relational operators like in languages such C (e.g., `+,-,*,/,&&,||,==,<=`). In addition, it provides the `in` operator for `table` and `set` types. It returns a `bool` if a given index is in a given table or set. For example,

```
21/tcp in sensitive_services
```

is true if the TCP port number 21 is in the given set.

The Bro language specifies event handlers and functions. Event handler declarations are prefixed by the `event` keyword and followed by an identifier and arguments. The identifier must be the same as the event the handler treats. For example,

```
event ftp_request(c: connection, command: string, arg: string)
```

starts the declaration of an event handler treating `ftp_request` events. Functions are similar to event handlers except that they return a value and are prefixed with keyword `function`.

Bro offers two levels of scope for variables: global to the entire script or local to an event handler or function. Global variables are declared using the `global` keyword while local variables use the keyword `local`.

Statements in Bro are the assignment with the `=` operator, `if` conditions and some predefined routine calls (e.g, `add` adds an element to a set, `delete` removes an element from a set or table and `print` writes a string to a file).

Figure 3.7 shows a sample of a real Bro script delivered with the Bro archive []. This sample handles `pop3_reply` events generated by the POP3 protocol analyzer in the event engine. For each POP3 reply emitted by a server, the handler checks if it contains an error message (indicated by the command `ERR`). If it is, it means that the request was not valid and that the originator potentially acts something suspicious. A counter is thus incremented for the connection. If that counter grows beyond a defined threshold, an alarm is raised. It is interresting to note that application-layer POP3 sessions is followed by the script and not by the event engine. The script maintains a table `pop_connections` indexed by a connection record and yielding to records containing information about the current POP3 sessions. A new entry is added in the table when a new POP3 connection is established. Similarly, an entry is removed when the corresponding connection terminates.

3.5.2 Comparing Bro to ASAX

Bro and ASAX look similar in a number of aspects. But they also differ in some fundamental points. As for STAT, we now provide a comparison between both systems, involving an additional axe: stateful analyses. This new axe is the aspect on which ASAX and Bro diverge a lot.

```
type pop3_session_info: record {
  id: count;             # Unique session ID.
  quit_sent: bool;       # Client issued a QUIT.
  last_command: string;  # Last command of client.
};

global pop_connections: table[conn_id] of pop3_session_info;
global pop_connection_weirds: table[addr] of count;


event pop3_reply(c: connection, is_orig: bool, cmd: string, msg: string)
{
  local conn = get_connection(c$id);

  pop_log(conn, cmd,
    fmt("%.6f #%d < %s %s", network_time(), conn$id, cmd, msg));

  if ( cmd == "OK" )
  {
    if ( conn$quit_sent )
      delete pop_connections[c$id];
  }

  else if ( cmd == "ERR" )
  {
    ++pop_connection_weirds[c$id$orig_h];
    if ( pop_connection_weirds[c$id$orig_h] > error_threshold )
      print log_file, fmt("%.6f #%d %s/%d > %s/%d WARNING: error count exceeds threshold",
        network_time(), conn$id, c$id$orig_h, c$id$orig_p, c$id$resp_h, c$id$resp_p);
    }
  }

function get_connection(id: conn_id): pop3_session_info
{
  if ( id in pop_connections )
    return pop_connections[id];

  local conn: pop3_session_info;

  conn$id = ++pop_session_id;
  conn$quit_sent = F;
  conn$last_command = "INIT";
  pop_connections[id] = conn;

  print log_file, fmt("%.6f #%d %s/%d > %s/%d: new connection",
    network_time(), conn$id, id$orig_h, id$orig_p, id$resp_h, id$resp_p);

  return conn;
}
```

Figure 3.7: Sample of a Bro script

Architecture

Some architectural aspects of both systems are similar. They both divide the analysis into two layers: a layer treating raw data and a layer analyzing the result of the former. In Bro, the event engine produces off every packet a sequence where each element is an event treated by an associated code (event handler) in the upper layer. In ASAX, the format adapter produces off every input raw record a normalized NADF record further analyzed by a set of active instances of rules.

We also can observe a similarity between Bro events and instances of rules in ASAX. The latter are implemented by a C-structure containing a pointer to the rule code to execute. Events in Bro are implemented by C++ objects containing a "similar" pointer to the code of the event handler to execute. The event queue generated by the event engine is thus comparable to the *current set* of rules in ASAX.

Despite these similarities, the philosophy of the approach taken by each system differs however. The NADF format used by ASAX is simpler and more flexible than Bro's model. The rationale of defining this format is to allow existing audit trails to be translated to it in a straightforward way. Moreover, using the NADF format allows one to mix heterogeneous audit streams and to apply different kinds of analyses on them, independently to a specific application domain. The simplicity of the NADF format entails a simple design for the analysis process, amenable to general optimisations we develop in this thesis. Bro is specialized to the network domain and does not need to analyze other sources than network traffic. Thus, audit records in Bro, i.e., events, are represented by a less generic model, particularly by an object-oriented model implemented in C++. This is less flexible because modifying the concepts used by analyzes, i.e., the event taxonomy, requires to modify the imple mentation of Bro.

A last observation we can make is that Bro is less modular than ASAX. Assuming the user wants to analyze a new application protocol, he needs to implement the corresponding analyzer. Hence, he must investigate and master the source code of the event engine in order to integrate his own analyzer properly. About ASAX, this problem does not exist since the whole analysis procedure is implemented in RUSSEL. Thus, the user does not need to investigate the internal mechanism of ASAX.

Language

RUSSEL (the ASAX's language) is based on simple concepts that are applicable to any kind of analysis. RUSSEL supports two primitive types only: (signed) `integer` and `string of bytes`. The latter is the generic type for all data not being integer, and for NADF fields. Thus, the programmer must take that point into account when tailoring analysis rules and must

use external C-routines to make data interpretation more domain-specific. For example, a routine suite has been developed in [65] to manipulate data specific to the network domain. In Bro, there are many primitive types related to the network domain (e.g., `addr, port, conn_id`). In addition, Bro offers aggregate types `record, table` and `set` to ease data manipulation.

Stateful analyses

Both systems are able to express stateful scenarios (also known as multi-steps scenarios). In ASAX, the current states of tracked scenarios are stored in RUSSEL rule parameters and in (especially frozen) global variables (see Section 2.4.2). In Bro, there is no concept of rule parameters keeping information. State handling is performed inside two layers of Bro: the event engine and the policy interpreter.

Bro's event engine keeps state of every "connection". State information is stored inside objects instantiated for each "connection". When analyzing a packet, the event engine retrieves then the state of the associated "connection" by looking up the "connection" in a hash table. Since this task is automatically done by this layer, the programmer is dispensed with it.

It is inside the policy layer that states of application sessions must be handled, and thus by the programmer. Typically, policy scripts delivered with Bro [1] maintain tables holding session information for each supported application protocol. These information are retrieved by indexing the tables with connection ID records (`conn_id` type predefined in Bro). For example, Figure 3.8 shows the definition of a Bro record holding information (i.e., state) of a FTP session. A table indexed by a `conn_id` and yielding to `ftp_session_info` records. When treating an (FTP) event, event handlers retrieve the corresponding record by accessing the table with the `conn_id` given to the handler. This is illustrated by Figure 3.9. The remainder of the code is then free to modify the content of the record and thus change the state of the session.

The advantage of the approach of ASAX is that one can write simple rules without involving structures like tables; passing parameters to rules gives direct access to the right state.

```
type ftp_session_info: record {
  id: count;
  connection_id: conn_id;
  user: string;
  anonymized_user: string;
  anonymous_login: bool;

  request: string;          # pending request or requests
  num_requests: count;      # count of pending requests
  request_t: time;          # time of request
  log_if_not_denied: bool;  # unless code 530 on reply, log it
  log_if_not_unavail: bool; # unless code 550 on reply, log it
  log_it: bool;             # if true, log the request(s)

  reply_code: count;        # the most recent reply code
  cwd: string;              # current working directory

  pending_requests: ftp_pending_cmds; # pending requests
  delayed_request_rewrite: table[count] of ftp_cmd_arg;

  expected: set[addr, port]; # data connections we expect
};

global ftp_sessions: table[conn_id] of ftp_session_info;
```

Figure 3.8: Declaration of a table of `ftp_session_info` records

```
event ftp_reply(c: connection, code: count, msg: string, cont_resp: bool)
{
  local id = c$id;
  local response_xyz = parse_ftp_reply_code(code);

  if ( id !in ftp_sessions )
    new_ftp_session(c, T);

  local session = ftp_sessions[id];

  if ( code != 0 || ! cont_resp )
    session$reply_code = code;

  ...
```

Figure 3.9: Sample of Bro code retrieving and modifying FTP session state

Part III

How to optimally select and execute relevant rules only

Chapter 4

Modifying the memory management to keep rule instances alive

4.1 Introduction

A significant part of the analysis time is spent to deallocate, allocate instance descriptors and to copy parameters. Since the rules mostly do nothing because they search for specific records, almost all instances re-trigger themselves. It is obvious that such instances do not need to be killed and recreated. We call *self-trigger* the action of a rule instance that re-triggers itself with the same parameters values for the next record. Keeping alive the instances that self-trigger and simply moving them to the next rule set would allow ASAX to benefit a significant speed-up.

In this chapter, we modify the memory management of ASAX to not perform useless reallocations. Our optimization consists of the following. At compile time, a static analysis is applied on the internal codes of all declared rules to detect all self-triggers. Afterwards, the internal codes of each self-trigger is replaced by a new single box we designed, called *selftrigger box*. Its execution performs a self-trigger without reallocating the instance descriptor. We have conducted experiments to calculate the average speed-up, i.e., the number of times the evaluation runs faster than the original version of ASAX, that we benefit from the optimization. Our optimization allows us to benefit an average speed-up of about 2.04 on the Darpa benchmark.

This chapter is organized as follows. Section 4.2 depicts the procedure to identify and replace self-triggers. Section 4.3 presents the results of the experiments we conducted with the enhanced version of ASAX. Finally, Section 4.4 draws the conclusion.

4.2 Replacing the self-triggers with a more efficient box

The internal representation of a trigger action is composed of a chain of boxes with one successor. The chain is made of three parts. The first part is a group of boxes that computes the values of the actual parameters and fixes the size of the memory area to allocate for the parameters. The results of the computed values are stored inside the working area (see Section 2.5.4 for a complete description of the working area). The second part is the trigger box, which allocates the instance descriptor and put it into the targeted rule set. Finally, the third part is a group of boxes whose role is to copy the actual parameters inside the descriptor. The first and last groups are organized in the same order as the passed parameters. Consider the following trigger action:

trigger off for_next `foo(x+1,y)`

Figure 4.1 depicts the corresponding chain of boxes. We did not present the boxes with operators *setLen*, *mv_int* and *cpy_int* in Chapter 2. The box setLen fixes the size of the memory area to allocate to store the parameters. x and y are two integers, hence the size to allocate is 8 bytes. The box *mv_int* copies the value of y at address $addr2$. The box *cpy_int* copies the value at address $addr2$ inside the instance descriptor.

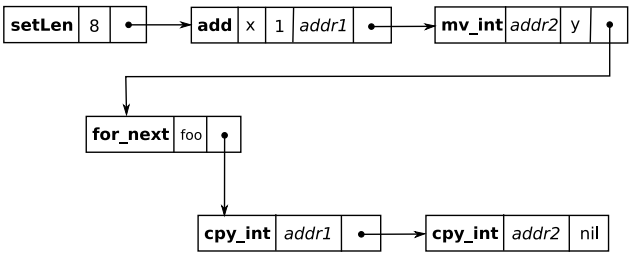


Figure 4.1: The box chain of a trigger action example

To keep instances alive, we need to detect self-triggers. There are two solutions to achieve this task: either we detect self-triggers at run time or we detect them at compile time. The first solution is costly because the parameter values must be checked each time a trigger "for next" action

occurs. The second solution is a static analysis on the internal code. Since it takes place during the compilation, it is impossible to test the actual values of parameters. As a result, only the trigger "for next" actions that reuse formal parameters are identified as self-triggers. Though this solution is less accurate than a run-time analysis, it is more convenient to avoid any overhead to occur during the execution. Consequently, we implement the second solution to identify self-triggers.

Once all self-triggers are identified, all the corresponding chains of boxes are removed from the internal codes and replaced by a single new box we designed, called the *selftrigger box*. Its structure is as follows:

It has no operand and one successor. The effect of its execution depends on whether the current instance has already been re-triggered by a previous box. In all cases the first self-trigger moves the current instance to the next rule set. There is no more memory allocation and less boxes are executed, hence the analysis should speed up. In case of multiple self-triggers are executed, additional instance descriptors are needed. As a result, the next self-trigger executions duplicate the descriptor of the current instance and put the copies into the next rule set. Let the following rule:

```
rule foo(p1,p2:integer);
begin
   if f = A
     --> trigger off for_next foo(p1,p2)
   fi;
   trigger off for_next bar
end;
```

Figure 4.2 illustrates the corresponding compiled codes before and after the self-trigger replacement.

Rule instances that self-trigger must not be killed once executed. This implies to modify the *analyze_record* algorithm as it is presented in Algorithm 4.1. The function *evaluate_rule*() returns now a boolean which equals *true* if the executed instance has been re-triggered.

4.2.1 Implementation details

For each chain of boxes composing a trigger "for next" action, the self-trigger identification is achieved in two steps. First, the triggered rule and the one performing the trigger must be the same. Recall that there is a pointer to the triggered rule descriptor inside the trigger box. It has to refer to the current analyzed rule. Second, whenever a formal parameter is passed to a trigger action, the computing of its value amounts to copy its content inside the working area. As a result, we scan the first part of the chain for

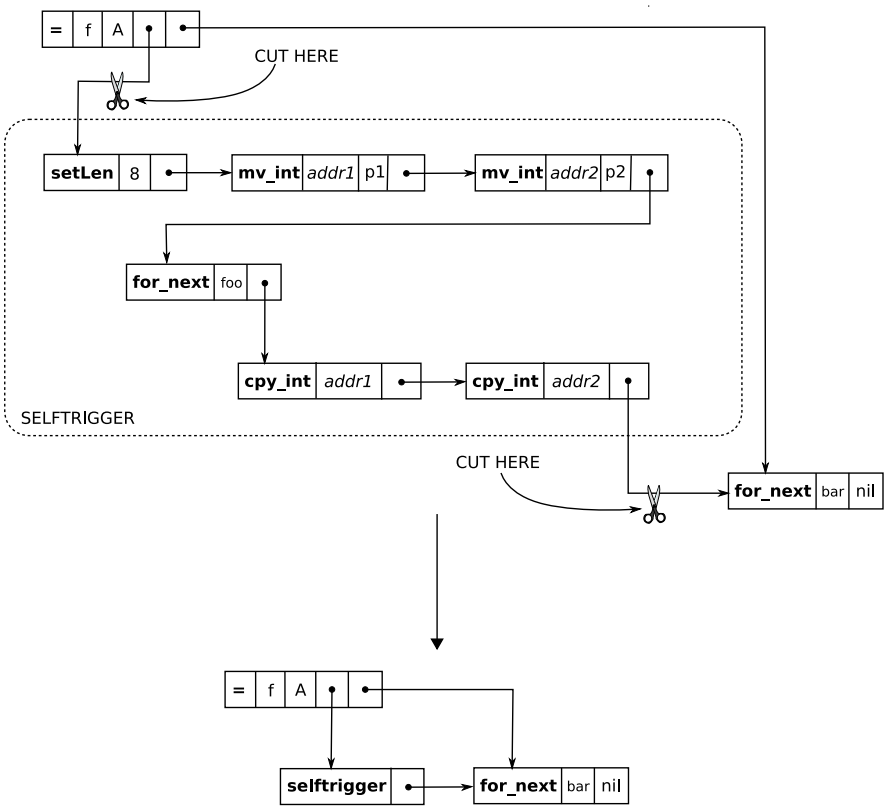


Figure 4.2: The box chain of a trigger action example

the corresponding boxes and the rest is ignored. These boxes are called *mv boxes*. Their structure is as follows:

op	dest	src	α

where *op* is either *mv_int* or *mv_str*, *dest* is the destination address of the copy while *src* is the source address. Figure 4.3 depicts the general situation a box-represented self-trigger. There must be a mv box for every formal parameters of the rule and each *src* pointer must refer to a distinct parameter inside the parameter area. Moreover, the mv boxes should be organized in the same order as the formal parameters.

4.3 Experimental measures

4.3.1 Description of the tests

To asses the improvement provided by the optimization, we execute the enhanced version of ASAX on the Darpa benchmark and we compare the analysis times with the former results from the original ASAX.

Algorithm 4.1 analyze_record(R, current) - Record analysis (modified version to handle self-triggers)

Precondition:

- R is the current NADF record.
- current is the current rule set.

Postcondition:

- All instances in current have been applied to the record R.

```
begin
  pre_process(R)
  while current is not empty do
    begin
      inst ← pick_one_instance(current)
      APA ← param(inst)
      selftriggered ← evaluate_rule(code(inst))
      if selftriggered = false then
        kill(inst)
    end
end
```

4.3.2 Results

Table 4.1 compares the analysis times of the original version (column ORIG) and of the enhanced version of ASAX (column STB) for each audit file (column Log). The column Gain indicates the percentage of the original time we spare. We witness an average gain of about 50.91% of the original time, proving that the original ASAX is spending half of its time in average to deallocate and to allocate instances that self-trigger. The speed-up of the analysis is of about 2.04 in average.

Log	ORIG	STB	Gain (%)
monday	55.765	26.155	53.1
thuesday	44.091	23.002	47.83
wednesday	109.313	56.304	48.49
thursday	55.477	28.495	48.64
friday	52.849	22.984	56.51

Table 4.1: Analysis times with the *selftrigger* box

Figure 4.4 illustrates analysis times with respect to the average number

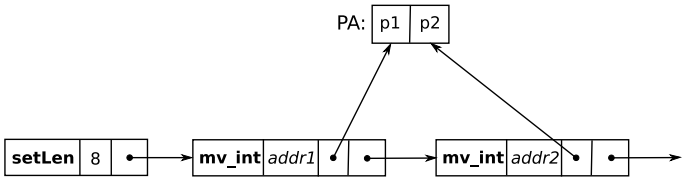


Figure 4.3: General form of a self-trigger

of instances for both original and enhanced versions of ASAX. The gain is more or less than 50% regardless the number of instances. The execution time of the enhanced enhanced version remains thus linear. Consequently, as it is pictured in Figure 4.5, adding the `buf_to_root_shell` module into the applied rule set still has a significant impact on the analysis time.

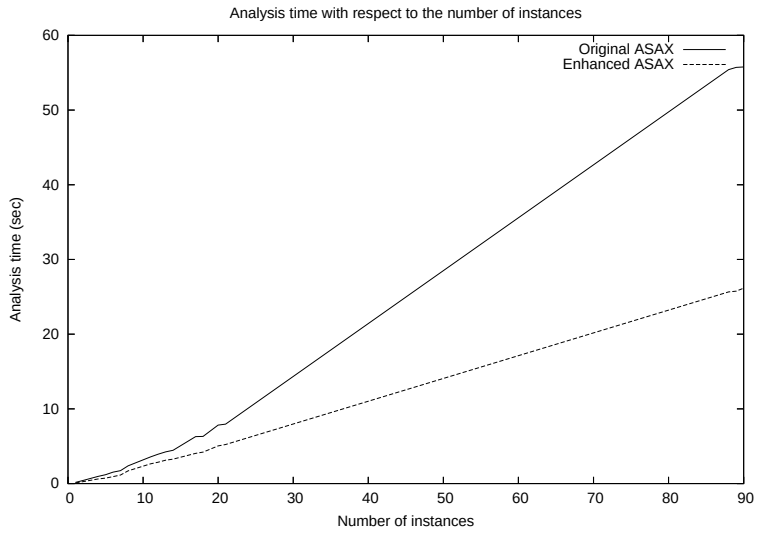


Figure 4.4: Analysis time with respect to the number of instances (with selftrigger box)

4.4 Conclusion

In this chapter, we introduced and integrated a new box whose role is to prevent rule instances that re-trigger themselves to be recreated. Experiments showed us that the enhanced analysis process exhibits a significant average speed-up of about 2.04 on the Darpa benchmark. However, it is

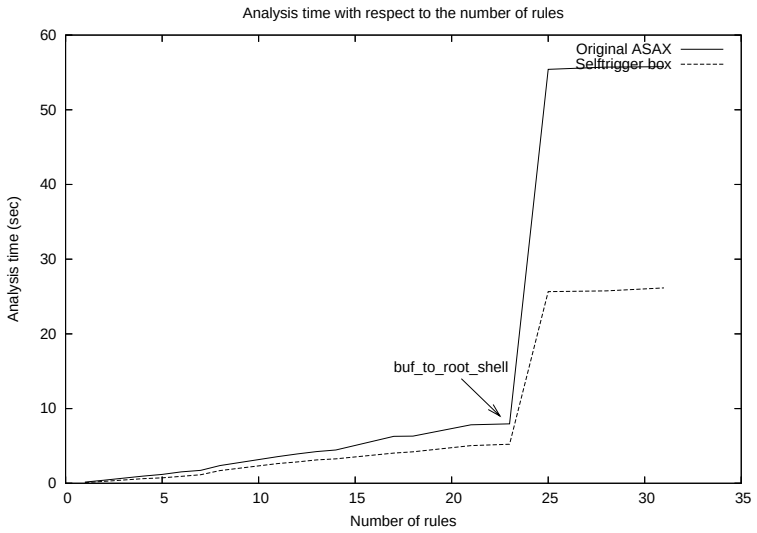


Figure 4.5: Analysis time with respect to the number of rules (with selftrigger box)

possible to reduce more the analysis time. Most instances are irrelevant to the analysis of the current record. Yet, they are executed on all records. The next chapter presents a technique to select those rule instances in order to prevent their executions.

Chapter 5

Using decision trees to execute relevant instances of rules

5.1 Introduction

Most of rule instances spend their time re-triggering for the next record because they search for specific records. Yet, they are executed on all records. Instances that are irrelevant to the analysis of the current record should not be executed. They can be either discarded or re-triggered for the next record without performing any other actions. In this chapter, we propose an optimization technique to prevent irrelevant rule instances to be executed. A static analysis of the internal code of the rules allows us to build for each rule a decision tree featuring the right decisions and conditions on the current event and on the rule parameters. At each ASAX cycle, binary decision trees are used to cluster the instances in three distinct sets: the instances *to execute*, the instances *to re-trigger* and finally the instances *to discard*. Only the instances from the first set are executed. The instances to re-trigger are moved to the next rule set. The instances to discard are deallocated. Since the execution is reduced to the relevant instances in addition to the evaluation of the decision trees, the overall execution time benefits from a speed-up of about 8.03 in average on the Darpa benchmark, with respect to the original implementation of ASAX.

The remainder of this chapter is organized as follows: Section 5.2 outlines the general idea of the optimization. Section 5.3 introduces some definitions and notations about binary decision trees that we use in the remainder of the dissertation. Then, we explain how decision trees are produced and how to integrate them in the analysis process respectively in Section 5.4 and 5.5. All details about the implementation of the optimization are described in Section 5.6. We then discuss experimental results in Section 5.7 before

concluding the chapter with Section 5.8.

A preliminary version of the work presented in this chapter is described in [40].

5.2 Using local decision trees to anticipate the rules behaviors

The key idea to avoid useless executions is to anticipate the behavior of the rules with respect to the current environment and the current record: either a rule is relevant to the analysis because it produces side effects and should be executed, either it is not relevant and is re-triggered or it simply dies.

Our approach to anticipate the rule behaviors can be summarized as follows. At compile time, a specific binary decision tree is built for each rule. We call it a *local decision tree*. This is a tree whose nodes contain single conditions about the current record to be analyzed. The leaves contain one decision among four possible ones: *execute*, *selftrigger*, *discard* or *parameters*. The first three decisions stand for an anticipated behavior. Whenever the decision is parameters, it is associated to a residual subtree, called *parametric subtree* whose conditions involve the actual parameters of the rule instances. The leaves of this residual subtree may contain three final decisions: execute, selftrigger or discard.

At run time, the local decision tree of each active rule is evaluated with respect to the current record at a pre-processing stage. It results in a specific decision for every active rule or for every rule instance when a parameters decision is taken. Assuming no parameters decision is taken, the decisions execute, selftrigger and discard means that all instances of the rule associated to the tree are respectively executed, self-triggered and discarded. Whenever a rule self-triggers, all its instances are moved altogether to the next rule set (the operation is in constant time independently of the number of instances to re-trigger). If the evaluation results in a parameters decision, the associated residual subtree is evaluated for each instance of the rule and decisions are taken individually. In this situation, the advantage of the self-trigger in constant time is lost.

Not optimizing rules that are triggered for the current record only

It is assumed that rules that are triggered for the current record only always need to be executed. Indeed, when they are triggered, it is for a specific purpose which surely entails side effects. Otherwise, they are useless. Thus, we do not optimize such rules by creating their local trees since it is useless.

5.3 Definitions

Before getting into the subject of the creation of local decision trees, we need to define a few abstract data types and several functions. We reuse all notations introduced in this section all along the remainder of the dissertation.

5.3.1 Binary trees

A binary tree is:

- either a leaf containing an arbitrary object.
- or a root containing an arbitrary object and two binary subtrees (left and right).

We define the following functions:

- leaf(L): denotes the binary tree composed of a leaf whose content is the object L.
- root(I, LST, RST): denotes the binary tree composed of a root, which contents the object I, and of LST and RST, respectively the left and right subtrees.
- external(T): true if T is a leaf, false otherwise.
- object(T): returns the object contained in the tree T.
- lst(T): returns the left subtree of T assuming external(T) is false.
- rst(T): returns the right subtree of T assuming external(T) is false.

5.3.2 Local decision tree

A local decision tree is:

- either a leaf containing a decision.
- or a binary decision tree with a root containing a condition C and two local decision subtrees STT and STF respectively the left and right subtrees.

We define the following functions:

- decision(T): returns object(T) assuming T is a leaf.
- condition(T): returns object(T) assuming T is a rooted tree with two childs.

- true(T): returns STT assuming T is a rooted tree of the form root(I, STT, STF).
- false(T): returns STF assuming T is a rooted tree of the form root(I, STT, STF).

5.3.3 Functions applied on conditions

- local(C): true if C is a condition on a local variable of a rule.
- global_frozen(C): true if C is a condition on a global frozen variable.
- parametric(C): true if C is a condition involving one or more parameters.

5.4 Extracting local decision trees

This section aims at describing the complete extraction process of local decision trees. The three following steps are applied for each rule after the compilation:

1. A local decision tree is extracted from the graph representation of the rule.
2. The resulting decision tree is reorganized so that conditions on parameter are moved to the bottom of the tree.
3. Each path of the tree is cut at the first condition on parameter. A *parameters* decision is associated to the path and the cut subtree is attached to the decision.

Each step is explained in details in the remainder of this section.

5.4.1 Building a local decision tree from a rule

Local decision trees are composed of conditions extracted from their corresponding rules. As the local decision trees are assumed to be evaluated at a pre-processing stage, only conditions whose evaluation result does not vary during a whole cycle can safely be extracted. Such conditions must test *constant expressions* only. Constant expressions involve only the following elements: literals, record fields, global frozen variables and actual rule parameters. In case an external C-function is used in the expression, it is assumed to not produce any side-effect and exhibits a constant behavior during a whole ASAX cycle. It is the responsibility of the programmer to ensure that constraint. On the contrary, the behavior of a rule cannot be

anticipated with the help of conditions involving *dynamic variables*, i.e. local variables and global variables (not frozen). Indeed, the values of these variables are defined by the rule executions and might change during a cycle.

We define the *selected path* of a rule as the execution path resulting from the evaluation of conditions. Considering that a relevant rule execution produces side-effects, we use the following principle to know what decision to take for a rule:

- A rule has to be executed if its selected path contains side-effect instructions. The instructions that we consider as producing side-effects are:
  - Procedure call: We make the arbitrary choice to consider that all procedures produce side-effects because they do not return any value. However, we consider that functions never produce side-effects so that we can extract conditions involving function calls. As a result, evaluating local trees featuring functions that produce side-effects might entail unspecified results. This is the programmer's responsibility to write rules accordingly.
  - Assignment to a global variable (frozen or not). An aassignmentto a local variable is not considered as a side-effect instruction because the variable scope is limited to the execution of the current instance.
  - Trigger of another rule.
  - Trigger of the same rule with parameters different from the formal parameters.
  - Trigger for the current record.
- A rule has to be executed if the selected path contains conditions on dynamic variables since we have no choice but execute one by one all rule instances to know their actual values.
- A rule has also to be executed if the selected path contains more than one self-trigger since triggering more instances for the next record is a side-effect.
- A rule has to be re-triggered for the next record if its selected path contains only one selftrigger without any side-effect instruction.
- A rule has to be discarded if its selected path contains only conditions on constant values.

Figure 5.1 depicts the selected path (in plain line) of the rule `count_failed_login` (introduced in Section 2.5.5) when `EVENT = login` and `RESULT = failed` are both evaluated to true. The path contains an

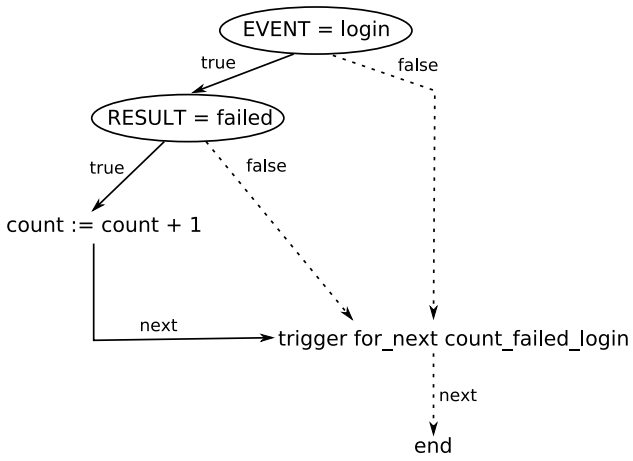


Figure 5.1: Selected path leading to the *execute* decision

assignment to the variable `count`. Since it is a global variable, that path produces a side-effect and the rule must be executed. In Figure 5.2, the selected path assumes that `EVENT = login` and `RESULT = failed` are evaluated to true and false, respectively. Since the path contains a single self-trigger, the rule must be self-triggered and not executed. Building a local

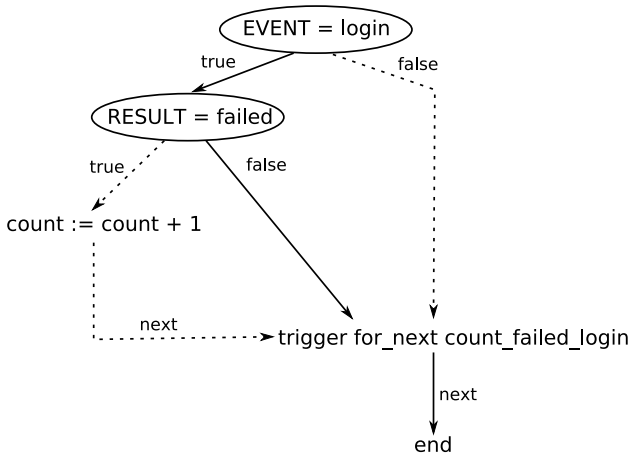


Figure 5.2: Selected path leading to the *selftrigger* decision

decision tree consists of regrouping all combinations of condition evaluations which lead to a specific decision inside a binary tree. Figure 5.3 presents the local decision tree corresponding to the rule `count_failed_login`. We adopt the following convention to graphically represent decision trees: taken paths resulting from a "true" condition evaluation are represented in plain

line while those resulting from a "false" condition evaluation are represented in dashed line.

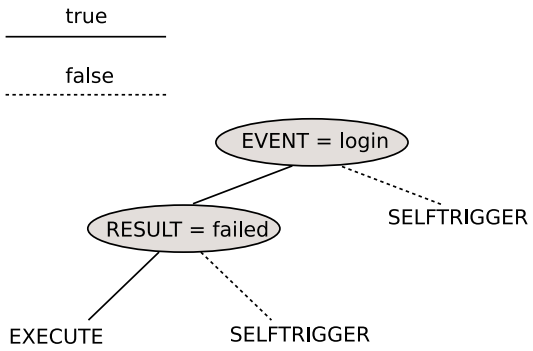


Figure 5.3: Example of a local decision tree

Algorithm 5.1 is applied to every rule to build their corresponding local decision trees. It is recursive and consists in a preorder traversal of a rule graph representation. Note that it reuses auxiliary functions introduced in Section 2.5.5. We define two additional types for nodes in rule graphs: a *side-effect node* represents an instruction producing side-effects and a *self-trigger node* stands for a trigger instruction which consists of a self-trigger. The function $condition(N)$ returns the corresponding condition of node N. The algorithm can be summarized as follows:

- If the current node of the graph represents an instruction producing side-effects, the leaf containing the execute decision is returned. Since we aim to anticipate rules behaviors, there is no need to continue further in the path. Assuming that fact, conditions occurring in the remainder of the path might not be extracted if any other path does not lead to them.

- If the current node of the graph represents a self-trigger, a leaf containing the execute decision is returned in case another selftrigger has been encountered earlier in the path. Otherwise, that encounter is memorized and the result of the recursive call on the next node is returned.

- If the current node is the special node "end", it means that the end of the current branch is reached. If a self-trigger has been encountered, a leaf containing the selftrigger decision is returned. Otherwise, assuming the execution of that path, the rule performs "nothing". A leaf which contains the discard decision is thus returned.

- If the current node represents an instruction which does not produce

any side-effect (e.g. an assignment to a local variable), it is ignored and the result of the recursive call on the next node is returned.

- If the current node represents a condition, the result of the algorithm depends on the condition type. If it is a condition involving dynamic variables, the rule needs to be executed to evaluate it. Thus, a leaf containing the execute decision is returned. Otherwise, the auxiliary algorithm *subtree* (depicted in Algorithm 5.2) builds and returns a subtree composed of the condition as root and of the subtrees resulting from the recursive calls on both child nodes. Besides, if both resulting subtrees are identical (it often happens that both subtrees are actually execute decisions), the condition is discarded and one of the identical subtrees is returned. Testing if two subtrees are identical is achieved in constant time thanks to a caching mechanism we explain in Section 5.6.4.

It is possible that a local decision tree of a rule contains conditions involving rule parameters. In that case, it is impossible to anticipate the behavior of the rule by evaluating the decision tree only once. Indeed, the conditions on parameter need to be executed for each instance of the rule. We explain how to solve this issue in the next section.

5.4.2 Reorganizing the local decision trees

Local decision trees featuring conditions on parameters are subject to an evaluation issue. Indeed, such conditions cannot be evaluated only once since the actual values of the parameters might differ from one instance to another. We call these conditions *parametric conditions* in contrast with the *global conditions*, which are conditions whose evaluation result does not differ from one instance to another.

One solution we propose consists of a post-processing which is applied to each local decision tree and is composed of two steps: The first step reorganizes the local trees so that parametric conditions are moved to the bottom of their trees. All paths are then of the form: $< c_1, \ldots, c_n, p_1, \ldots, p_m, d >$ where $c_1, \ldots, c_n$ are global conditions, $p_1, \ldots, p_m$ are parametric conditions and d is a decision. Moreover, during the reorganization, subtrees that have identical branches are simplified. Figure 5.4 illustrates an example of a reorganized decision tree assuming the condition P is parametric. Algorithm 5.3 is used to reorganize the trees. It uses the additional following functions: the function $get\_condition\_not\_param(T)$ returns the first encountered condition in tree T assuming a preorder traversal. The function $simplify(T, C, bool)$ returns a simplification of tree T according to the boolean value *bool* of condition C. Both functions have a $O(n)$ complexity where n is the number of nodes of the tree T. Since all subtrees of a local

Algorithm 5.1 BUILD(G, N) - Build a local decision tree from a rule

Precondition:

- G is a graph of RUSSEL rule and N is a node of G.
- status has two possible values: disc or selft.

Postcondition:

- Let the environment composed of a NADF record and of actual parameters for the rule. The algorithm returns a decision tree T so that if evaluating G leads to node N without producing side effects (status = disc) or if evaluating G leads to node N with producing a self-trigger as only side effect (status = selft) then:
  - if the evaluation of T returns selftrigger for the current environment then the execution of G only produces a re-trigger of the current instance;
  - if the evaluation of T returns discard for the current environment then the execution of G produces no side-effects.

```
if type(N) = end then
   if status = selft then
      return leaf(SELFTRIGGER)
   else
      return leaf(DISCARD)
else if type(N) = side-effect
   return leaf(EXECUTE)
else if type(N) = conditional then
   if local(condition(N)) or not global_frozen(condition(N)) then
      return leaf(EXECUTE)
   else
      begin
         STT := BUILD(G, succ(N, true), status)
         STF := BUILD(G, succ(N, false), status)
         return subtree(condition(N), STT, STF)
      end
else if type(N) = self-trigger then
   if status = selft then
      return leaf(EXECUTE)
   else
      return BUILD(G, succ(N, next), selft)
```

Algorithm 5.2 subtree(C, ST1, ST2) - Subtree construction

Precondition:

- C is a condition
- ST1 and ST2 are respectively left and right subtrees

Postcondition:

- If ST1 and ST2 are identical, the algorithm returns one of them. Otherwise, it returns the subtree with a root containing C, with ST1 as left subtree and with ST2 as right subtree right .

if ST1 = ST2 **then**
If ST1 and ST2 are identical, the current condition is skipped to simplify the tree and the unique resulting subtree is returned.
return ST1
else
return root(condition(N), ST1, ST2)

tree are recursively reorganized, the overall complexity of the algorithm is $O(n^2)$.

The second step cuts the paths at the first parametric condition. A parameter decision is then associated to the path $< c_1, \ldots, c_n >$ and the remaining cut subtree is attached to the decision and becomes a so-called *parametric subtree*. Figure 5.5 shows the final form of the decision tree presented in Figure 5.4. The task is performed by Algorithm 5.4.

In some situations, moving parametric conditions to the bottom of the local trees is not a good idea because some conditions –normally tested after a parametric condition– might be uselessly evaluated. As a result, another solution consists of mixing the parametric conditions with the global conditions. Subtrees whose root is the first condition on parameter of a branch are still replaced by a parameters decision. It is thus possible that the parametric subtrees contain global conditions. The drawback is that the latter may be evaluated more than once, i.e., for many instances. But the advantage is that less conditions are evaluated whenever another decision than parameters is computed.

Knowing which solution is the best is not trivial. If it is ensured, for a local tree, that no condition is evaluated uselessly, reorganizing the tree should be the most rewarding since it does not entail multiple evaluations of global conditions. We assess both solutions with the Darpa benchmark in Section 5.7.

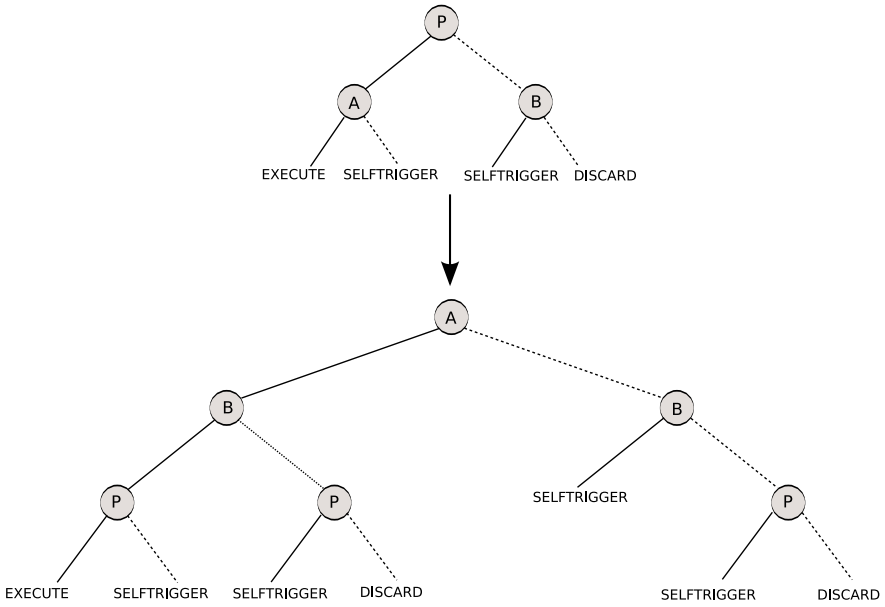


Figure 5.4: Example of a reorganized local tree with parameters at bottom

5.5 Evaluating local decision trees

Algorithm 5.5 presents a modification of the record evaluation algorithm. In case all instances of a rule are self-triggered, it is convenient to move all of them altogether to the next rule set in constant time, regardless of the number of instances to re-trigger. Therefore, instances are grouped by rule inside rule sets and the algorithm now processes the current rule set rule by rule. The local decision tree of each active rule is evaluated by Algorithm 5.6.

Assuming no parameter decision is taken, the optimization is achieved at the rule level, i.e., all instances of the rule are either executed, self-triggered or discarded. In fact, the analysis process only benefits from the two latter situations since it prevents the execution of any instance. On the contrary, if the parameters decision is taken for a rule, the optimization is achieved at the instances level and is thus less rewarding. It is still rewarding because the advantage of evaluating global conditions only once is still acquired. However, a subtree involving parametric conditions is evaluated for all instances to apply individual decisions to each of them. In this situation, the advantage of achieving massive self-triggers in constant time is lost since all self-triggered instances are treated separately.

Instances of rules that are triggered for the current record are not put into the current rule set anymore but into a set called *current_triggered*. These instances are then executed once the current rule set is fully treated.

The algorithm uses the additional following functions: The function

Algorithm 5.3 reorder_params(T) - Move parametric conditions to the bottom of a local decision tree

Precondition:

- T is a binary decision tree.

Postcondition:

- returns T' where all paths are of the form: $< c_1, \ldots, c_n, p_1, \ldots, p_m, d >$ where $c_1, \ldots, c_n$ are global conditions and $p_1, \ldots, p_m$ are parametric conditions and d is a decision.
- For all environments, evaluating T' returns the same decision as evaluating T.

```
if external(T) then
   return T
else
   begin
      C ← get_condition_not_param(T)
      STT ← reorder_params(simplify(T, C, true))
      STF ← reorder_params(simplify(T, C, false))
      return subtree(C, STT, STF)
   end
```

pick_one_rule(s) removes and returns an arbitrary rule from the set s. The function *local_tree*(R) returns the local decision tree of the rule R. The function *instances*(R) returns a list containing all instances of rule R. The function *move_instances*(R, s) puts all instances of rule R into the rule set s (in constant time). The function *put_instance*(I, s) puts the instance I into the rule set s. Finally, Algorithm 5.6 uses the function *evaluate_condition*(C), which returns *true* if the condition satisfies, $false$ otherwise.

5.6 Implementation details

The internal representation of rules is their internal codes, so Algorithm 5.1 is implemented to work at a box level. Considering decisions are special instructions, a decision tree is actually a RUSSEL rule containing conditions only. As a result, local trees are implemented by boxes and are injected into the ASAX evaluator engine to collect decisions. Additional boxes implement the decisions. When a condition is extracted to be part of a local tree, its corresponding internal code is copied to the tree. This section presents the different adjustments we made to the compiler and to the virtual machine to integrate local trees. In addition, we have implemented a tree caching mechanisms to allocate identical subtrees only once.

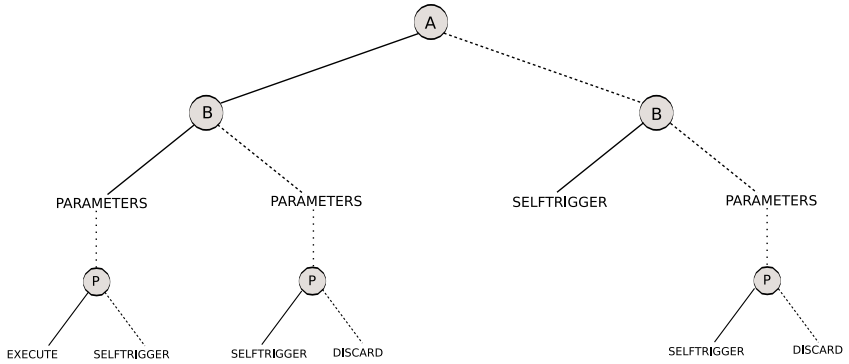


Figure 5.5: Example of a final local tree with *parameters* decisions

5.6.1 Modifying the compiler to fetch conditions

Recall that relational conditions are composed of a relational box preceded by boxes calculating the tested expressions. Internal codes, as produced by the compiler, do not allow us to locate the "first box" of relational conditions. So when rule codes are scanned to build local trees, relational conditions are identified thanks to their final (relational) box. However, we need to know from which box they actually start to extract them properly. To achieve it, we have modified the content of the relational box, as depicted in Figure 5.6, to hold a pointer to the "first box" forming the conditional instruction. The compiler is modified accordingly.

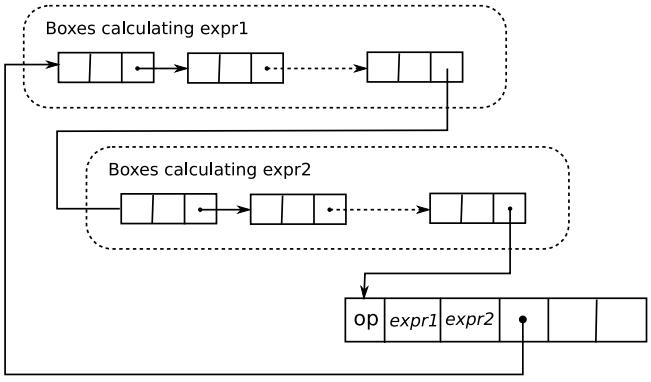


Figure 5.6: Relational Box Modification

Algorithm 5.4 place_parameters_decisions(T) - place the parameters decisions in a local decision tree

Precondition:

- T is a reorganized binary decision tree with parametric conditions at bottom.

Postcondition:

- returns T', the binary decision tree T where all paths (of the form $< c_1, \ldots, c_n, p_1, \ldots, p_m, d >$) are cut at the first condition on parameter (p_1). A *parameters* decision is associated to the path $< c1, \ldots, c_n >$ and the remaining cut subtree is attached to the decision.

```
if external(T) then
   return T
else if parametric(condition(T)) then
   return leaf(parameters(T))
else
   begin
     STT ← place_parameters_decisions(true(T))
     STF ← place_parameters_decisions(false(T))
     return root(condition(T), STT, STF)
   end
```

5.6.2 A meta box for conditions

The caching mechanism we describe in Section 5.6.4 implies that we have to test if two conditions are equal. Testing whether two relational conditions are equal is not trivial. The chain of boxes composing both conditions must be identical excepting that all references inside the working area are ignored. These references may change from one chain to another even if they represent the same condition.

To ease the manipulation of conditions while building local decision trees and also to spare memory allocation, we have designed descriptors for conditions. A condition descriptor identifies a condition and contains the chain of boxes forming the internal code of the condition. Thus, testing if two conditions are equal simply amounts to compare the addresses of their descriptors instead of comparing the whole box chains. A special novel box called *condition box* is used to represent internal nodes of local decision trees. Every condition box contains the reference to the descriptor of the condition represented by the corresponding node. Conditions boxes that represent the same condition contains the same descriptor reference. Thus, for all local trees, there is only one occurrence of the chains of boxes forming the conditions allocated in memory. The structure of the condition box is as follows:

Algorithm 5.5 analyze_record(R) - Record analysis (modified version with local decision trees)

Precondition:

- R is the current NADF record.

Postcondition:

- All instances in current have been treated (either executed, self-triggered, or discarded depending on the evaluation of their corresponding local decision trees)

```
begin
   pre_process(R)
   while current is not empty do
     begin
       rule ← pick_one_rule(current)
       decision ← evaluate_decision_tree(local_tree(rule))
       process_rule(rule, decision)
     end
   while current_triggered is not empty do
     begin
       inst ← pick_one_instance(current_triggered)
       APA ← param(inst)
       evaluate_rule(code(inst))
       kill(inst)
     end
end

process_rule(R, D)
if D = execute then
   for all inst in instances(R) do
     begin
       APA ← param(inst)
       selftriggered ← evaluate_rule(code(inst))
       if selftriggered = false then
         kill(inst)
     end
else if D = selftrigger then
   move_instances(rule, next)
else if D = discard then
   for all inst in instances(R) do
     kill(inst)
else (D = parameters)
   for all inst in instances(R) do
     process_instance(inst, param_subtree(D))

process_instance(I, PST)
begin
   APA ← param(inst)
   decision ← evaluate_decision_tree(PST)
   if decision = execute then
     begin
       selftriggered ← evaluate_rule(code(inst))
       if selftriggered = false then
         kill(inst)
     end
   else if decision = selftrigger then
     put_instance(inst, next)
   else if decision = discard then
     kill(inst)
end
```

Algorithm 5.6 evaluate_decision_tree(T) - Evaluation of a decision tree

Precondition:

- T is a decision tree

```
if external(T) then
   return decision(T)
else if evaluate_condition(condition(T)) = true then
   return evaluate_decision_tree(true(T))
else
   return evaluate_decision_tree(false(T))
```

condition	*cdescr*	α_t	α_f

where *cdescr* is the pointer to the associated condition descriptor. The pointer to the next box to execute is α_t in case the associated condition is true, α_f otherwise. Evaluating this box amounts to evaluate the (boxes of the) condition from the condition descriptor pointed by *cdescr*.

5.6.3 Modification to the virtual machine

In order to group instances by rule and thus to fetch all instances of a rule when they need to be self-triggered, rule sets are splitted in smaller subsets. Each subset contains the instances of one distinct rule. As a result, rule sets are implemented by a list of lists (linked). The first level of the list is the rule level and its cells are implemented by rule descriptors which have been extended for the occasion. The second level is the instance level and it uses the original implementation involving instance descriptors. Figure 5.7 pictures the structure of an extended rule descriptor. It contains the rule name, a pointer to its internal code and a pointer to its local decision tree. There is only one occurrence of a rule descriptor in memory. As a result, the same rule descriptors are physically used to form the current rule list and the next rule list at the same time. Rule descriptors contain thus two pointers to the next rule descriptor in both lists: one refers to the next rule in the current rule list while the other refers to the next rule in the next rule list (respectively pointers *nextcurrent* and *nextnext* in the figure). Besides, rule descriptors contain two pointers referring respectively to the list holding the current instances of the rules and to the list holding the next instances. A self-trigger is performed by concatenating the first list onto the end of the second list. To perform the operation in constant time, we need to maintain pointers referring to the first and last instance of both lists. Moreover, we have implemented a mechanism to avoid to treat all rule descriptors at the end of each ASAX cycle when moving the instances of the next rule set to the current rule set. Assuming that the two lists containing the current and

next instances of the rule are named $list1$ and $list2$, $list1$ is the current list for odd records while $list2$ is the next list. The situation is inverted for even records. Despite these modifications of the rule set implementation, the operation time of adding new instances into rule sets remains constant. Indeed, trigger boxes contain a pointer to the descriptor of the triggered rule and, consequently, to the right cells in rule lists.

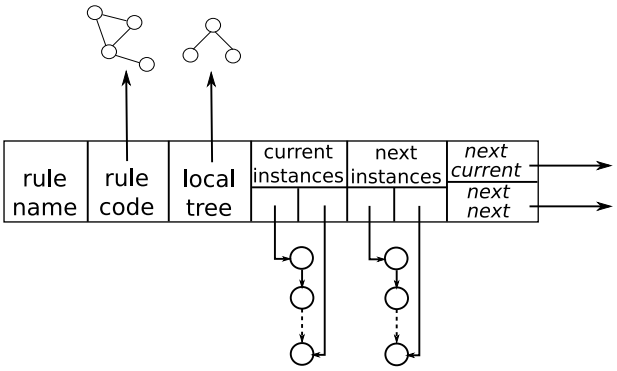


Figure 5.7: Structure of the rule descriptor

5.6.4 Optimizing space allocation

We have implemented a mechanism that aims to spare memory when lots of subtrees are built. In fact, it happens very often that many subtrees are equal. We use memory caching based on hash tables for all subtrees the algorithms generate. A cached tree is composed of either a single leave or a condition with two cached subtrees. Having a recursive structure allows us to be optimal since it is only needed to compare one condition (descriptor) and two pointers to test if a generated subtree is already cached. Using cached subtrees to build local decision trees implies that they actually are diagrams.

This optimization is also rewarding in terms of time when the algorithm `subtree` (Algorithm 5.2) tests whether two subtrees are identical. Indeed, as cached subtrees are identified by their pointer, testing the equality of subtrees amounts to compare their pointers and the operation is thus achieved in constant time.

5.7 Experimental measures

5.7.1 Description of the tests

To measure the improvement provided by our optimization, we execute an ASAX version featuring local decision trees on our Darpa benchmark. We

compare the analysis times with the previous results from both the original version and the version enhanced with the selftrigger box.

5.7.2 Results

Table 5.1 compares the analysis times of the enhanced version of ASAX to the previous versions for each audit file. The column Log indicates the analyzed file. The columns ORIG and STB are the analysis times of respectively the original version of ASAX and the version with the selftrigger box optimization. Columns LOC and RLOC indicate the analysis times for versions featuring local decision trees respectively with and without reorganization. Columns Gain provides the percentage of the original time we spare for both versions of local decision trees. We can see that the average gains are of 87.33% and of 87.13% of the original time, which is a significant improvement since STB lead to an average gain of 50.91%. However, reorganizing local trees or not does not make a significant difference (for our benchmark). It is possible that this difference appears more drastically for other benchmarks. If we retain the best results, i.e results from local trees without reorganization, an additional average speed-up of about 3.95 is observed with respect to STB, making the overall average speed-up raise to 8.03.

Log	ORIG	STB	LOC	Gain (%)	RLOC	Gain (%)
monday	55.765	26.155	6.671	88.03	6.759	87.88
tuesday	44.091	23.002	6.502	85.25	6.682	84.84
wednesday	109.313	56.304	11.328	89.64	11.332	89.63
thursday	55.477	28.495	6.618	88.07	6.774	87.79
friday	52.849	22.984	7.572	85.67	7.664	85.5

Table 5.1: Analysis times with the local decision trees

Recall that we progressively add rule modules into the rule set to measure the evolution of the analysis time with respect to the number of rules. We can see in Figure 5.8 that adding the `buf_to_root_shell` module does not have a dramatic impact on the execution time anymore as for the previous versions. It is explained by the fact that stateful rules nearly never found relevant records (actually the module detects two attacks only). As a result, these rules receives the selftrigger decision for most records, hence their instances are never executed but rather massively self-triggered in constant time. Moreover, it appears that STB is the fastest version as long as the analysis does not involve rules with many instances. It means that local decision trees do not suit situations involving stateless rules only because evaluating their local decision trees amounts to evaluating the rules themselves (considering the selftrigger box optimization is integrated) and thus

it results in executing stateless rules twice when they receive the execute decision.

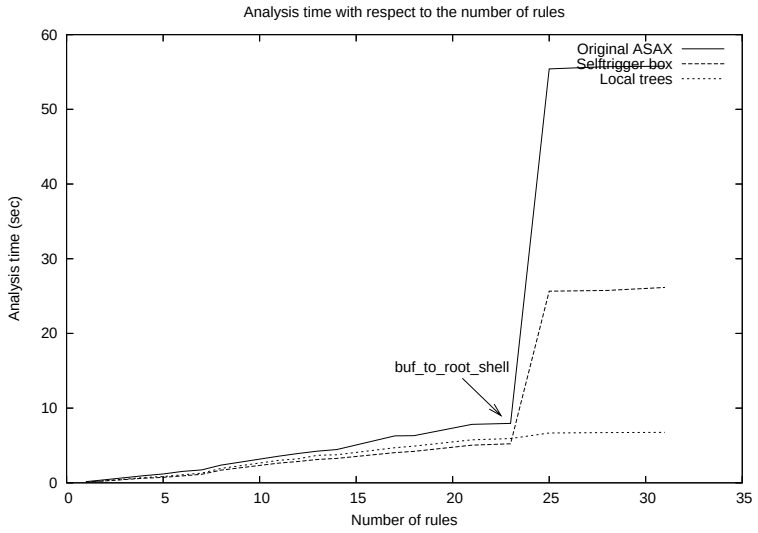


Figure 5.8: Analysis time with respect to the number of rules (with local trees)

Figure 5.9 depicts the evolution of the analysis times with respect to the number of instances. The observations are similar to the ones for the previous figure. The analysis times beyond 23 instances involve stateful rules. Since they keep self-triggering during the analysis, the execution time is constant regardless the number of instances (the curve is flat starting from 23 instances). Moreover, as for the previous figure, we see that STB is fastest as long as only stateless rules are active (before 23 instances).

5.8 Conclusion

In this chapter, we have described an optimization technique introducing local decision trees to execute relevant instances of rules only. Irrelevant instances are either self-triggered for the next record or discarded. The optimization is the most rewarding when the selftrigger decision is taken for all instances of the rule. The self-trigger operation is performed in constant time regardless the number of instances. The analysis process benefits now of an overall speed-up of 8.03 on the Darpa benchmark. Nevertheless, it can be improved further. First, local decision trees do not suit situations involving stateless rules only because evaluating their local decision trees amounts to evaluating the rules themselves. The new optimization is thus less rewarding than the ASAX version which features the selftrigger box

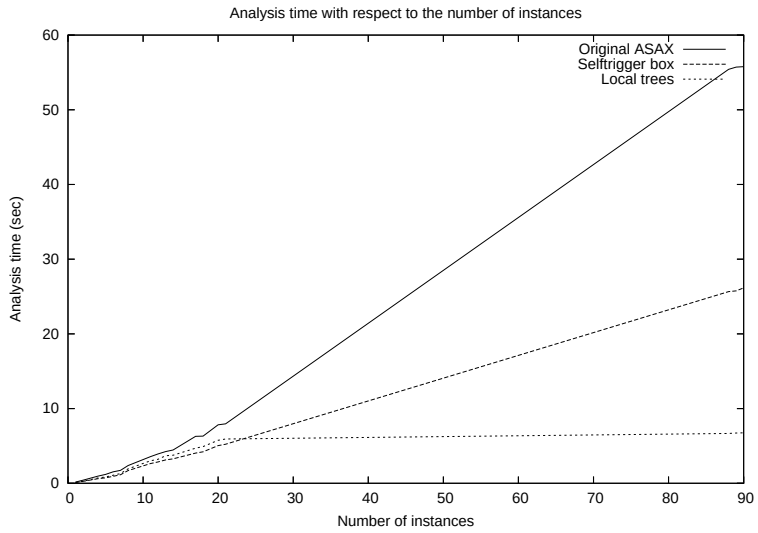


Figure 5.9: Analysis time with respect to the number of instances (with local trees)

optimization solely. Second, conditions involving parameters need to be evaluated several times as opposed to the global conditions. As a result, local trees are reorganized to separate the global conditions from subtrees which are subject to be evaluated several times. It entails that global conditions that are less critical than some parametric conditions may be evaluated uselessly.

The next chapter addresses the first issue by proposing another optimization consisting in merging all local decision trees into a global tree. Since intrusion detection rules usually contain common (or incompatible) conditions, evaluating a single global tree should be more efficient than evaluating local trees separately

Chapter 6

A global decision tree to avoid redundant tests

6.1 Introducing the problem

In the previous chapter, we have presented a technique to select relevant rules to execute. It consists of evaluating, at each cycle, a local decision tree for each active rule to determine which rule instances to execute. Despite that the provided overall speed-up of the analysis process is significant, it is still possible to improve on it. Indeed, it is likely that local decision trees share common conditions. Such conditions are evaluated several times, i.e., for each tree in which they occur. The idea of this chapter is to factorize condition evaluations so that each condition is tested at most once. To achieve it, we propose to merge all local decision trees to form a single *global decision tree*. Evaluating the global decision tree gives the same decisions as evaluating all local decision trees separately. Each branch contains at most one occurrence of any condition.

We develop the merging algorithm in several steps. First, we set up an algorithm that does not consider the (possibly) growing size of the global tree. It is not a good idea because the size actually grows exponentially and we cannot achieve a complete compilation for our benchmark. We call this algorithm the *naive merging*. We need additional optimization techniques to keep the size of the global tree as small as possible during the merging operation. To simplify algorithms, we use *sorted trees*, which require a reorganization of the nodes (according to some order) inside each local tree. Merging sorted trees allows us to merge faster but, the size of the global tree may still grow exponentially and the compilation cannot be completed on the Darpa benchmark.

We introduce two optimization techniques to reduce the size of the global tree during the merge operation. First, if two local trees share no common condition, evaluating them separately is as efficient as evaluating a merge

of the two. It is thus advantageous to avoid the costly merging process. *Sequential trees* generalize this idea to **all subtrees of the global tree**. It consists of "sequentializing" independent subtrees, i.e., subtrees that do not share common conditions, instead of merging them. This method prevents node duplications and thus reduces the global tree size. The main difficulty of this approach is to identify independent subtrees.

Second, we say that two conditions are *exclusive* if they cannot be true at the same time or, more generally, if the value of one of the two is determined by the value of the other. For example, the conditions A = 10 and A = 20 are exclusive, that is: they cannot be true at the same time. Thus, we *simplify* the global tree by not inserting the second condition in the "true" branch of the first one during the merging operation.

The two techniques stated above can be used separately but are complementary since they optimize two distinct aspects. Using sequential trees or simplified trees alone makes the compilation very fast (between 0.5 and 1 second in average). This result is drastically improved when we use both techniques together however (the compilation takes about 0.02 seconds). In addition, using the resulted global decision tree makes the overall analysis process benefit from a speed-up of 16.9 (with respect to the original version of ASAX) in the best case we measured on the Darpa benchmark.

To illustrate the various techniques we present in this chapter, we use a simple running example consisting of three different local trees with only one condition node and two decision leaves, depicted in Figure 6.1. The decisions E and T stand respectively for execute and selftrigger.

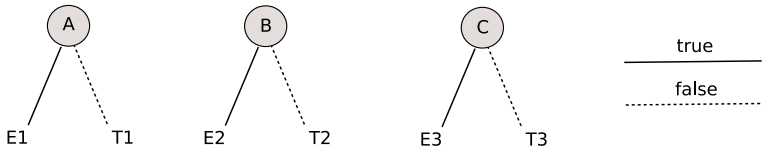


Figure 6.1: Three local trees to be merged

The remainder of this chapter is organized as follows. Section 6.2 describes the naive merging. Section 6.3 introduces sorted trees to set up the basis of the next optimizations. Section 6.4 and Section 6.5 define and assess sequential and simplified trees respectively. Section 6.6 puts them together and provide the final results. Section 6.7 concludes the chapter.

A preliminary version of the work presented in this chapter is described in [40] (naive merging only) and in [39].

6.2 Naive merging

6.2.1 Merging local decision trees

The *naive merging* consists of merging all local decision trees to form a bigger tree whose leaves are decision sets, instead of single decisions. Each set contains a single decision for every rule. To avoid redundant evaluations, each branch may not have more than one occurrence of any single condition. Our merging algorithm is a loop, which is executed on each local tree. It starts with an empty global decision tree, which is merged with a local decision tree at each iteration.

Algorithm 6.1 is applied at each iteration. The principle of the merging is simple: an occurrence of the local decision tree is added at the end of each branch of the global decision tree. The local tree is inserted node by node. Before inserting a condition, the algorithm checks if it is already present in the current branch of the global tree. This operation is achieved by traversing the current branch of the global tree starting from its root (it has thus $O(n)$ complexity where n is the number of nodes in the global tree). If the condition to insert is already present in the branch, the condition is skipped and the merging continues with its "true" or "false" branch depending on which branch has been taken from the first occurrence. The algorithm uses a function $decision\_set(G, L)$, which adds the decision L to the set of decisions G.

Figure 6.2 depicts the resulted merge involving the trees of our running example, assuming they are merged successively in the order they are presented in Figure 6.1.

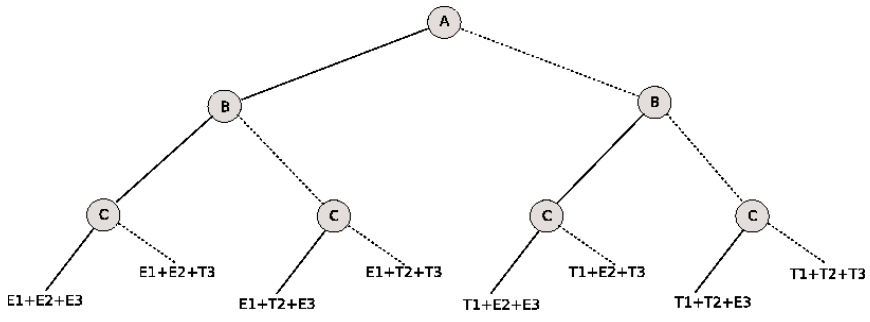


Figure 6.2: A global tree

Figure 6.3 depicts another example of a resulted merge for one iteration. The local tree features a condition (A) which is already present in the global tree. The leaves noted S_x are sets of decisions.

Algorithm 6.1 MERGE(G, L) - Naive merging of a local decision tree with a global decision tree

Precondition:

- G is a global decision tree and L is a local decision tree.

Postcondition:

- The algorithm returns a global decision tree GT such that if the evaluation of G returns a decision set Ds for the current environment, and if the evaluation of L returns a decision D for the current environment, then evaluating GT returns a decision set Ds' such that $Ds' = Ds \cup \{(rule\_name, D)\}$.

```
if external(G) then
   if external(L) then
      return leaf(decision_set(G, L))
   else if condition(L) is present in the current built branch then
      if 'true' path taken from the occurrence of condition(L) in the current
build branch then
         return MERGE(G, true(L))
      else
         return MERGE(G, false(L))
   else
      begin
         STT ← MERGE(G, true(L))
         STF ← MERGE(G, false(L))
         return subtree(condition(L), STT, STF)
      end
else
   begin
      STT ← MERGE(true(G), L)
      STF ← MERGE(false(G), L)
         return subtree(condition(G), STT, STF)
   end
```

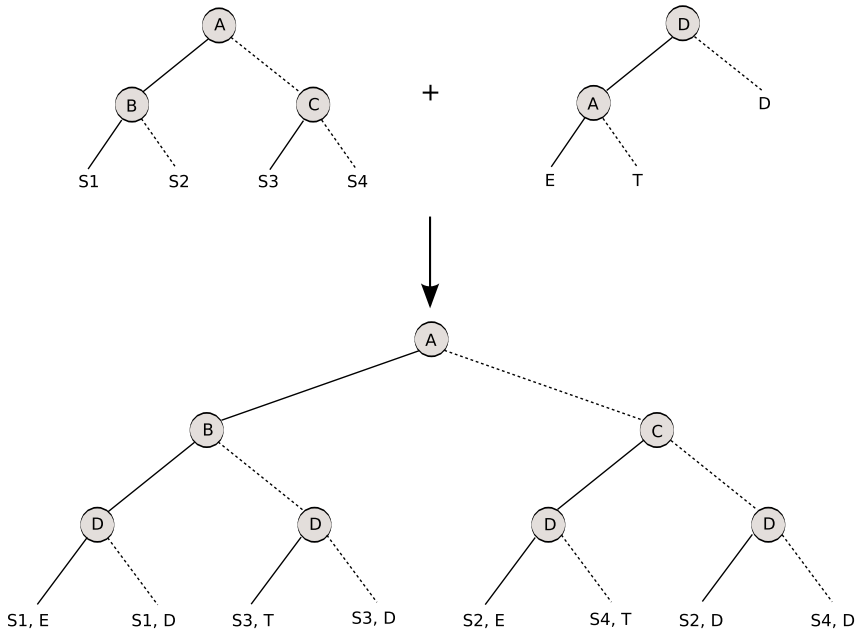


Figure 6.3: Example of a merge between a global decision tree and a local decision tree

6.2.2 Evaluating the global decision tree

Aside the fact that it returns a decision set instead of a single decision, the evaluation algorithm of the global decision tree is exactly the same as for the local decision tree. However, the record analysis algorithm differs now slightly; local decision trees are no more evaluated and are replaced by the evaluation of the global decision tree just before processing the active rules. Algorithm 6.2 presents the new version of the algorithm.

6.2.3 Implementation details

The structures used to implement the global decision tree are exactly the same as for local decision trees. However, a decision set is represented by a chain of boxes where each box stands for a single decision. Besides, to associate the decisions with their corresponding rules, we modify the structure of the decision box to hold a pointer referring to the descriptor of the associated rule.

To spare compilation time, local trees which are composed of a single leaf are not merged. Rules having such local trees have a fixed decision during the whole analysis. It is thus useless to calculate it and to integrate it inside the global decision tree. However, fixed decisions must be stored in memory to be able to process their corresponding rules properly.

Algorithm 6.2 analyze_record(R) - Record analysis (modified version with the global decision tree)

Precondition:

- R is the current NADF record.
- The environment contains a global decision tree GDT.

Postcondition:

- All instances in current have been treated (either executed, self-triggered, or discarded depending on the evaluation of the global decision tree GDT)

```
begin
  pre_process(R)
  DS ← evaluate_decision_tree(GDT)
  while current is not empty do
    begin
      rule ← pick_one_rule(current)
      decision ← get_decision(DS, rule)
      process_rule(rule, decision)
    end
  while current_triggered is not empty do
    begin
      inst ← pick_one_instance(current_triggered)
      APA ← param(inst)
      evaluate_rule(code(inst))
      kill(inst)
    end
end
```

6.2.4 Experimental measures

We have implemented the merging algorithm in ASAX and tested our approach on our IDS benchmark. Table 6.1 shows that our optimization is unpractical when the merging involves many local trees, i.e., many rules. The columns CT and GN provide the merging time (in seconds) and the number of nodes for the global tree up to the rule given in the first column. The column GNNC contains the number of nodes when the tree caching is not activated. We stopped merging the trees after the first twelve ones, which took more than two minutes. Obviously, completing the process for the nineteen remaining rules is hopeless. The merging time is exploding because the size of the global tree grows exponentially. Indeed, the worst case for the size of the global tree is $2^{C+1} - 1$ where C is the number of

RULE	CT	GN	GNNC
create	0.02	6	13
mod_perm	0.02	22	111
exec	0.02	54	303
exit	0.02	70	607
mod_perm	0.03	158	1,411
write	0.05	293	3,283
write	0.09	423	6,163
read	0.63	1,202	54,291
access1	3.41	1,970	150,483
access2	7.34	2,530	162,003
access3	39.77	3,430	216,623
write	133.69	8,550	1,046,063

Table 6.1: Merge Results

distinct conditions.

We can also make the observation that the figures provided by the column GNNC are way bigger than the numbers in the column GN. This proves that tree caching is an essential feature to spare memory allocation.

6.3 Sorted trees

The inefficiency of naive merging is principally due to the exponential growth of the global tree size. Besides, another important cause of inefficiency is the fact that the algorithm checks if a condition is already present in the current branch before inserting it.

To speed up and simplify the merging algorithm, it is convenient to assume an order over the set of all conditions and to sort the trees accordingly. This allows us to go down in parallel in the two trees that are merged and to avoid the costly checking process. Common conditions are simplified when the algorithm merges two subtrees with the same root.

6.3.1 Sorting local decision trees

To sort local decision trees, we first need to define an order over the set of all conditions inside the local trees. The choice of a good ordering is crucial to speed up both the merging process and the evaluation of the global tree. We discuss this issue in Section 6.3.4.

Local decision trees are sorted by Algorithm 6.3. The sort process is similar to a selection sort. It selects the smallest condition of the tree, i.e., the most priority condition, to be the root of the sorted tree. Then, the tree is simplified assuming the selected condition is true or false. The two simplified

trees are sorted recursively and become the subtrees of the returned sorted tree. The algorithm uses the auxiliary function $get\_smallest\_condition(T)$, which returns the smallest condition of the tree T and $simplify(T, C, val)$, which returns the tree T simplified assuming the value val of C. Since selecting the smallest condition and simplifying the tree requires scanning all n nodes of the tree three times (once for the selection and twice for the simplification) and since it has to be done for all nodes, the algorithm has $O(n^2)$ complexity.

Algorithm 6.3 SORT_LOCAL_TREE(T) - Sort a local decision tree

Precondition:

- T is a local decision tree

Postcondition:

- returns T' so that:
  - all paths of T' are of the form: $< c_1, ..., c_n, d >$ where $c_1, ..., c_n$ are conditions and d is a decision.
  - $\forall i, j : 1 \leq i < j \leq n : c_i < c_j$
  - the evaluation of T' gives the same results as for T.

```
if external(T) then
  return T
else
  begin
    C ← get_smallest_condition(T)
    STT ← SORT_LOCAL_TREE(simplify(T, C, true))
    STF ← SORT_LOCAL_TREE(simplify(T, C, false))
    return subtree(C, STT, STF)
  end
```

6.3.2 Merging sorted decision trees

Algorithm 6.4 merges sorted trees. The main difference with merging unsorted local trees is that we go down in the two trees in parallel. Since common conditions are simplified when the algorithm merges two subtrees with the same root, it is no more needed to check if a condition is already present in the current branch.

We assume that the conditions in our running example (see Figure 6.1) are ordered as follows: A < B < C. The local trees are sorted since they only contain a single condition. Figure 6.4 depicts the corresponding global tree. In general, the worst case for the size of the global tree is $2^{C+1} - 1$

where C is the number of distinct conditions. The worst case is the same as for naive merging.

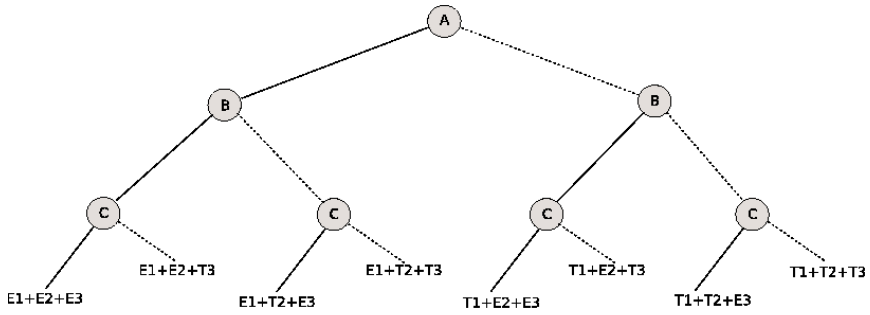


Figure 6.4: A sorted global tree

6.3.3 Implementation details

To compare conditions with respect to the defined order, a unique number ranging from 1 to the number of conditions is assigned to each condition. This number stands for the priority of a condition according to the order. The lower is the number the higher is the priority of its condition. The number of a condition is stored inside its descriptor.

6.3.4 Choosing an order on conditions

The choice of a good ordering on conditions is crucial to both build and evaluate global trees efficiently. To keep global trees small –and thus to make the merging process faster– we should put at the top conditions that occur in many local trees. However, reorganizing the local trees may entail useless evaluations of conditions, which do not take place when evaluating unsorted local trees. This may have a significant impact on the execution time. Ideally, we should use an ordering that leaves the local trees "as unchanged as possible" to have a small number of useless evaluations. This can be contradictory to the objective of having a global tree as small as possible. Therefore, we define various orders proposing a compromise between the two criteria. Every ordering we describe now is defined by a keyword that is used to discuss our experiments in Section 6.3.5. Note that none of the following order criteria can determine a total order between conditions, in general. The "non comparable" conditions are sorted by their order of appearance.

Algorithm 6.4 MERGE(A, B) - Merging two sorted decision trees

Precondition:

- A and B are sorted decision trees

Postcondition:

- The algorithm returns GT, the merging of A and B (see specification of Algorithm 6.1).
- Each condition in GT is bigger than its ancestors (with respect to the defined order on the conditions)

```
if external(A) then
   if external(B) then
     return leaf(union(decision(A), decision(B)))
   else
     begin
       STT ← MERGE(A, true(B))
       STF ← MERGE(A, false(B))
       return subtree(condition(B), STT, STF)
     end
else if external(B) then
   begin
     STT ← MERGE(true(A), B)
     STF ← MERGE(false(A), B)
     return subtree(condition(A), STT, STF)
   end
else if condition(A) = condition(B) then
   begin
     STT ← MERGE(true(A), true(B))
     STF ← MERGE(false(A), false(B))
     return subtree(condition(A), STT, STF)
   end
else if condition(A) > condition(B) then
   begin
     STT ← MERGE(A, true(B))
     STF ← MERGE(A, false(B))
     return subtree(condition(B), STT, STF)
   end
else (condition(A) < condition(B))
   return MERGE(B, A)
```

Order based on frequencies

- **(FR)** We order conditions according to their frequency in rules. The most occurring conditions are merged at top of the global tree, hence reducing its size.

Orders based on probability of evaluation

We can use an ordering that puts at the top conditions having a high probability of being evaluated at least once for each record. We calculate these probabilities as follows:

Let us consider two rules and a condition c. If p_i is the probability for the condition c to be evaluated for each rule i (assuming the rule is active) and if q_i is the probability for each rule i to be active, the probability for the condition c to be evaluated at least once in rule i for each record is $q_i \times p_i$. Thus, the global probability for the condition c to be evaluated at least once for two rules is:

$$(q_1 \times p_1) \cup (q_2 \times p_2) = q_1 \times p_1 + q_2 \times p_2 - q_1 \times q_2 \times p_1 \times p_2.$$

And for n rules we have:

$$\cup_{i=1}^{n}(q_i \times p_i) = \cup_{i=1}^{n-1}(q_i \times p_i) + (1 - \cup_{i=1}^{n-1}(q_i \times p_i)) \times q_n \times p_n$$

To calculate each p_i of the condition c, we assign to all conditions c' present in the local trees a probability $t(c')$ to be true and a probability $f(c')$ $(1 - t(c'))$ to be false independently of its position in the tree. Then, for each local tree, we apply Algorithm 6.5.

We propose the following orders based on the evaluation probabilities:

- **(PE)** The probabilities $t(c)$ and $f(c)$ for each condition c are defined arbitrarily and it is assumed that all rules i are active ($q_i = 1$), resulting in a simplified formula to calculate the evaluation probabilities:

$$\cup_{i=1}^{n} p_i = \cup_{i=1}^{n-1} p_i + (1 - \cup_{i=1}^{n-1} p_i) \times p_n$$

- **(PE')** The same as PE but we assume that non permanent rules i are active with probability 0.1 ($q_i = 0.1$)

- **(TP)** The same as PE but the probabilities p_i and q_i for each rule i are "actual" probabilities. The probability q_i for each rule i as well as the $t(c)$ and $f(c)$ probabilities for each condition c are estimated experimentally with respect to the given trace to analyze (assuming the analysis is off-line). The calculated p_i are then "actual" too. This ordering is not very practicable since it requires to perform preliminary analyses. However we introduce it to check whether PE is "close to reality" when we discuss our experimental results in Section 6.3.5.

Algorithm 6.5 calculate_proba(T, c)

Precondition:

- T is a local decision tree
- c is a condition

Postcondition:

- returns the probability for c to be evaluated when T is evaluated

```
if external(T) then
   return 0
else if condition(T) = c
   return 1;
else
   return t(condition(T))× calculate_proba(true(T), c)
                  + f(condition(T))× calculate_proba(false(T), c)
```

Orders based on the number of evaluations

Another ordering consists of giving a higher priority to the most evaluated conditions for each record and thus putting them at top of the global tree. We calculate thus an estimation of the average number of evaluations of every condition c for each ASAX cycle. This is estimated by the following formula:

$$\sum_{i=1}^{n} q_i \times p_i$$

By applying a sum of all probabilities to each condition c to evaluated in rule i, we obtain an order that is closer to the order based on frequencies than the one based on probabilities to be evaluated at most one. Thus, we expect to have a smaller global tree.

We propose the following orders based on the number of evaluations:

- **(AE)** We order conditions on the average number of evaluations, assuming all rules are active.
- **(AE')** The same as AE but we assume that non permanent rules are active with probability 0.1.
- **(TA)** As AE but with "actual" probabilities for conditions to be true and for rules to be active.

Orders based on both frequency and probability of evaluation

We can also make a compromise between basing the order on condition frequencies in rules and basing the order on their probabilities of evaluation.

The frequency of each condition is multiplied by its probability of evaluation. We have thus the following orders:

- **(FP)** We use FR × PE.
- **(FP')** We use FR × PE'.
- **(FTP)** We use FR × TP.

6.3.5 Experimental measures

We have implemented sorted trees in ASAX as well as all orders we defined in the previous section. In order to calculate the orders based on arbitrary probabilities, we have defined the probability of being true ($t(c)$) of each kind of condition as follows:

- The probability 0.1 is assigned to conditions testing an *equality*.
- The probability 0.9 is assigned to conditions testing a *disequality*.
- The probability 0.5 is assigned to conditions testing an *inequality*.
- The probability 0.9 is assigned to conditions testing the presence of a field.

Before discussing our experimental results, additional information about the used benchmark has to be provided.

Description of the rules

Table 6.2 extends Table 2.1 with additional information: The column NN provides the number of nodes for the local decision trees extracted from the rule. The column AP gives the probability for the rule to be active measured on the specific trace used for the experiments. Stateless rules have all probability 1. The remaining columns present estimations for execution probability and selftrigger probability. These estimations are computed with the help of an algorithm similar to Algorithm 6.5. We use two different manners to estimate them. First, we use the arbitrary probabilities of being true of the conditions. This allows us to estimate the probability that the rule will be simply self-triggered (column SP), executed (column EP) or discarded. No column is depicted for the last case, since, in the benchmark, very few rules are discarded. Second, we have computed the actual probability for each condition to be true or false in the audit trail and we have recomputed the probability for self-triggering (column SPT) and execution (column EPT). It can be observed that the assignment of arbitrary probabilities gives results very similar to the "actual" estimation. The probability of being simply self-triggered is very high (close to 1) for almost all rules.

RULE	MODULE	S	NN	AP	SP	EP	SPT	EPT
create	restricted_dir_write	N	6	1	0.983	0.017	1	0
mod_perm	mail_spool_hack	N	9	1	0.999	0.999	1	0
exec	execute_exit	N	5	1	0.829	0.171	0.993	0.007
exit		N	3	1	0.9	0.1	0.992	0.008
mod_perm	enable_suid_sgid	N	7	1	0.999	0.001	1	0
write	non_owner_write	N	8	1	0.875	0.125	0.999	0.001
write	restricted_write	N	6	1	0.983	0.017	1	0
read	restricted_read	N	8	1	0.969	0.031	1	0
access1	secret_unauth	N	9	1	0.997	0.003	1	0
access2		N	7	1	0.998	0.002	1	0
access3		N	6	1	0.999	0.001	1	0
write	system_program_write	N	9	1	0.999	0.001	0.999	0.001
exec	suid_shell_script	N	10	1	0.999	0.001	1	0
delete	unauth_delete	N	6	1	0.927	0.073	0.999	0.001
access1	secret_remote	N	12	1	0.999	0.001	1	0
access2		N	10	1	0.999	0.001	1	0
access3		N	9	1	0.999	0.001	1	0
exec1	buf_overflow	N	10	1	0.999	0.001	1	0
exec	lpd_delete	N	6	1	0.983	0.017	1	0
read		Y	13	0.355	0.999	0.001	1	0
delete		Y	9	0	0.999	0.001	0.999	0.001
hardlink	sh_minus_hack	N	10	1	0.999	0.001	1	0
exec		Y	9	0	0.998	0.002	1	0
exec1	buf_to_root_shell	N	8	1	0.998	0.002	0.999	0.001
exec2		Y	16	0.019	0.987	0.002	0.997	0.001
create_file	ftp_write	N	8	1	0.875	0.125	0.999	0.001
login		Y	6	0	0.983	0.017	0.999	0.001
read_rhosts		Y	16	0	0.987	0.013	0.997	0.003
exec1	suid_shell	N	9	1	0.999	0.001	1	0
read		Y	11	0	0.749	0.251	1	0
exec2		Y	15	0	0.988	0.002	0.999	0

Table 6.2: Additional information about the rules of the Darpa benchmark

The fact that those probabilities are exactly equal to 1 for many rules in column SPT in fact means that the corresponding attack does not occur in this particular audit trail. We refer to those figures later on when discussing the efficiency results.

Description of the conditions

Table 6.3 describes the conditions found in the local trees. Non parametric conditions are given first. Parametric conditions are at the bottom. The column Nr is the order of appearance of the condition for the merging algorithm. The column AF is the audit field to which the condition applies. The column O shows the operator used by the condition. Equality operators are different for integers (=) and for strings ($Str =$). The operator *present* checks if a given event field is present within the current event record. The rest of the columns provides information to define the orders depicted in Section 6.3.4. The column FR gives the number of rules in which the condition occurs. The column EP depicts the (artificial) probability for a condition to be evaluated at least once at each event record evaluation. The average number of evaluations at each record evaluation is presented in column AE, assuming that all rules are active. Columns EP' and AE' are similar to AE and EP but it is assumed that non permanent rules are active with probability 0.1. The column PT contains the probability of the condition to be true in the benchmark trace. Columns EPT and AET are similar to EP' and AE' but computed with probabilities PT and probability AP in table 6.2.

Results

Table 6.4 shows that using sorted trees is still unpractical for the same reasons as for the naive merging. We recall that the columns CT and GN provide the merging time (in seconds) and the number of nodes for the global tree up to the rule given in the first column, for the naive merging. In the six other columns we provide the corresponding figures for global trees sorted according to the orders FR, AE and PE. Obviously, the merging time is still exploding with sorted trees. We observe that, for trees of similar sizes, the merging time is lower for sorted trees than for unsorted trees. However, merging several local trees takes more time because sorted trees are bigger.

Because of the exponential growing size of the global tree, the merging algorithm is not scalable. The next two sections address this issue by providing two additional optimization techniques to reduce the size of the global tree.

Nr	AF	O	FR	EP	AE	EP'	AE'	PT	EPT	AET
3	Event ID	=	7	1	6.3	1	6.3	0.001	1	6.999
4	Event ID	=	7	1	7	1	7	0.001	1	7
11	Event ID	=	2	1	2	1	2	0	1	2
12	Event ID	=	10	1	9	0.999	5.76	0.008	1	6.019
13	Event ID	=	10	1	10	1	6.4	0	1	6.019
14	Event ID	=	4	1	3.993	1	1.299	0.008	1	1.019
21	Event ID	=	6	0.999	4.374	0.984	2.406	0	0.995	2.733
22	Event ID	=	6	0.999	4.86	0.995	2.673	0	0.995	2.733
23	Event ID	=	6	0.999	5.4	0.999	2.97	0	0.995	2.733
24	Event ID	=	6	1	6	1	3.3	0.185	1	3.355
27	Event ID	=	4	1	4	1	3.1	0.003	1	3
44	Event ID	=	1	1	1	1	1	0.001	1	1
1	Path	=	1	0.171	0.171	0.171	0.171	0	0.001	0.001
10	Path	=	1	0.1	0.1	0.1	0.1	0	0	0
19	Path	=	1	0.171	0.171	0.171	0.171	0	0.001	0.001
20	Path	=	1	0.31	0.31	0.31	0.31		0.168	0.168
25	Path	=	6	0.061	0.063	0.061	0.063	0	0.17	0.171
28	Path	=	1	0.014	0.014	0.014	0.014	0.137	0.001	0.001
32	Path	=	3	0.026	0.026	0.026	0.026	0	0.001	0.001
38	Path	=	1	0.171	0.171	0.171	0.171	0.001	0.007	0.007
39	Path	=	1	0.003	0.003	0.001	0.001	0.001	0.001	0.001
42	Path	=	1	0.001	0.001	0.001	0.001	0.001	0	0
47	Path	=	1	0.015	0.015	0.015	0.015	0.999	0.001	0.001
49	Path	=	1	0.014	0.014	0.001	0.001	0.001	0.001	0.001
50	Path	=	1	0.015	0.015	0.002	0.002	0.001	0.001	0.001
51	Path	=	1	0.171	0.171	0.017	0.017	0.001	0	0
54	Path	=	1	0.015	0.015	0.002	0.002	0.001	0	0
26	Audit User ID	=	3	0.479	0.571	0.479	0.571	0.997	0.171	0.171
34	Audit User ID	=	3	0.056	0.057	0.056	0.057	0.003	0	0
40	User ID	=	2	0.036	0.036	0.004	0.004	0.822	0.001	0.001
5	File Mode	=	1	0.001	0.001	0.001	0.001	0.243	0	0
6	File Mode	=	1	0.001	0.001	0.001	0.001	0.456	0	0
7	File Mode	=	1	0.001	0.001	0.001	0.001	0.124	0	0
8	File Mode	=	1	0.009	0.009	0.009	0.009	0.454	0	0
15	File Mode	=	1	0.008	0.008	0.008	0.008	0.001	0	0
16	File Mode	=	6	0.577	0.797	0.577	0.797	0.002	0.024	0.024
29	File Mode	=	2	0.139	0.139	0.139	0.139	0.242	0.001	0.001
43	File Mode	=	1	0.008	0.008	0.008	0.008	0.242	0.001	0.001
36	Exec Args	=	1	0.015	0.015	0.015	0.015	0.001	0	0
30	Audit User ID	!=	1	0.154	0.154	0.154	0.154	0.997	0.001	0.001
17	User ID	!=	3	0.344	0.391	0.344	0.391	0.178	0.003	0.003
33	User ID/ Audit User ID	!=	7	0.563	0.732	0.261	0.284	0.836	0.008	0.008
18	User ID/Owner User ID	!=	1	0.154	0.154	0.154	0.154	0.379	0.001	0.001
2	Return Value	!=	29	0.999	5.893	0.991	4.191	0.905	0.524	0.682
31	Owner User ID/Audit User ID	!=	6	0.598	0.814	0.385	0.455	0.902	0.063	0.063
37	Exec Args	>	1	0.029	0.029	0.029	0.029	0.001	0.001	0.001
9	Attr Field	present	9	0.669	1.025	0.66	0.997	0.834	0.02	0.02
45	User ID	=	1	0.017	0.017	0.002	0.002	0	0	0
41	PID	=	3	0.403	0.459	0.045	0.046	0.001	0.054	0.054
48	PID	=	2	0.536	0.632	0.062	0.063	0.382	0.001	0.001
55	PID	=	1	0.154	0.154	0.015	0.015	0	0	0
52	File Inode ID	=	1	0.028	0.028	0.003	0.003	0	0	0
53	User ID	!=	1	0.279	0.279	0.028	0.028	0	0	0
46	Path	Str=	1	0.171	0.171	0.017	0.017	0	0	0
35	Machine ID	Str=	3	0.479	0.571	0.479	0.571	0	0.171	0.171

Table 6.3: Conditions

RULE	CT	GN	CTFR	GNFR	CTAE	GNAE	CSPE	GNPE
create	0.02	6	0.04	6	0.02	6	0.01	6
mod_perm	0.02	22	0.04	27	0.02	25	0.01	27
exec	0.02	54	0.04	54	0.03	53	0.01	61
exit	0.02	70	0.04	103	0.03	101	0.01	123
mod_perm	0.03	158	0.04	195	0.03	191	0.02	211
write	0.05	293	0.06	399	0.05	379	0.03	399
write	0.09	423	0.09	607	0.08	723	0.06	743
read	0.63	1,202	0.77	1,403	0.7	1,487	0.7	1,575
access1	3.41	1,970	3.31	2,355	3.47	3,211	2.46	3,307
access2	7.34	2,530	9.94	2,967	10.23	4,223	9.64	4,335
access3	39.77	3,430	39.06	4,643	51.56	7,123	53.76	7,383
write	133.69	8,550	213.08	11,419	414.45	16,483	475.16	16,743

Table 6.4: Compilation times of sorted trees

6.4 Sequential trees

To avoid size explosion of global trees, we observe, in our benchmark, that it can be the case that many conditions appear only in a single local tree or in a small number of local trees. If two local trees share no common conditions, merging them generates a high number of condition duplications. Moreover, evaluating them separately is as efficient as evaluating a merge of the two. Thus, we can avoid the costly merging process. *Sequential trees* generalize this idea to **all subtrees of the global tree**.

In this section, we apply the concept above on global trees and all their subtrees. Two trees are independent when they do not share common conditions. Merging two independent trees results in a sequential tree. Figure 6.5 depicts the sequential tree corresponding to our running example. Its size is the sum of the sizes of the local trees plus an extra node for the sequence (not shown on figure). Obviously, this example is trivial since there is no shared common decision. Figure 6.6 shows another example involving two trees sharing a common decision. The resulted merged tree illustrates well that the concept of sequentialization is applied on all subtrees.

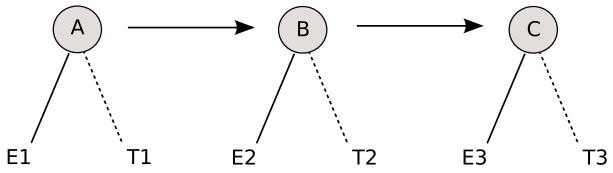


Figure 6.5: A sequential global tree

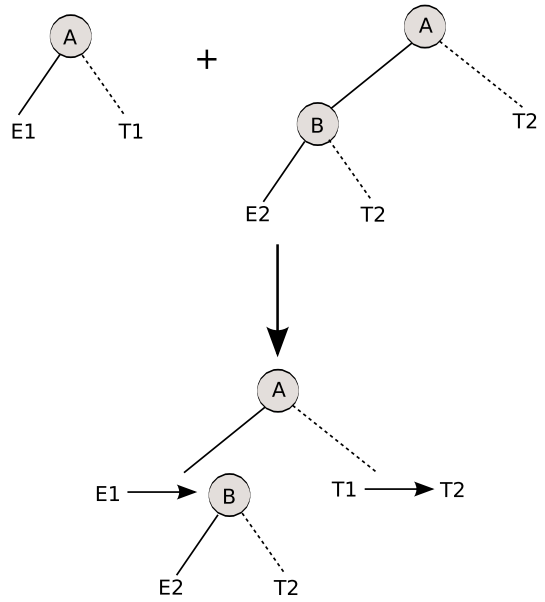


Figure 6.6: Sequentializing the merge of two trees sharing a common condition

6.4.1 Independent trees

Determining efficiently if two trees are independent, i.e., do not contain common conditions, is not trivial. We can achieve it by associating each subtree with a sorted list of its conditions. Whenever two subtrees are merged, their lists are scanned in parallel for common conditions. If the scan fails, the subtrees are *sequentialized*. Otherwise they are merged as sorted trees. This approach is rather "time consuming" since it entails to scan many conditions each time subtrees are merged.

A more efficient but less accurate solution is to associate each subtree with its lowest condition as well as its greatest. The lowest is actually the root. Two trees are independent if one of them has a greatest condition lower than the smallest condition of the other. This approach is less accurate because it identifies independent trees only if the intervals formed by the lowest and greatest conditions of both trees do not overlap. For example, Figure 6.7 shows two identified independent trees. The first tree forms the interval [1,3] while the second tree forms the interval [4,6]. The trees are identified as independent because the greatest condition of the first tree (3) is lower than the lowest condition of the second tree (4). In Figure 6.8, the two formed intervals are [1,4] and [3,6] and overlap. As a result, none of the greatest conditions is lower than the lowest of the other tree and we cannot identify whether they are independent. It is reasonable to assume that most situations fit the second example and, thus, we discard this solution.

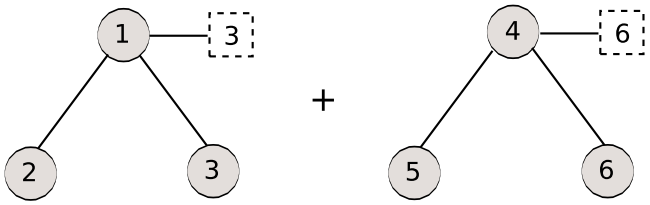


Figure 6.7: Identified independent trees

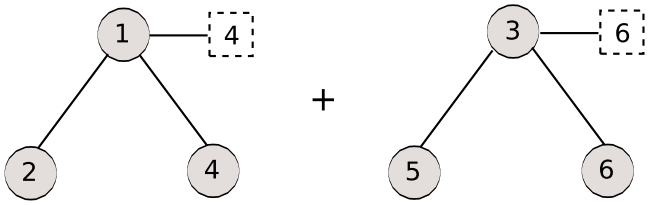


Figure 6.8: Unidentified independent trees

The solution we choose does not need additional data structures to determine if two trees are independent. There is a merge algorithm that directly recognizes independent trees during the bottom-up construction of the merged tree. This is explained in details in the next subsection.

6.4.2 Definition of sequential trees

We define *sequential trees* as follows:

A sequential tree is:

- either a decision leaf.
- or a rooted tree of the form $root(C, At, Af)$ where At and Af are sequential trees.
- or a tree sequence, of at least two elements, noted $A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_n$, where $\forall i : 1 \leq i \leq n : A_i$ is a decision or a rooted tree.

Notice that the position of a tree in a tree sequence is irrelevant as tree sequences in fact represents tree sets. Therefore we consider that all permutations of a tree sequence are equals.

The notation '$\Rightarrow$' is also used to decompose a tree sequence into an element randomly picked and the rest of the sequence. For example, $T = X \Rightarrow Y$ decomposes the sequential tree T composed of a tree sequence into the single tree (decision or rooted) X and a sequential tree Y composed of the rest of the sequence.

6.4.3 Merging sequential trees

The Algorithm 6.6 is used to merge sequential trees. In this section we provide a proof a correctness of the algorithm at the same time as explaining its mechanism.

Before proving the correctness of the merging algorithm, we must make the definition of $sequentialize(A, B)$ (used in the algorithm) fully precise. In full generality, we can write $A = A_1 \Rightarrow \ldots \Rightarrow A_m$ and $B = B_1 \Rightarrow \ldots \Rightarrow B_n$ where all A_i and B_i are either decisions or rooted trees. (In particular, if A is a decision or a rooted tree, we have $m = 1$ and $A = A_1$. A similar remark holds for B.). With these conventions, $sequentialize(A, B)$ simply returns the tree sequence $A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow B_1 \Rightarrow \ldots \Rightarrow B_n$.

It has to be noted that the algorithm uses a new function: $sequential(T)$, which returns $true$ if T is a sequential tree composed of a tree sequence, $false$ otherwise.

Proof of Algorithm 6.6

Properties to prove. To prove the correctness of Algorithm 6.6, we have to prove the following statements.

1. The algorithm terminates.
2. It returns a well-formed sequential tree GT.
3. GT is equal to $sequentialize(A, B)$ if and only if A and B do not share any condition.
4. Evaluating the sequential tree GT, with respect to any environment, returns the same set of decisions, as evaluating A and B, separately.
5. GT is "optimal" in the sense that, during any evaluation of it, no condition is evaluated more than once, and no decision is taken more than once.
6. GT is "minimal" in the sense that, if we successively merge, in any order, any number of sequential trees $A_1, \ldots, A_n$, that are all decisions or rooted trees, the final result is always the same, i.e., a tree sequence $X_1 \Rightarrow X_2 \ldots \Rightarrow X_m$ such that there is a partition $P_1, \ldots, P_m$ of $\{A_1, \ldots, A_n\}$ such that two sequential trees belonging to two different sets P_i and P_j share no common condition and that no P_i can be divided into two smaller sets without losing this property.

The proof uses the assumption that A and B both enjoy Properties 5 and 6. Moreover it must be assumed that A and B do not contain any common decision (in fact, no decision relative to the same rule). The proof proceeds by structural induction on A. We use the following well-founded relation:

Algorithm 6.6 MERGE(A, B) - Merging sequential trees

Precondition:

- A and B are sequential trees.

Postcondition:

- The algorithm returns GT, the merging of A and B (see specification of Algorithm 6.1).
- Each condition in GT is bigger than its ancestors (with respect to the defined order on the conditions).
- If A and B are independent, then GT is a sequential tree, i.e., the sequentialization of A and B.

```
if external(A) or external(B) then
   return sequentialize(A, B)
else if sequential(A) then
   Let X⇒Y the decomposition of A where X is not sequential and Y might
be sequential.
   if MERGE(X, B) = X⇒B then
      return sequentialize(X, MERGE(Y, B))
   else return MERGE(MERGE(X, B), Y)
else if sequential(B) then
   return MERGE(B, A)
else if condition(A) = condition(B) then
   begin
      STT ← MERGE(true(A), true(B))
      STF ← MERGE(false(A), false(B))
      return subtree(condition(A), STT, STF)
   end
else if condition(A) > condition(B) then
   begin
      ABl ← MERGE(A, true(B))
      ABr ← MERGE(A, false(B))
      if ABl = A⇒true(B) and ABr = A⇒false(B)
         return sequentialize(A,B)
      else
         return subtree(condition(B), ABl, ABr)
   end
else (condition(A) < condition(B))
   return MERGE(B, A)
```

> A' **is smaller than** A either if A' is a strict part of (is strictly included in) A, or if A' and A are two tree sequences and A' is shorter than A, or if A' and A both are rooted binary trees such that the condition at the root of A' is strictly greater than the condition at the root of A.

We extend this well-founded relation into the following lexicographic relation:

> $< A', B' >$ **is smaller than** $< A, B >$ if A' is smaller than A or if $A' = A$ and B' is smaller than B.

The correctness proof.

1. If A (or B) simply is a decision, it is obvious that all properties 1 to 6 hold if we choose $GT = \mathit{sequentialize}(A, B)$. (We only have to check each condition.)

2. Let us assume that both A and B are rooted trees. Assume that C is the condition at the root of A.

   - C is smaller than the condition at the root of B. Let $A = \mathit{root}(C, At, Af)$. We can assume that the algorithm is correct for At and Af. Let $ABl = \mathit{merge}(At, B)$ and $ABr = \mathit{merge}(Af, B)$.
     - Assume that $ABl = \mathit{sequentialize}(At, B)$ and $ABr = \mathit{sequentialize}(Af, B)$. By Property 3, it means that neither At nor Af share a condition with B. Hence, A does not. Thus, $\mathit{sequentialize}(A, B)$ is a correct result, since it enjoys Properties 3, 4, 5 and 6.
     - Assume that $ABl \neq \mathit{sequentialize}(At, B)$ or $ABr \neq \mathit{sequentialize}(Af, B)$. By Property 3, either At or Af share a condition with B. Hence, A also does. Thus, $GT = \mathit{subtree}(C, ABl, ABr)$ fulfills Properties 3 to 6. This is rather obvious for Properties 3, 4 and 6. Let us check Property 5. Assume that GT is not optimal. Since, obviously, C is evaluated only once in GT, some conditions should be evaluated twice or more in ABl or ABr. But this is impossible by the induction hypothesis.
   - The same condition C is at the root of A and B. Let $A = \mathit{root}(C, At, Af)$ and $B = \mathit{root}(C, Bt, Bf)$. By induction hypothesis, the algorithm is correct for the pairs (At, Bt) and (Af, Bf). Let $GT = \mathit{subtree}(C, \mathit{merge}(At, Bt), \mathit{merge}(Af, Bf))$. Property 3. clearly holds for GT since A and B share the condition C. Property 4. holds by construction of GT and by the induction

hypothesis. Property 5. holds because C is evaluated only once in GT and by the induction hypothesis. Property 6 holds for the tree sequences contained in GT by induction hypothesis.

- C is greater than the condition at the root of B. Then, B is greater than A according to the chosen well-founded relation. Hence, $merge(B, A)$ is a correct result, by induction hypothesis, using the lexicographic well-founded ordering.

3. Let us assume that A is a rooted tree and that B is a tree sequence. Let X be the first sequential tree in the sequence and let Y be the remaining part of the sequence. X is shorter than B. Thus, we can use the induction hypothesis for the pair $< A, X >$.

   - Assume that $merge(A, X) = sequentialize(A, X)$. It means that A and X share no condition by Property 3. We can also compute $merge(A, Y)$, by induction hypothesis. Then, clearly, $GT = sequentialize(X, merge(A, Y))$ enjoys Property 3 since X has no condition in common with A and Y. Evaluating GT brings the same decisions as evaluating, X, A, and Y separately. Thus Property 4. holds. No condition is evaluated more than once in GT since X share no condition with $merge(A, Y)$ and by induction hypothesis. Hence Property 5 holds. Property 6 holds by induction hypothesis applied to $merge(A, Y)$.
   - Let us now assume that $merge(A, X) \neq sequentialize(A, X)$. Then, it must be a rooted tree (see B.). By Property 3, A and X share a common condition. Thus, obviously $merge(A, B)$ must be equal to $merge(merge(A, X), Y)$ and all required properties hold by induction.

4. If B is a rooted tree and A is a tree sequence, we just swap A and B. Then, the same reasoning applies.

5. The last case arises when both A and B are tree sequences. Let X be the first sequential tree of A and let Y be the remaining tree sequence. The algorithm returns $GT = merge(X, merge(Y, B))$. Since X and Y both are smaller than A, GT is well-defined (the induction hypothesis can be used). We observe that $GT = sequentialize(A, B)$ if and only if A and B does not share any common condition since, in that case, $merge(X, merge(Y, B))$

   $= merge(X, sequentialize(Y, B))$

   $= sequentialize(X, sequentialize(Y, B))$

$$= sequentialize(sequentialize(X, Y), B)$$

$$= sequentialize(A, B).$$

Properties 4, 5, and 6 hold (mainly) by induction hypothesis.

6.4.4 Evaluating sequential trees

We must modify the evaluation algorithm to handle sequential trees. The result is presented in Algorithm 6.7. It uses the new function $subtrees(T)$, which returns the tree sequence of the sequential tree T assuming the latter is a sequence. Every subtree is evaluated separately and the results form the decision set to return. Since the algorithm visits a leaf for each single rule to form the decision set, it has complexity $O(c + r)$ where c is the number of distinct conditions present in the tree and r is the number of rules whose local tree has been merged into the global tree.

Algorithm 6.7 evaluate_decision_tree(T) - Evaluation of a sequential tree

Precondition:

- T is a sequential tree

Postcondition:

- The algorithm returns a set composed of (rule, decision) couples with respect to the current environment. Each element of the set corresponds to a distinct rule whose local tree has been merged in the global tree.

```
if leaf(T) then
   return {decision(T)}
else if sequential(T) then
   begin
      res ← {}
      for all ST in subtrees(T) do
         res ← res ∪ evaluate_decision_tree(ST)
      return res
   end
else
   begin
      if eval_condition(condition(T)) = true
         return evaluate_decision_tree(true(T))
      else
         return evaluate_decision_tree(false(T))
   end
```

6.4.5 Implementation details

To implement sequential trees, we introduce a new box to represent a tree sequence. It is called *sequential box*, has no successor (actually, the value of its α pointer is always *nil*) and contains a simply-linked list holding the subtrees of the sequence. Figure 6.9 depicts the structure of the box.

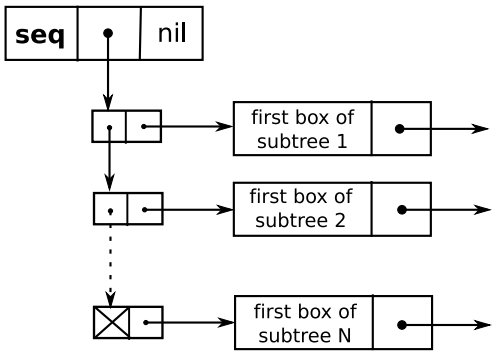


Figure 6.9: The sequential box

6.4.6 Experimental measures

We have implemented sequential trees in ASAX and we are now able to compile global decision trees for our Darpa benchmark in more decent time. In addition, to assess the improvement provided by the optimization, we have the opportunity to test the different orderings we defined in Section 6.3.5.

We now provide the results of our experiments in Table 6.5. The column OPTIM specifies the chosen ordering. The column CT provides the compilation time, including parsing RUSSEL rules, extracting local trees, sorting local trees, and merging sorted local trees. We do not provide separate timings for the intermediate steps since the first three are almost negligible. The column ET is the total evaluation time for the benchmark, excluding reading the BSM file and translating records into NADF format. The column LET depicts the evaluation time based on sorted local trees evaluated sequentially, one after the other (note that only trees of active rules are evaluated). The column GN provides the number of nodes of the global tree. Between parentheses are the numbers of conditions, of tree sequences, and of decision nodes, respectively. Common subtrees are cached and they are counted only once. The column LN gives the number of nodes of all sorted local trees. The column ALE shows the average number of conditions that are actually evaluated when sorted local trees are considered in turn. The column AGE is the average number of evaluated conditions when the global tree is used. The column AUE gives the average number of conditions that

are uselessly evaluated in the global tree, i.e., they are not evaluated when local (not sorted) trees are used to take the decisions. The line STB stands for the implementation of ASAX which involves selftrigger boxes only and the line LOC gives the results when (unsorted) local trees are used instead of the global tree. Note that this version of local trees is not reorganized with parametric conditions at the bottom. The other lines provide the results for one of the orders defined previously.

We see that the merging time is dramatically improved. Notice that many subtrees are sequentialized which is the main explanation to the facts that the overall number of nodes is reduced and the merging times made shorter. Using the local trees to decide if the rules must be executed or self-triggered provides a speed-up of 3.95 (see Chapter 5). If we use the global tree, we get an additional speed up of 1.95 in the best case.

The evaluation times are similar for all orders except for FR, which is slower. This can be explained by the fact that FR modifies deeply the organization of the local trees and thus entails many useless evaluations (5.49 in average). The global trees have very different sizes and, in general, the merging times are very related to this size. It is worth noticing that the orders involving condition frequencies (FR, FP and FP') entail the smallest global trees. This proves that putting conditions that occur in many local trees at the top encourages a high factorization rate.

Moreover, by factorizing as most as possible at the top of the global tree, the merging algorithm quickly arrives at the situation sequentializing two independent subtrees, which is the main factor of the reduction of the node number. On the contrary, putting conditions which have a high probability to be evaluated at the top entails large global decision trees. In fact, we observe that local trees sorted by PE are the biggest ones. It results in having the biggest merged global tree also. However, using that order produces very few useless evaluations. This explains a better evaluation time.

Orders FR and PE are thus two extremes; the first one is best at compilation while the second one is the best at evaluation. Orders AE and FP intend to mix the advantages of both previous orders. We observe that, while providing similar evaluation times as PE, they exhibit reduced compilation times. Moreover, FP seems to be the best order regarding both its compilation time and the size of its corresponding global tree. Its evaluation time is a bit slower than PE but it is negligible.

About the orders which assume that non permanent rules are active with probability 0.1, i.e. PE', AE' and FP', we observe that their compilation times are a bit slower than PE, AE and FP respectively. The evaluation times from the orders based on "actual" probabilities (TP, TA, FTP) are very similar to the orders based on arbitrary probabilities.

A few general interesting observations can be made. First, if we compare the column LET with the column ET or the column ALE with the column AGE, we see the relevance of achieving factorization; the number of condi-

OPTIM	CT	ET	LET	GN	LN	ALE	AGE	AUE
STB	0	26.15	-	-	-	-	-	-
LOC	0.01	6.67	6.67	-	276	56.09	-	-
FR	0.36	4.63	11.84	539(243/207/89)	286	76.05	15.59	5.49
PE	0.76	3.42	6.78	2,886(1,315/1482/89)	295	52.01	12.68	0.36
AE	0.43	3.5	7.73	1,173(475/609/89)	293	59.62	13.72	1.9
FP	0.25	3.55	9.47	498(211/198/89)	287	72.19	13.16	2.79
PE'	0.84	3.6	7.04	2,900(1,338/1,473/89)	302	54.83	13.41	1.16
AE'	0.59	3.49	7.72	1,332(535/708/89)	287	58.45	12.98	1.16
FP'	0.26	3.51	9.34	624(246/289/89)	287	72.17	13.16	2.79
TP	0.72	3.51	6.71	2,269(1,008/1,172/89)	295	52.01	12.68	0.37
TA	0.76	3.66	6.69	1,879(751/1,039/89)	292	52.01	12.68	0.37
FTP	0.59	3.56	9.4	1,458(574/795 /89)	289	69.51	12.43	1.17

Table 6.5: Compilation and evaluation results of sequential trees

tions evaluated in the global tree is approximately five times less than for local sorted trees. Second, some orders entail less useless evaluations than others but they do not exhibit better evaluation times. In fact, the global tree for such orders is composed of far more tree sequences and evaluating a tree sequence, i.e., sequential box generates a non neglibible overhead. The time gained from evaluating less useless conditions is spent in that overhead. A well-representative example of this situation is provided by orders AE and FP.

6.4.7 Excluding conditions

In some situations, we would like to merge non critical conditions faster. Assuming we have chosen an order for the conditions with more significant conditions first, we can exclude all conditions beyond a given position from the merging process. A subtree with only excluded conditions is then considered as a single leaf and it is always inserted in a tree sequence during the merging process. We call such subtrees *excluded subtrees*.

Algorithm 6.8 depicts the modification of the merging algorithm to handle this feature. It uses the function $excluded(T)$, which returns *true* if the tree T contains excluded conditions only. Of course, with this approach, some conditions may be evaluated several times. Moreover, not sorting the excluded conditions inside the local trees allows us to preserve the original order between these conditions and thus to limit node duplication. Algorithm 6.9 presents the extended Algorithm 6.3 to sort local trees with excluded conditions.

Evaluating sequential trees with excluded conditions

If condition exclusion is used, some conditions are likely to be evaluated several times from one subtree to another. Thus, we make the complexity of

Algorithm 6.8 Merging sequential trees with excluded conditions

Precondition:

- A and B are sorted trees.

Postcondition:

- The algorithm returns GT, the merging of A and B (see specification of Algorithm 6.1).
- Each condition in GT is bigger than its ancestors (with respect to the defined order on the conditions)
- If A and B are independent, then GT is a sequential tree, i.e., the sequentialization of A and B.
- If A or B is an excluded subtree, then GT is a sequential tree, i.e., the sequentialization of A and B.

```
if external(A) or external(B) then
   return sequentialize(A, B)
else if excluded(A) or excluded(B) then
   return sequentialize(A, B)
else ...
```

the evaluation algorithm more precise for that situation. The worst case is illustrated by the excluded conditions which are present in every rule. Since the maximum length for the tree sequence of a sequential tree is the number of merged rules, the algorithm has complexity $O(ec \times r + nec + r)$ where ec is the number of excluded distinct conditions and nec is the number of distinct conditions which are not excluded.

Experimental results

To assess the impact of condition exclusion, we have selected the order which entails the longest compilation time, i.e. PE. We have tested this order with various exclusion rates ranging from 10% to 100% (of the less priority conditions). The results are provided in Table 6.6. The column ER indicates the exclusion rate. The column RE depicts the average number of repetitive evaluations. We observe that the higher is the exclusion rate the faster is the compilation. We witness a drastic improvement starting from 80%. However, the evaluation time starts to increase from that point. This is explained by the fact that excluded subtrees become too big, hence many excluded conditions are evaluated several times. We see the fast evolution of the number of repetitite evaluations in column RE. Starting from 70%,

Algorithm 6.9 SORT_LOCAL_TREE(T) - Sort a local decision tree with excluded conditions

Precondition:

- T is a local decision tree

Postcondition:

- returns T' so that:
  - all paths of T' are of the form: $< c_1, \ldots, c_n, ec_1, \ldots, ec_m, d >$ where $c_1, \ldots, c_n$ are conditions (not excluded), $ec_1, \ldots, ec_m$ are excluded conditions and d is a decision.
  - $\forall i, j : 1 \leq i < j \leq n : c_i < c_j$
  - the evaluation of T' gives the same results as T .

```
if external(T) then
   return T
else if excluded(T)
   return T
else
   begin
     C ← get_smallest_condition(T)
       begin
         STT ← SORT_LOCAL_TREE(simplify(T, C, true))
         STF ← SORT_LOCAL_TREE(simplify(T, C, false))
         return subtree(C, STT, STF)
       end
   end
```

we observe that number of useless evaluated conditions decreases. This can be explained by the fact that excluded subtrees are not sorted and thus do not entail useless evaluations. It is worth noticing that excluding 100% of the conditions creates a sequentialization of the unsorted local decision trees. Since evaluating the global tree amounts thus to evaluate each (unsorted) local tree separately, the corresponding evaluation time is higher than evaluating (unsorted) local trees for active rules only (see Table 6.5).

Choosing the right percentage of condition to exclude is not obvious. The optimal value for our benchmark is somewhere between 70% and 80%. However it is likely to vary depending on the content of the rules. As a result, condition exclusion should be used wisely.

ER	CT	ET	GN	LN	ALE	AGE	AUE	RE
10%	0.72	3.43	2,735(1,254/1,392/89)	295	52.01	12.68	0.36	0
20%	0.7	3.45	2,735(1,254/1,392/89)	295	52.01	12.68	0.36	0
30%	0.64	3.43	2,741(1,254/1,398/89)	295	52.01	12.68	0.36	0.01
40%	0.62	3.46	2,709(1,250/1,370/89)	295	52.01	12.68	0.36	0.01
50%	0.49	3.42	2,736(1,250/1,397/89)	294	52.01	12.68	0.36	0.01
60%	0.38	3.46	2,633(1,214/1,330/89)	294	52.01	12.68	0.36	0.01
70%	0.15	3.48	2,239(1,079/1,070/90)	297	52.08	12.54	0.2	0.03
80%	0.02	4.45	272(165/16/91)	302	52.07	22.56	0.19	10.04
90%	0.01	6.56	264(168/9/87)	276	52.59	46.62	0.19	34.1
100%	0.01	8.28	277(189/1/87)	276	52.59	67.62	0.19	55.1

Table 6.6: Compilation and evaluation results of sequential trees with excluded conditions

6.5 Simplified trees

We observe in Table 6.3 that many conditions test the same NADF record field (e.g., the field containing the identifier of the event) and only one of them is true at the same time. Hence, assuming the conditions `EVENT = exec` and `EVENT = exit`, there is no need to insert the second condition in the "true" branch of the first one during the merging operation. We generalize this observation for all conditions whose value is determined by the value of another condition. We say that such conditions are *exclusive*. Another typical example is illustrated by the following conditions: `EVENT != exec` and `EVENT = exec`.

Many nodes of the global tree may contain conditions that are exclusive with at least one of their ancestors. The optimization we introduce in this section aims to simplify the exclusive conditions to reduce the size of the global tree significantly. Assuming that conditions A, B and C are exclusive in our running example, e.g., they cannot be true at the same time, Figure 6.10 shows the resulted *simplified tree*. Note that simplification is not related to sequentialization. In general, if all conditions are exclusive, they form a unique branch and the size of the global tree is $2 \times C + 1$ where C is the number of distinct conditions. So it is linear in the number of conditions.

In the remainder of this section we describe in details how to simplify exclusive conditions. We then modify the merging algorithm to achieve the simplification and we conclude by assessing our novel optimization with the Darpa benchmark.

6.5.1 Simplifying trees

Simplifying a tree consists of removing conditions whose evaluation is useless, regarding the taken branches from their ancestor conditions. The removed conditions are replaced either by their "true" subtree or by their

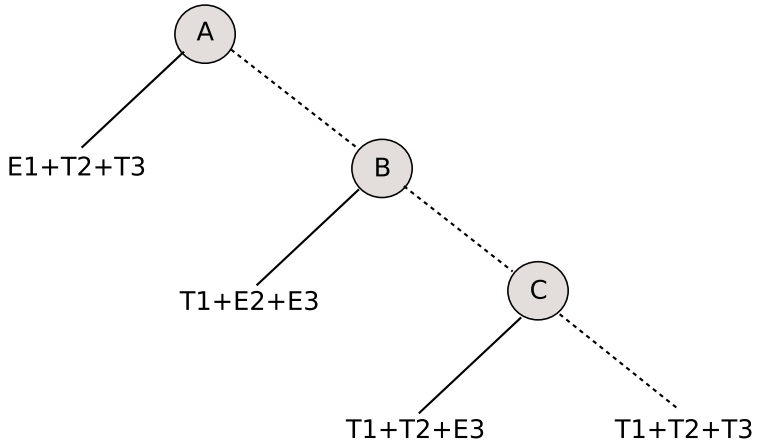


Figure 6.10: A simplified global tree

"false" subtree depending on their value. Our optimization only simplifies conditions involving NADF fields.

Generally, conditions involving NADF fields have one of the following form: either `present f` or `f op v` where f is a field name, v is a value and *op* is a relational operator =, !=, < or >. Let us assume two conditions A and B. Table 6.7 depicts all situations involving A and B that are subject to simplification. For each row, we indicate the form of conditions A and B and the value of B corresponding to the value of A. This table is used to simplify global trees. Assuming that A and B are located in the same branch and that their patterns match one of the situations depicted in the table, the following rule is applied to simplify trees: *if B is in the "A value" branch of A then B is simplified with its "B value" subtree.*

A	B	A value	B value
present f	f {=,<,>} a	false	false
	f != a	false	true
f = a	present f	true	true
	f = b ($b \neq a$)	true	false
	f != a	true	false
		false	true
	f != b ($b \neq a$)	true	true
f != a	f = a	true	false
		false	true
	f = b ($b \neq a$)	false	false
	f != b ($b \neq a$)	false	true

Table 6.7: Table of exclusive conditions

6.5.2 Merging simplified trees

We present now, in Algorithm 6.10, a modified version of our merging algorithm. It has been extended to merge simplified trees accordingly to the principle stated in Section 6.5.1. The simplification is performed when a condition is inserted inside the merged tree. In case the condition is exclusive with a condition present in the currently built branch of the merged tree, it is simplified. The algorithm uses the additional function $implicit\_value(C1, C2)$, which calculates the implicit value of condition $C1$ accordingly to the branch taken from condition $C2$, if $C1$ and $C2$ are exclusive. Checking for exclusive conditions is achieved in $O(c)$ time where c is the number of distinct conditions (this complexity is fully described in Section 6.5.3).

6.5.3 Implementation details

To allow a fast processing of exclusive conditions, the "exclusivity" between conditions is computed only once before merging. This process aims to attach a set of exclusive conditions, called *exclusivity set*, to every condition. Since ancestors of a condition are always lower thanks to sorted trees, there is no need to calculate "exclusivity" with greater conditions. To achieve the operation, conditions are treated in an ascending order (according to their defined ordering). For each condition, we scan the previous conditions for any "exclusivity" in a descending order. The scanning stops either at the first condition identified as exclusive or when all previous conditions have been tested. In the first situation, the condition whose exclusivity set is being calculated inherits the one of the encountered condition, in which the later is added. The second situation means that the treated condition has no exclusive conditions.

During the merging process, the algorithm tags the conditions belonging to the current branch being built. These conditions are then untagged when the algorithm backtracks to start a new branch. For each taged condition, we also retain which branch has been taken. This information is hold inside the condition descriptors. The first information allows us to easily identify exclusive conditions; before inserting a condition, the algorithm checks its exclusive set for a taged condition, i.e., an exclusive condition which is present in the current branch. If it is the case, a simplification has is achieved. The second information allows the function *implicit_value* to calculate the implicit value of the condition to simplify. Checking for an exclusive condition has complexity $O(c)$ where c in the number of distinct conditions. Indeed, the worst case is the situation involving a condition which is exclusive with most other conditions. We never observed such a case, however. So we think that it is reasonable to expect very few conditions with large exclusive sets. According to that fact, checking for an exclusive condition is achieved in a time closer to constant than linear.

Algorithm 6.10 MERGE(A, B) - Merging simplified trees

Precondition:

- A and B are sorted trees.

Postcondition:

- The algorithm returns GT, the merging of A and B (see specification of Algorithm 6.1).
- Each condition in GT is bigger than its ancestors (with respect to the defined order on the conditions)
- Every condition inside GT is not exclusive with one of its ancestors.

```
if external(A) or external(B) then
   return leaf(union(decision(A), decision(B)))
else if condition(A) = condition(B) then
   begin
     STT ← MERGE(true(A), true(B))
     STF ← MERGE(false(A), false(B))
     return subtree(condition(A), STT, STF)
   end
else if condition(A) > condition(B) then
   if condition(B) is exclusive with one of the conditions present in the
current built branch then
     Let Cond, the exclusive condition with condition(B)
     if implicit_value(condition(B), Cond) = true then
       return MERGE(A, true(B))
     else
       return MERGE(A, false(B))
   else
     begin
       STT ← MERGE(A, true(B))
       STF ← MERGE(A, false(B))
       return subtree(condition(B), STT, STF)
     end
else (condition(A) < condition(B))
   return MERGE(B, A)
```

OPTIM	CT	ET	LET	GN	LN	ALE	AGE	AUE
STB	0	26.15	-	-	-	-	-	-
LOC	0.01	6.67	6.67	-	276	56.09	-	-
FR	1.99	4.08	11.84	1,000(505/0/495)	286	76.05	15.01	5.48
PE	0.62	2.97	6.78	700(380/0/320)	295	52.01	11.82	0.36
AE	0.62	2.95	7.73	699(380/0/319)	293	59.62	12.91	1.89
FP	0.63	2.91	9.47	653(332/0/321)	287	72.19	12.42	2.79
PE'	0.61	3.03	7.03	690(370/0/320)	302	54.83	12.54	1.15
AE'	0.59	2.97	7.72	687(368/0/319)	287	58.45	12.24	1.16
FP'	0.61	2.95	9.34	647(326/0/321)	287	72.17	12.42	2.62
TP	0.67	2.88	6.71	739(375/0/364)	295	52.01	12.19	0.37
TA	0.67	2.92	6.69	739(375/0/364)	292	52.01	11.84	0.37
FTP	0.88	2.8	9.4	863(435/0/428)	289	69.51	11.68	1.17

Table 6.8: Compilation and evaluation results of simplified trees

6.5.4 Experimental measures

We now assess the improvement provided by simplified trees. We performed the same experiments as for sequential trees. Table 6.8 depicts the results. We can see that compilation times are all around 0.6 excepting for FR. Evaluation times are also all similar excepting for FR again. FR has still the lowest evaluation time for the same reason as for sequential trees, i.e., it entails the highest number of useless evaluations. Evaluation times are all a bit faster than those provided by sequential trees however (around 0.5 sec faster). We get here an additional speed-up of 2.38 with respect to local trees in the best case. This improvement is explained by the fact that exclusive conditions are not evaluated anymore. This is illustrated by the average numbers of global evaluations, which are lower than those entailed by sequential trees. It is also worth noticing that the average numbers of useless evaluations are nearly identical to those produced by sequential trees, proving that these figures mainly depends on the used ordering.

As for sequential trees, compilation times benefit significantly from simplified trees. However, the most rewarding improvement is the decrease of the evaluation times with respect to sequential trees. As a result, using both simplified and sequential trees together should provide optimal results.

6.6 Putting simplification and sequentialization together

6.6.1 Simplified sequential trees

We now combine the two techniques to benefit from their respective advantages, resulting in merging *simplified sequential trees*. Figure 6.11 depicts a simplified sequential tree with three exclusive conditions. More interestingly

Figure 6.12 presents a case where condition C has no dependencies with the others. It gives a better idea of how a simplified sequential tree looks like in general.

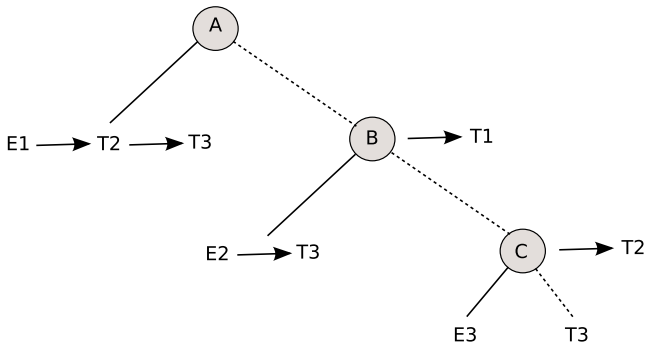


Figure 6.11: A simplified sequential tree with exclusive conditions

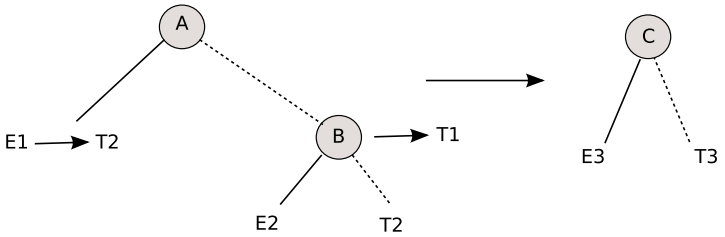


Figure 6.12: A simplified sequential tree with only two exclusive conditions

Algorithm 6.11 is the final version of the merging algorithm involving both sequential trees and simplified trees. It has to be noticed that excluded subtrees are not simplified when inserted. Since excluded conditions are expected to be insignificant and thus rarely evaluated, we think that evaluating exclusive conditions inside excluded subtrees is a negligible price to pay. As a result, we prefer not to simplify excluded subtrees to spare time during the merging operation.

6.6.2 Experimental measures

This time we present results for orders FR, PE, AE and FP only. The figures are depicted in Table 6.9. We observe that we have now almost negligible compilation times. This proves that combining both optimization techniques is the best solution to reduce global tree sizes drastically, which is obviously illustrated in column GN. Almost all global trees have less nodes than the set of local trees. Evaluation times are a bit slower than for simplified (non

Algorithm 6.11 MERGE(A, B) - Final Merging Algorithm

Precondition:

- A and B are sequential trees.

Postcondition:

- The algorithm returns GT, the merging of A and B.
- Each condition in GT is bigger than its ancestors (with respect to the defined order on the conditions)
- If A and B are independent, then GT is a sequential tree, i.e., the sequentialization of A and B.
- If A or B is an excluded subtree, then GT is a sequential tree, i.e., the sequentialization of A and B.
- Every condition inside GT is not exclusive with one of its ancestors.

```
if external(A) or external(B) then
   return sequentialize(A, B)
else if excluded(A) or excluded(B) then
   return sequentialize(A, B)
else if sequential(A) then
   Let X⇒Y the decomposition of A where X is not sequential and Y might be
sequential.
   if MERGE(X, B) = X⇒B then
     return sequentialize(X, MERGE(Y, B))
   else return MERGE(MERGE(X, B), Y)
else if sequential(B) then
   return MERGE(B, A)
else if condition(A) = condition(B) then
   begin
     STT ← MERGE(true(A), true(B))
     STF ← MERGE(false(A), false(B))
     return subtree(condition(A), STT, STF)
   end
else if condition(A) > condition(B) then
   if condition(B) is exclusive with one of the conditions in the current built branch
then
     Let Cond, the exclusive condition with condition(B)
     if implicit_value(condition(B), Cond) = true then
       return MERGE(A, true(B))
     else
       return MERGE(A, false(B))
   else
     begin
       ABl ← MERGE(A, true(B))
       ABr ← MERGE(A, false(B))
       if ABl = A⇒true(B) and ABr = A⇒false(B)
         return sequentialize(A,B)
       else
         return subtree(condition(B), ABl, ABr)
     end
else (condition(A) < condition(B))
   return MERGE(B, A)
```

OPTIM	CT	ET	LET	GN	LN	ALE	AGE	AUE
STB	0	26.15	-	-	-	-	-	-
LOC	0.01	6.67	6.67	-	276	56.09	-	-
FR	0.02	3.33	11.84	322(139/95/88)	286	76.05	15.04	5.49
PE	0.02	3.3	6.78	243(104/52/87)	295	52.01	11.84	0.36
AE	0.02	3.44	7.73	228(99/42/87)	293	59.62	12.96	1.9
FP	0.02	3.39	9.47	238(101/50/87)	287	72.19	12.6	2.79

Table 6.9: Compilation and evaluation results of simplified sequential trees

sequentialized) trees. This is probably due to the fact that our algorithm to traverse sequential trees is a bit more complicated and could be slightly improved. On the contrary, they are slightly better than for sequential (non simplified) trees thanks to the simplification. Thus combining simplification and sequentialization is the best choice on this benchmark and it is completely satisfactory. The order PE seems to be the best choice since it provides the best evaluation time –it provides an overall speed-up of 7.92 with respect to STB– and evaluates the least number of useless conditions. Moreover, the overall speed-up with respect to the original implementation of ASAX is now of about 16.9. Since compilation times are now completely satisfactory, we do not assess the impact of excluded conditions. In fact, the improvement is expected to be negligible. However, it is probably rewarding in situations involving a lot more distinct conditions and rules.

Recall that using local decision trees to calculate decisions is not advantageous for stateless rules (see Section 5.7). The evaluation factorization provided by the global tree now benefits to all kind of rules. This can be noticed in Figures 6.13 and 6.14; using a global tree –it has been built according to the PE order in these figures– provides the best analysis time regardless the number of rules or instances.

6.7 Conclusion

In this chapter, we have described an optimization technique to factorize the evaluation of conditions. This consists of merging all local decision trees to form a single global decision tree. It is necessary to consider the growing size of the global tree during the merging operation. The naive merging does not take that fact into account and, consequently, a complete merging cannot be achieved because the global tree grows exponentially. We had to deploy other optimization techniques based on sorted trees to keep the size of the global tree as small as possible during the merging. The first technique sequentializes independent subtrees, i.e., subtrees that do not share any common conditions, instead of merging them. It allows us to reduce the number of node duplications. The second technique simplifies all

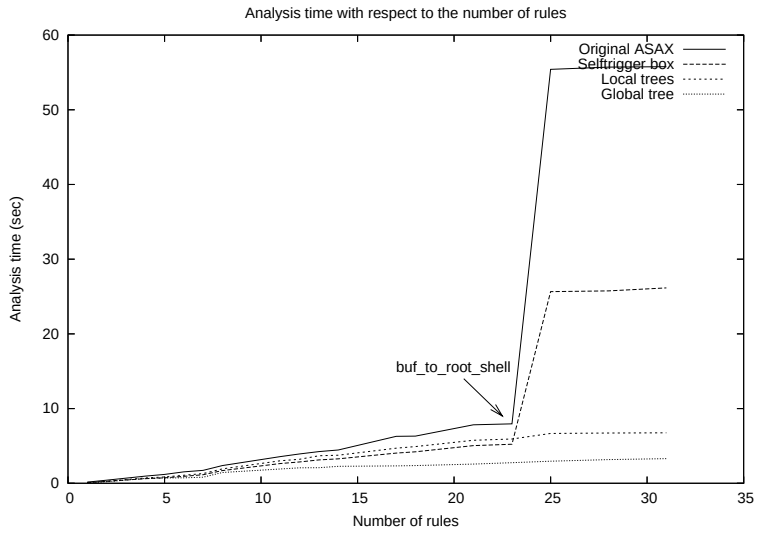


Figure 6.13: Analysis time with respect to the number of rules (with global tree)

nodes of the global tree whose evaluation is implicit regarding their ancestor conditions. Both techniques taken separately provide very good results; it is possible to compile the global tree in less than one second (for our benchmark). Moreover, taken together, they provide much better results; the compilation time drops to 0.02 seconds, which is quite satisfactory. Also, thanks to condition factorization, the evaluation time is more or less two times better than for evaluating local decision trees separately. It results in an overall speed-up of about 16.9 with respect to the original version of ASAX on the Darpa benchmark.

Nevertheless, there are some aspects that still need to be optimized. First, the problem about multiple evaluations of parametric conditions is still not solved. Moreover, global conditions that are moved inside parametric subtrees because of the condition ordering are also subject toe be evaluated several times. Second, the global decision tree includes all local decision trees, which is not really optimal. Indeed, calculating decisions for non active rules entails a waste of ressource. In fact, to be optimal, the global decision tree should include only the useful local decision trees. As a result, the global decision tree needs to be recalculated for each record regarding the active rules.

The next chapter addresses the first issue by designing an optimized evaluation technique for parametric conditions. In general, most records are relevant for only one rule instance and parametric conditions are crucial to select relevant records. In other words, parametric conditions ar true for

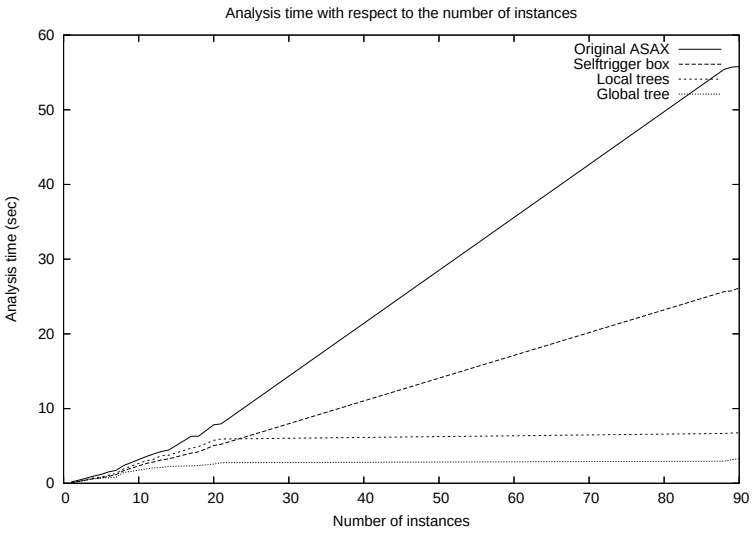


Figure 6.14: Analysis time with respect to the number of instances (with global tree)

only one instance (in general). The idea is thus to select the right instance in constant time when testing a parametric condition.

Chapter 7

Optimizing the evaluation of decision trees involving parametric conditions

7.1 Introduction

Till now, our optimizations are not well adapted for rules featuring conditions on parameters because the decisions are computed at a rule-level. Yet, instances of the same rule might have a different behavior depending on their actual parameters. We calculate a decision for every instance by evaluating parametric subtrees attached to *parameters* decisions. However, these subtrees are not optimized since their evaluation does not involve condition factorization. They are evaluated as many times as there are instances of the corresponding rule. Moreover, global conditions that receive an order number greater than at least one parametric condition may be located inside parametric subtrees and thus are likely to be evaluated several times.

In fact, our optimizations are very rewarding when most declared rules are stateless, i.e., do not have parameters. The massive self-triggers and the factorization reduce the evaluation time significantly. To the contrary, applications involving **many** instances of stateful rules cannot benefit from the optimizations. Indeed, in general, each event in the analyzed data stream belongs to at most one scenario and thus is relevant for only one rule instance. Moreover, stateful rules select relevant records by testing one or several NADF fields with respect to the actual parameters of each instance. For example, consider a RUSSEL module that monitors all TCP sessions in a network traffic. Typically, it creates a rule instance for every tracked session. Each instance receives the 4-uple identifier of its corresponding session as parameters. It consists of the source IP address, the destination IP address, the source port and the destination port. These four parameters are tested with respect to the current record to identify network packets

belonging to the right session. A network packet (thus a record) belongs to at most one TCP session. Yet, all instances are executed despite only one is actually useful.

In this chapter, we design a new algorithm to optimize the evaluation of parametric conditions. This novel optimization does not require to "isolate" these conditions in apart subtrees. The parameters decision is thus no longer used. The main idea consists of using hash tables to retrieve in minimum time all instances for which an evaluated parametric condition is true.

The Darpa benchmark does not benefit from the optimization because few rule instances are involved. Thus, to illustrate the efficiency of the technique, we perform experiments with an application involving thousands of rule instances. This application is the stateful honeytank [65].

The remainder of the chapter is organized as follows: Section 7.2 presents our new evaluation algorithm. Section 7.3 describes the low-level techniques used to implement the algorithm efficiency. Section 7.4 presents our experiments with the Darpa benchmark. Section 7.5 describes the experiments we have conducted with the stateful honeytank. Finally, Section 7.6 draws the conclusion.

7.2 Clustering rule instances while evaluating decision trees

Our optimization consists of a new evaluation algorithm for decision trees. We can describe the main changes by the following. The evaluation of a decision tree returns a triplet containing three sets of rule instances (E, T, D) where:

- the set E contains the instances to execute.
- the set T contains the instances to re-trigger.
- the set D contains the instances to discard.

During the evaluation of the global tree, each subtree is associated to a "current" set of rule instances of which it calculates the decisions. Every evaluation of a parametric condition splits the "current" set of the subtree whose root is the evaluated condition into two subsets: a) the instances for which the condition is true and b) the instances for which the condition is false. The first subset is associated to the "true" subtree while the second one is associated to the "false" subtree. The evaluation continues then through both "true" and "false" subtrees. When a global condition is evaluated, the "current" set is not splitted but the whole set is associated to the subtree determinated by the evaluation. The evaluation of the global tree then continues through that subtree.

It the evaluated subtree is composed of a tree sequence, the current instance set is split into smaller subsets. Each subset is associated to a subtree of the sequence and contains instances of rules whose decisions are calculated in this subtree. Note that instances cannot belong to more than one subset since all subtrees calculate decisions for distinct rules. Once the evaluation arrives at a leaf, the corresponding decision is assigned to all associated instances.

The new evaluation algorithm is depicted in Algorithm 7.1. We use the following representation for triplets:

- The empty triplet, i.e. the triplet containing three empty sets is represented by {}.
- The content of a triplet is indexed by decision names. For example, D[execute] denotes the set of instances to execute inside the triplet D.
- The operator $\cup$ performs the union of two triplets:
  Assuming that D1 = (E1, T1, D1) and D2 = (E2, T2, D2),
  D1 $\cup$ D2 = (E1 $\cup$ E2, T1 $\cup$ T2, D1 $\cup$ D2).

Assuming we know for each subtree the rules for which it calculates the decisions, the function $instances\_for\_subtree(I, ST)$ returns a subset of I containing instances of rules whose decisions are calculated in subtree ST. The function $evaluate\_condition(C, I)$ splits the instance set I into two subsets which are returned; one contains the instances for which the evaluated condition is true and the other contains the remaining instances.

Figure 7.1 depicts an example of a decision tree being evaluated by our new algorithm. Conditions A, B and C are parametric conditions. Assuming the tree is evaluated with a set of instances identified by distinct numbers ranging from 1 to 5, the table provided in the figure gives the values of the parametric conditions for each instance. The value "any" means that the condition is either true or false. For each subtree, we see the "current" instance set that is associated to it according to the values provided by the table. Once the tree has been evaluated, the returned triplet is ({1}, {2,3,4}, {5}).

Since the evaluation algorithm of decision trees does not return a set containing a decision for each rule, the algorithm evaluating a record needs to be adapted. The new version of this algorithm is provided in Algorithm 7.2. Instances to execute, to re-trigger and to discard are treated in turn in that order.

7.3 Implementation details

Despite the fact that the abstract algorithm is very simple, its implementation can be inefficient if we do not choose appropriate data structures

Algorithm 7.1 evaluate_decision_tree(T, I)

Precondition:

- T is a sequential decision tree.
- I is the "current" set of instances for the tree T

Postcondition:

- returns the triplet (E, ST, D) where:
  - E contains the instances of T to execute.
  - ST contains the instances of T to re-trigger.
  - D contains the instances of T to discard.

```
if external(T) then
   begin
     D ← {}
     D[decision(T)] ← I
     return D
   end
else if sequential(T) then
   begin
     res ← {}
     for all ST in subtrees(T) do
       res ← res ∪ evaluate_decision_tree(ST, instances_for_subtree(I, ST))
     return res
   end
else
   begin
     (TI, FI) ← evaluate_condition(condition(T), I)
     DT1 ← evaluate_decision_tree(true(T), TI)
     DT2 ← evaluate_decision_tree(false(T), FI)
     return DT1 ∪ DT2
   end
```

Algorithm 7.2 analyze_record(R) - Record analysis (modified version with local decision trees)

Precondition:

- R is the current NADF record.
- The environment contains a global decision tree GDT.

Postcondition:

- All instances in current have been treated (either executed, self-triggered, or discarded depending on the evaluation of the evaluation of the global decision tree GDT)

```
begin
  pre_process(R)
  (E, ST, D) ← evaluate_decision_tree(GDT, current)
  for all inst in E do
    begin
      APA ← param(inst)
      selftriggered ← evaluate_rule(code(inst))
      if selftriggered = false then
        kill(inst)
    end
  move_instances(ST, next)
  for all inst in D do
    kill(inst)
  while current_triggered is not empty do
    begin
      inst ← pick_one_instance(current_triggered)
      APA ← param(inst)
      evaluate_rule(code(inst))
      kill(inst)
    end
end
```

	A	B	C
1	true	true	any
2	true	false	any
3	true	false	any
4	false	any	true
5	false	any	false

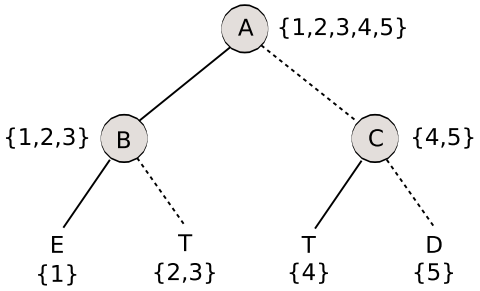


Figure 7.1: Associating instance sets to each subtree

wisely. In this section, we describe implementation designs for the two following problems:

1. Representing the "current" instance sets.

2. Splitting the "current" instance sets into two subsets while evaluating parametric conditions.

If we naively implement the abstract evaluation algorithm by using linked lists to represent the "current" instance sets (I in the algorithm), splitting these lists into sublists would be very time consuming. For example, splitting a list when evaluating a sequential tree amounts to apply the function $instances\_for\_subtree(I, ST)$ for each subtree ST of the sequence. This function would scan I for instances of rules associated to the subtree ST. Thus, the splitting operation has $O(n^2)$ complexity where n is the length of I.

Each parametric condition tests parameters of one single rule. If the instance selection process is implemented by evaluating the parametric condition for each instance of its rule, the actual parameters of the instance are copied to the parameters area (APA) of the virtual machine for each evaluation. If there are more than one parametric condition in an evaluated branch, actual parameters are most likely copied several times. The evaluation process suffers thus from a waste of resource. In many situations, the performance may be worse than the previous algorithms.

7.3.1 Representing the "current" instance sets

The "current" instance sets of subtrees are represented by the current lists inside rule descriptors. The implementation of our evaluation algorithm adopts the following invariant:

Invariant 1. For each evaluated subtree, the "current" instance set is represented by the current lists of all rules involved in the subtree.

When evaluating a sequential tree, we know that each subtree of the sequence calculate decisions for distinct rules. Thus, the subsets for each subtree are composed of the current lists of one or several rules already composing the "current" instance set. As a result, there is no need to "physically" split the instance set and Invariant 1 is maintained for the evaluation of each subtree.

Evaluating a parametric condition removes all instances for which the condition is true from the current list of its rule. They are backuped in a list inside the node to form the instance set for the "true" subtree evaluation. The remaining instances (in the current list) compose the instance set for the "false" subtree. The latter is first evaluated. Then, the backuped instances are restored inside the current list of the rule and the "true" subtree is evaluated.

When evaluating a decision, the following operations are performed depending on the decision:

- *execute*: All instances in the current list of the associated rule are (massively) moved to another list called the *to_execute list*. This operation is performed in constant time just like if it was a self-trigger operation. The to_execute list represents the instance set to execute E in the triplet returned by the abstract algorithm.

- *selftrigger*: All instances in the current list of the associated rule are self-triggered, i.e., they are (massively) moved to the next list of the same rule. This is the same self-trigger operation as before except the fact that it might be applied on a subset of the rule instances. Once the global tree is evaluated, all instances that are in the next lists of their respective rules compose the set of instances to re-trigger T in the triplet returned by the abstract algorithm. However, they are already re-triggered at that moment.

- *discard*: All instances in the current list of the associated rule are deallocated.

7.3.2 Splitting the "current" instance sets into two subsets while evaluating parametric conditions

Using hash tables to evaluate parametric conditions

To be optimal, instances for which an evaluated parametric condition is true should be retrieved in constant time. We propose to use hash tables to achieve this operation. If we assume that parametric conditions have the following form:

$$\mathrm{p} = \mathrm{v}$$

where p is a parameter and v a value, all instances having v as value for the parameter p form a set, which is identified by the value v. So, it is natural to define the entries inside hash tables with v as the key and with its corresponding instance set as the value. The idea is to assign a distinct hash table to each parametric condition. Whenever a parametric condition is evaluated, the tested value is used to access the hash table of the condition and thus to retrieve all instances for which the condition is true.

Our approach is not applicable to all parametric conditions. First, keys are single values. Interval of values cannot be used to access hash tables and, consequently, parametric conditions testing inequalities (operators $<$ and $>$) cannot be optimized. Note that disequalities (!=) are optimizable because the hash table actually returns the instances for which the condition is false in that case. Second, parametric conditions cannot involve more than one parameter to be optimized. Keys accessing hash tables are associated to only one parameter. It would be difficult to design generic keys for an arbitrary number of parameters. Finally, the tested parameters must not be part of a compound expression since key values in hash tables are actual parameter values. For example, assuming the condition `if p=10`, the corresponding hash table returns all instances whose parameter `p` equals to 10. Now, if we assume the condition `if p+1=10`, the hash table still returns the same instances though it should return those whose parameter `p` actually equals to 9.

Now that we have presented what the optimized evaluation of parametric conditions consists of, we are in position to give an algorithm for the function *evaluate_condition*(C, I), which is used in Algorithm 7.1. It is depicted in Algorithm 7.3. Actually, there are three different cases to treat: global conditions, optimizable parametric conditions and non optimizable parametric conditions. Just like the previous version of the algorithm, global conditions are evaluated once (by function *eval_global_condition*(C)). All instances associated to the evaluated subtree are then moved to the right subset accordingly. The other subset is returned empty. For each optimizable parametric condition, the function *get_true_instances*(C, v, I) returns a subset of I containing instances for which the condition C has value v (it accesses the hash table of condition C). Depending on if the condition is

an equality or a disequality (respectively tested with functions `eq(C)` and `neq(C)`), the retrieved instances have the tested condition equaling true and false respectively and are assigned to the right subset. Note that we always consider that hash tables returns "true" instances in the following of this thesis for clarity. The remaining instances in I are assigned to the other subset. Non optimizable parametric conditions are evaluated by function $eval\_for\_instance(C, i)$ for each instance i of the rule involving the condition. The instances to test are selected thanks to two successive operations. First, the function $in\_rule(C)$ returns the rule r involving the condition C. Then, instances of the associated set I that instantiate the rule r are extracted by function $get\_instances\_from\_rule(I, r)$. The tested instances are added to the right subset according to the evaluation results. The remaining instances in I do not have to be tested. So they are added to any subset indifferently (we have chosen the "false" subset in our algorithm).

Using magic numbers to select the right instances

Let us assume the following invariant concerning all hash tables:

Invariant 2. For each parametric condition optimized by a hash table, for each value v of the tested parameter, the associated hash table returns a list of all instances having the value v for the parameter.

This invariant implies that hash tables return instances that are not in the instance set associated to the current subtree. Indeed, it is possible that some returned instances are either in a backuped list in another node, in the list to_execute, or self-triggered (in the next list of the rule). Of course, instances that have been discarded cannot be returned since they have been deallocated and thus removed from hash tables. The problem is that only instances in the current list of the rule are eligible to form the instance subset for the "true" subtree. Assuming condition B is true for instance 4 in Figure 7.1, Figure 7.2 illustrates well the problem by depicting the situation where instance 4 must be self-triggered and executed for the same ASAX cycle. The reason is that when instance 4 is moved the next rule set of its rule, it remains accessible and able to be selected in all hash tables. Thus, instance 4 is selected for the second time when condition B is evaluated (recall that a "true" subtree is evaluated after its sibling "false" subtree).

One solution is to select instances by testing the presence of each returned instance in the current list of its rule. This is inefficient however because this operation is quadratic in the number of instances. We chose thus another solution involving *magic numbers* to achieve the selection of the correct instances. It consists of assigning an identifier number to each instance list. The identifier of the current list of a rule is stored inside its

Algorithm 7.3 evaluate_condition(C, I) - Condition evaluation

Precondition:

- C is a condition
- I is an instance set.

Postcondition:

- Let the current environment, the algorithm returns a pair (TI, FI) where:
  - TI is the instance set extracted from I for which the evaluation of C is true.
  - FI is the instance set extracted from I for which the evaluation of C is false or which do not evaluate C.

```
begin
  if parametric(C) then
    Assuming the involved parameter is compared to value v
    if eq(C) then
      begin
        TI ← get_true_instances(C, v, I)
        FI ← I \ TI
      end
    else if neq(C) then
      begin
        FI ← get_true_instances(C, v, I)
        TI ← I \ FI
      end
    else
      begin
        TI ← {}
        FI ← {}
        r ← in_rule(C)
        Inst ← get_instances_from_rule(I, r)
        for all i in Inst do
          if eval_for_instance(C, i) = true then
            TI ← TI ∪ {i}
          else
            FI ← FI ∪ {i}
        FI ← FI ∪ (I \ Inst)
      end
  else
    if eval_global_condition(C) = true then
      begin
        TI ← I
        FI ← {}
      end
    else
      begin
        TI ← {}
        FI ← I
      end
  return (TI, FI)
end
```

	A	B	C
1	true	true	any
2	true	false	any
3	true	false	any
4	false	*true*	true
5	false	any	false

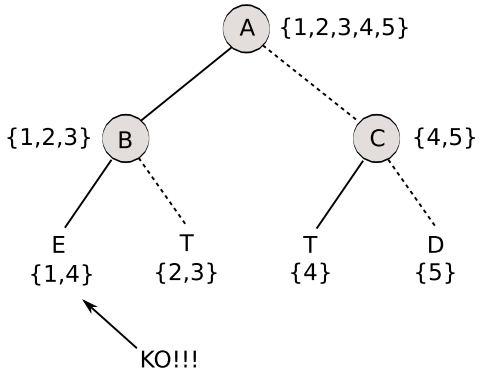


Figure 7.2: Example of a wrong instance selection

descriptor and every instance holds the identifier (magic number) of the list containing it inside its own descriptor as well. Thus, to select an instance returned by a hash table, we have to check if its magic number is equal to the magic number of the current list of its rule. This operation is achieved in constant time. Checking all instances returned by a hash table has therefore O(number of instances for which the tested condition is true) complexity. However, it is expected that this operation is achieved in constant time in most cases. Indeed, in most situations, parameters are tested to select relevant records. Since every record is generally relevant for at most one instance, hash tables return a list composed of one single instance in these situations.

Magic numbers are assigned to instances when they are created or when they are selected by a parametric condition evaluation (when they move to another list). They are assigned in an ascendant order and the next number to use is stored in the global variable N. When evaluating a parametric condition, all selected instances are backuped and receive the magic number N. N is then incremented by one. The magic numbers of the remaining instances are unchanged, as well as the number of the current list of the rule. Once the "false" subtree is evaluated, the backuped instances are restored in the current list, their magic number is assigned to the latter and the "true" subtree is evaluated. Note that the "true" subtree is not evaluated if no instances have been selected. However, we cannot know

if there are remaining instances for the "false" subtree. Indeed, it is too costly to scan all rule descriptors for at least one non empty current list at each evaluation of a parametric condition . Thus, the "false" subtree is systematically evaluated. Assuming all instances have the magic number 5 at the beginning of the cycle in Figure 7.1, Figure 7.3 how magic numbers are assigned during the evaluation of the tree. We can notice that instance 4 can no more be selected by condition B since the instance has the magic number 7 and the current list of its rule has number 6 at that moment.

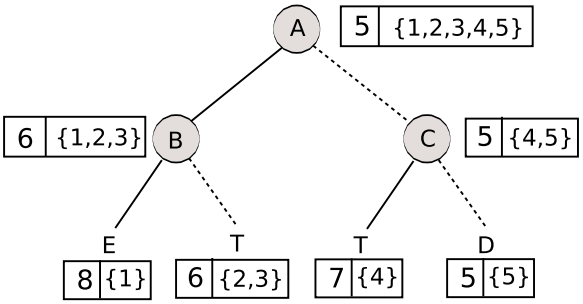


Figure 7.3: Example of magic number distribution

Once the global tree is evaluated, all instances that self-trigger are in the next lists of their respective rules. They all have a magic number lower than the final value of N and the magic numbers might differ from one instance to another (since subsets of instances might have been self-triggered in distinct leaves and thus with different numbers). Also, instances triggered by the executed ones receive a magic number lower than N. Consequently, at the beginning of the next cycle, all current instances, as well as all current lists of rules, have a magic number lower than the value of N. We call these lists *initial lists*. As magic numbers of initial lists are not synchronized to the magic numbers of their instances, the selection process cannot be performed when accessing a hash table. Therefore, initial lists have to be treated differently. Considering N_start_cycle, the value of N at the beginning of the ASAX cycle. An initial list is identified when its magic number is lower than N_start_cycle. Then, if the initial list is actually the current list, only instances having also a number lower than N_start_cycle are selected.

To ensure the correct evaluation of the global tree, the two following invariants must be satisfied:

Invariant 3. Let a rule r, all its instances having a magic number lower than N_start_cycle or having the same magic number are:

- either all in the current list of the rule r.
- either all in a backuped list inside a node.

- either all in the to_execute list.
- or all in the next list of the rule r.

Invariant 4. During the evaluation of the global tree, the initial list of instances of each rule has been splitted like follows:

- Instances to be executed have been placed in the to_execute list. The magic numbers of these instances are lower than N.
- Instances to re-trigger have been placed in the next list of their rule. The magic numbers of these instances are lower than N.
- A residue of the initial list where the numbers of the list and of all instances are lower than N_start_cycle.
- The rest has been placed inside (current or backuped) lists with a magic number between N_start_cycle and $N-1$. All instances have the same number as their host list.

Binding hash tables to rule instances

When a new instance is created, it has to be added to the hash tables of every optimizable parametric condition the local tree of the instantiated rule contains. When an instance is discarded or dies after its execution, it must be removed from all hash tables containing it. We design now a data structure that allows us to achieve these operations in constant time.

Assuming the local tree of a single rule contains n parametric conditions optimized by a hash table, each instance of the rule figures in $n+1$ lists. n lists correspond to one condition (hash table). The $n+1^{th}$ list is either the current list of the rule, the next list of the rule, a selected backuped list or the list to_execute. We call these lists the *main lists*. Cells composing the main lists are instance descriptors. They are double-linked in order to remove instances in constant time.

The cells of the n lists in hash tables point to the instances in the main lists. They are also double-linked to allow the removal of references to instances in constant time (when instances are discarded). For each instance, there are n cells in the hash tables that point to this instance. These cells are regrouped in a *bind list*. Each instance descriptor contains a pointer to the bind list corresponding to the instance.

Figure 7.4 illustrates the invariant of the general situation for one instance. We only represent the current lists of rules among the main lists for clarity.

Let us illustrate now that adding and removing an instance of a rule to and from our structure is achieved in constant time. In fact, these operations

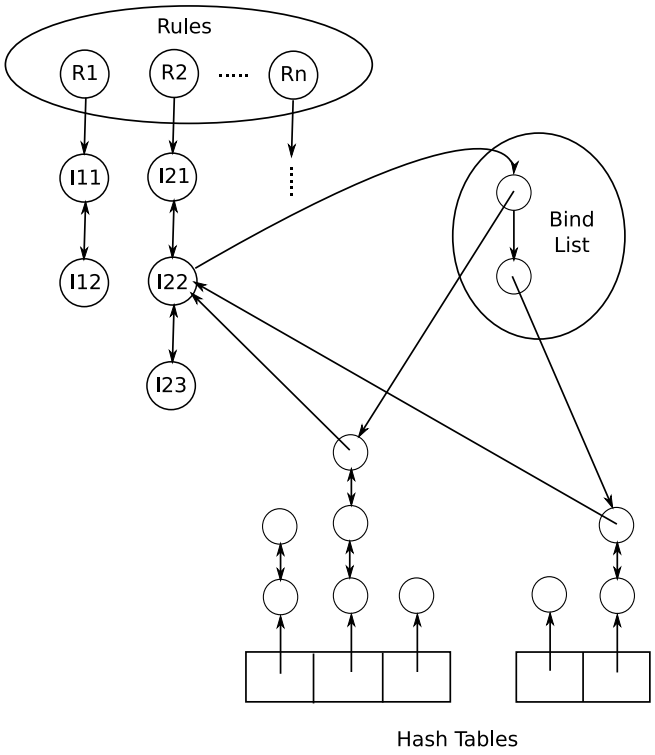


Figure 7.4: Example of a bind list

are linear in the number of parametric conditions for the rule, which is actually constant since this number is fixed during the whole analysis.

Adding an instance to hash tables Each time a rule instance is triggered, cells referring to it have to be added to all hash tables corresponding to the parametric conditions present in the rule. To achieve it, we extend rule descriptors to contain a list of conditions for each parameter of the rule. When an instance is created by a trigger action, for each parameter, a cell referring to the instance is added to the hash tables of the conditions present in the corresponding list. The bind list is built in the same time; for each inserted cell in the hash tables, a corresponding cell is added to the bind list. Finally, a reference to the bind list is put inside the descriptor of the created instance.

Removing an instance from hash tables Each time a rule instance is discarded, its references have to be removed from all hash tables. The cells to remove are retrieved thanks to the bind list of the instance. Each cell of the bind list is deallocated in parallel with its corresponding cell in the hash tables.

7.4 Experimentation with the Darpa Benchmark

We have tested our new optimization technique on the Darpa benchmark. For this experiment, we use the PE order (see 6.3.4) to sort the decision trees. Table 7.1 compares the results with our previous implementation presented in Chapter 6. We remind that the column CT provides the compilation time, including parsing RUSSEL rules, extracting local trees, sorting local trees, and merging sorted local trees. The column ET is the total evaluation time for the benchmark, excluding reading the BSM file and translating records into NADF format. The column GN provides the number of nodes of the global tree. Between parentheses are the numbers of conditions, of sequential subtrees, and of decision nodes, respectively. Common subtrees are cached and they are counted only once. The column AGE is the average number of evaluated conditions when the global tree is used. The column AUE gives the average number of conditions that are uselessly evaluated in the global tree, i.e., they are not evaluated when local (not sorted) trees are used to take the decisions. The line NO PARAM provides the results for our previous implementation, which does not involve parametric conditions optimization while the line WITH PARAM gives the results for the implementation involving our novel optimization.

	CT	ET	GN	AGE	AUE
NO PARAM	0.02	3.3	243(104/52/87)	11.84	0.36
WITH PARAM	0.02	3.76	243(105/70/68)	12.48	0.72

Table 7.1: Results with optimized parametric conditions

We observe that the sizes of both global trees are identical. However, the repartition between conditions, sequential subtrees and decision nodes differs significantly. Indeed, since there are no more parametric subtrees in the WITH PARAM version, more subtrees are merged. This illustrated by a drop of the number of decision nodes, along with a significant increase of sequential subtrees. The average number of evaluations increases slightly as well as the average number of useless evaluations. It is a consequence of merging the parametric subtrees. It entails an evaluation time a bit slower with respect to the NO PARAM version.

Moreover, in our Darpa benchmark, parametric conditions are not systematically evaluated for all records. Some global conditions are more critical and are used before parametric conditions to filter relevant records. In addition, there are very few instances for stateful rules, hence the impact of using hash tables is not rewarding enough to overcome the loss entailed by the additional useless evaluations.

As a result, we assess our approach with the help of another benchmark in next section. This benchmark features many rule instances and thus

should highlight the improvement provided by the optimization of parametric conditions.

7.5 Experimentation with the stateful honeytank

The work of Nicolas Vanderavero on honeytanks [65] uses the optimization techniques presented in this chapter. His work includes experiments emphasizing the efficiency of the optimization of parametric conditions.

In this section, we present a sample of the experiments presented by Vanderavero in his PhD thesis. The experiments composing the sample feature the stateful version of the honeytank. The stateful honeytank is an application implemented in RUSSEL and involving a high number of rule instances in parallel. Part of the instances simulate a virtual host with a unique IP address. Other instances simulate TCP sessions open on these virtual hosts. Results show that the optimized version of ASAX is able to simulate an arbitrarily large number of hosts. The reason is that, for each incoming network packet, ASAX executes only the rule instances identified by two IP addresses and two port numbers (actually very few among many). All other rule instances are re-triggered in constant time. The original version is not able to simulate a large number of hosts because all rule instances are executed for every incoming packet.

We first give a short description of honeytanks as well as their objectives in Section 7.5.1. To benefit from the optimizations, rules have to be designed carefully. We explain in Section 7.5.2 how this is achieved. Finally, we present and discuss the results of the experiments in Section 7.5.3.

7.5.1 Definition of honeytanks

In the field of networking, honeypots [58] are systems specifically designed to be probed, attacked, or compromised by an online attacker. They are used to collect malicious traffic for further analyses or to simply act as simple alarms. High-interaction honeypots are actual systems with operating systems and applications. Low-interaction honeypots simulate a subset of the functionalities of a real system. Their degree of realism depends on the simulated functionalities. Low-interaction honeypots usually simulate several hosts but their number cannot be large because of efficiency issues.

Honeytanks [66, 67] are low-interaction honeypots that are designed to simulate large amount of hosts. Both a stateless and a stateful version are implemented in RUSSEL. For our experiments, we only focus on the stateful version since it creates large sets of rule instances (with parameters). In terms of functionalities, the objectives of the stateful honeytank are the following[1]:

[1]We assume the reader is familiar with the TCP protocol

- **Tracking TCP sessions.** The first objective of the stateful honeytank is to accurately track TCP sessions in order to identify to which session a given segment belongs.

- **Handling TCP session initiation and clearing phases.** With its ability to track TCP session, the stateful honeytank knows in which state is each TCP session. As a result, it correctly handles the 3-way handshake and TCP clearing phases in every situation.

- **Detection of out of order TCP segments.** The sequence number present in TCP segments is used to detect lost, duplicated or desequenced segments. The stateful honeytank detects retransmitted and desequenced segments in the data exchange phase of a session.

- **Retransmission of SYN+ACK segments.** As TCP is a reliable protocol, segments which are not acknowledged in time must be retransmitted. This part of the TCP specification is not implemented. However, to prove that a retransmission mechanism is feasible, the stateful honeytank retransmits SYN+ACK segments which are not acknowledged in time during the 3-way handshake phase.

- **Supporting fingerprinting scans.** The TCP protocol have many different implementations (called TCP/IP stacks) which can be found in different operating systems. It is possible to identify a particular TCP/IP stack by observing its behavior in specific cases. This technique is called *Operating System fingerprinting*. The last objective of the stateful honeytank is to simulate various TCP/IP stacks to respond correctly to fingerprinting tools.

We invite the reader to refer to [65] for a more detailed discussion of honeytanks.

7.5.2 Implementing honeytanks to benefit from optimizations

The Stateful honeytank is well adapted for our optimizations because every incoming packet belong to at most one session. Consequently, only one rule instance among a large number should be executed.

To fully benefit from the optimizations, the honeytank rules are written accordingly to a methodology developed by Nicolas Vanderavero in his PhD thesis. We now present a description of two situations needing that methodology. For a full description of the methodology, we invite the reader to refer to the chapter 7 of Vanderavero's thesis.

Selecting relevant network packets

Rules tracking TCP sessions usually have as parameters the 4-uple identifier of the tracked TCP session. To determine if they are interested in the current record (network packet), they check if it has the same 4-uple as the one passed as parameters. Thus a conjunction of four parametric conditions is evaluated. It is expected that only one instance satisfies all conditions. However, one single condition may satisfy many instances. For example, if the monitored network traffic contains many parallel TCP sessions opened to the same server, the "TCP destination port" parameter has the same value for many rule instances. As a result, the accessed hash table returns a long list of "true" instances to treat. The right instance is thus not selected in constant time.

The solution to this problem is to consider the identifying parameters as a whole. Instead of the identifying parameters, rules receive as parameter their concatenation. To determine if the current event must be handled by the rule, it tests if the concatenation of the identifying attributes of the current record matches the unique identifier passed as parameter. Unfortunately, during the implementation of the stateful honeytank, this solution was not usable due to an incomplete implementation of our optimizations. Indeed, if the concatenation of the identifying attributes of an event is larger than four bytes, it must be stored in a string variable. Using string as key to access hash tables was not available at that time. A temporary solution has thus been developed by using a hash of the identifying parameters and a stub rule. By hashing a set of identifying parameters, we increase the distribution of parameter values across the instances thus limiting the risk of having many instances with the same value for a single parameter. In addition to the identifying parameters, the stub rule receives as parameter a hash of those parameters. This hash is stored in a four bytes integer location. To determine if the current packet must be handled by the rule, the stub rule tests if the hash of the identifier of the current packet equals to the hash passed as parameter. If it is not the case, the (stub) rule re-triggers itself. Otherwise, it triggers for the current record the complete version of the rule. The complete rule compares as usual the identifier of the packets with its parameters to ensure that there was no collision (due to the hashing). If the complete rule self-triggers or triggers another rule that also proposes a stub, it must trigger the stub version of the rule instead. Thus, the complete version of the rule is only triggered for the current record ever. As a result, the rule is not optimized and the efficiency issue does not appear anymore while evaluating the global tree.

Optimizing timeouts

As explained earlier, an inequality test on a parameter amounts the evaluation of the condition for all instances of the rule involving the condition. It is obviously inefficient and using such conditions must avoided as most as possible. However, honeytanks use such conditions to implement a timeout mechanism.

To avoid the evaluation of an inequality test at each record, the author of the honeytank manages to calculate an estimation of the number of the record for which a timeout will be expired. This number is given as parameter to the created instances which involve timeouts. Testing a timeout consists then of testing an equality between the number of the current record and the number passed as parameter. Fortunately, the values of this parameter are likely to be randomly spread amongst rule instances. It is thus fully optimizable. If the current record number is not the estimated record number, the instance simply self-triggers. Otherwise, it means that it should be in timeout. It thus performs the real timeout test (the inequality) to ensure that the instance is really expired. Indeed, this has to be achieved since the number of record mechanism is only an estimation. If the last test does not satisfy, it means that the instance is not expired. The rule thus computes a new estimation and re-triggers itself with the new value. For complete details about the estimation mechanism, we invite the reader to refer to the chapter 7 of Vanderavero's PhD thesis [65].

7.5.3 Assessing the honeytank

The experiment we present now aims to emphasize the benefit brought by our optimizations to the honeytank. We show that the stateful version of the honeytank simulates an arbitrary number of hosts and TCP sessions with an ASAX cycle time (noted T_{ASAX}) remaining constant (bounded by a constant). It means that optimizing the evaluation of parametric conditions allows ASAX to handle an arbitrary number of instances of the same rule in constant time. When using a non-optimized version of ASAX, T_{ASAX} is linear in the number of instances.

Presentation of the experiments

We assess two versions of the stateful honeytank. One is designed for the non-optimized version of ASAX while the other is crafted for the optimized version. We denote the two versions as HT_{opt} and as $HT_{no\text{-}opt}$, respectively. We also denote the optimized and non optimized versions of ASAX respectively as $ASAX_{opt}$ and $ASAX_{no\text{-}opt}$. Both versions of the stateful honeytank are executed by both versions of ASAX. There are thus four configurations in total.

The four configurations are assessed by the following experiment. A *network traffic generator*, implemented in RUSSEL and executed by $ASAX_{opt}$, opens a given number of TCP sessions at a given rhythm towards distinct hosts simulated by the honeytank. The honeytank is configured to stop opened TCP session after 10 seconds of inactivity. The number of opened TCP sessions is 10,000. For each configuration we perform a *slow* test and a *fast* test. The slow test consists of opening a session every 10,000 microseconds while the fast test opens a session every 1,000 microseconds. In total, we have thus eight tests.

For each test, we measure the evolution of T_{ASAX} over time and the response time of the honeytank, i.e., the elapsed time between the emission of a SYN segment by the traffic generator and the reception of the SYN+ACK reply.

Experimental results

Non-optimized stateful honeytank executed by the non-optimized ASAX

We assess the behaviour of $HT_{no\text{-}opt}$ executed by $ASAX_{no\text{-}opt}$. Figure 7.5 depicts the evolution of T_{ASAX} and the response time of the honeytank during the slow test. We can see that, as a rule instance is created for each opened session, T_{ASAX} grows linearly during the first 10 seconds. At that moment, the first established session has reached its timeout value of 10 seconds. Thus, the honeytank starts to discard sessions (and thus instances) while the traffic generator continues to open sessions. T_{ASAX} is then stable between 2,400 and 2,700 microseconds because sessions are opened and discarded at the same rhythm. When the generator has opened 10,000 sessions, it remains silent and the honeytank discards the remaining sessions progressively during the last 10 seconds. Since the response time mostly depends on T_{ASAX}, it also grows linearly in the number of active rule instances.

Figure 7.6 depicts the evolution of T_{ASAX} and the response time of the honeytank over time during the fast test. To avoid scale problems and keep the figure readable, values higher than the 98th percentile are filtered out. We observe that T_{ASAX} grows faster and reaches over 25,000 microseconds, which is almost 10 times slower than for the slow test. This is because more sessions have been established before the honeytank starts discarding them. Thus, as the generator emits a SYN every 1,000 microseconds, the honeytank cannot cope with the packet arrival rhythm since it takes more than 1,000 microseconds to treat a packet. Packets waiting to be treated are accumulated in a queue in the format adapter. As a result, the response time of a SYN is the time used to treat it plus the time used to treat all packets in the queue before this SYN packet. The response time thus reaches

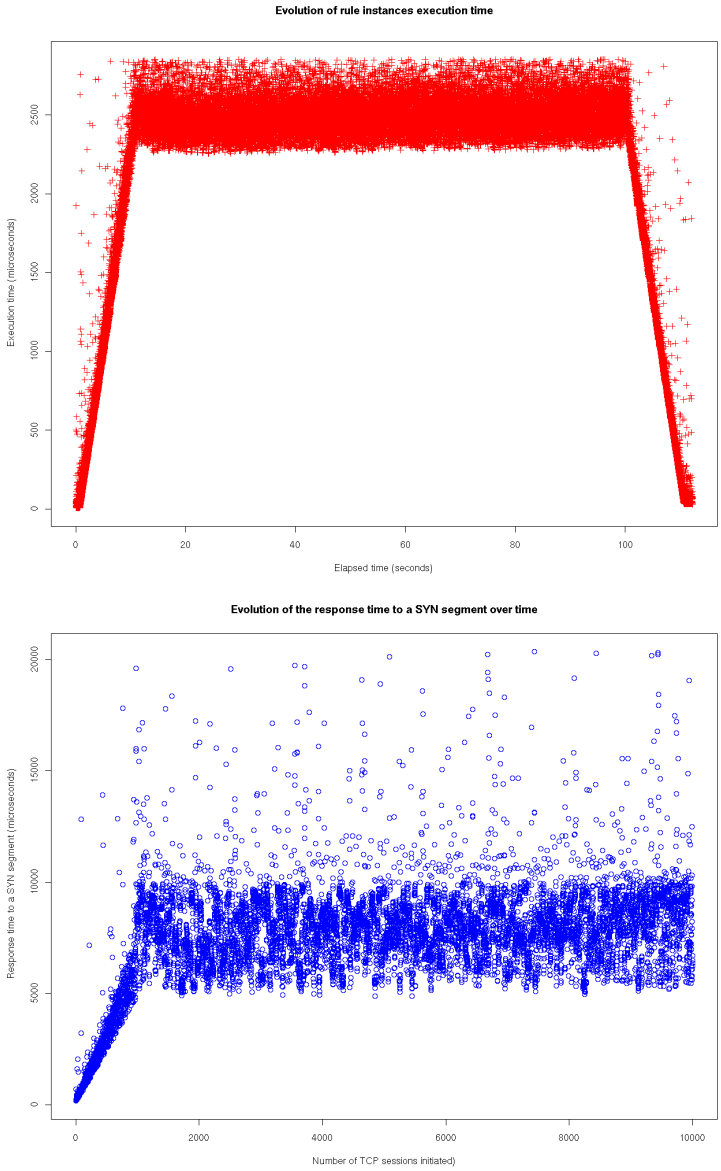


Figure 7.5: $HT_{no\text{-}opt}$ - $ASAX_{no\text{-}opt}$ - slow test: Evolution of T_{ASAX} and response time

150 seconds.

Optimized stateful honeytank executed by the optimized ASAX

We assess now the behaviour of HT_{opt} executed by $ASAX_{opt}$. Figure 7.7 depicts the evolution of T_{ASAX} and the response time of the honeytank over time during the slow test. Unlike the previous test, T_{ASAX} never goes over 500 microseconds and thus the response time remains stable around 375 microseconds. We see that, with the optimized version of ASAX, T_{ASAX} is bounded and is lower than for the previous test. We can also notice that the time to treat all sessions (102 seconds) is faster than for the previous test (115 seconds).

Figure 7.8 depicts the evolution of T_{ASAX} over time during the fast test. We do not provide the evolution of the response time because it is similar to the slow test. Like in the previous test, we can see that T_{ASAX} globally remains below 500 microseconds. The peaks are explained by the fact that calls to the `libpcap` library[2] stalls the execution of ASAX on some occasions. However, it is worth noticing that all sessions are treated in 12 seconds instead of 300 seconds for the previous test. This good performance is due to the fact that T_{ASAX} remains below 1,000 microseconds so that ASAX can cope with the packet arrival rhythm.

Non-optimized stateful honeytank executed by the optimized ASAX

We assess now the behaviour of $HT_{no\text{-}opt}$ executed by $ASAX_{opt}$. We do not show graphs for this test as the results are similar to the configuration where $ASAX_{no\text{-}opt}$ is used. Both T_{ASAX} and the response time grow linearly in the number of rule instances. It means that, in order to benefit from the optimizations, RUSSEL rules need to be specifically written for the optimized version of ASAX. Moreover, the performance of $ASAX_{opt}$ executing $HT_{no\text{-}opt}$ is not worse than the performance of $ASAX_{no\text{-}opt}$. The optimized version of ASAX can consequently be safely used in all cases.

Optimized stateful honeytank executed by the non-optimized ASAX

Finally, we assess the behaviour of HT_{opt} executed by $ASAX_{no\text{-}opt}$. The results are similar to the configuration where $HT_{no\text{-}opt}$ is used. Both T_{ASAX} and the response time grow linearly in the number of rule instances. However, the absolute results are twice better in this test. It might be due to the fact that writing the rules specifically for the optimized version of the honeytank has forced the programmer to write more structured code.

[2]The library used by the network format adaptor to capture the traffic

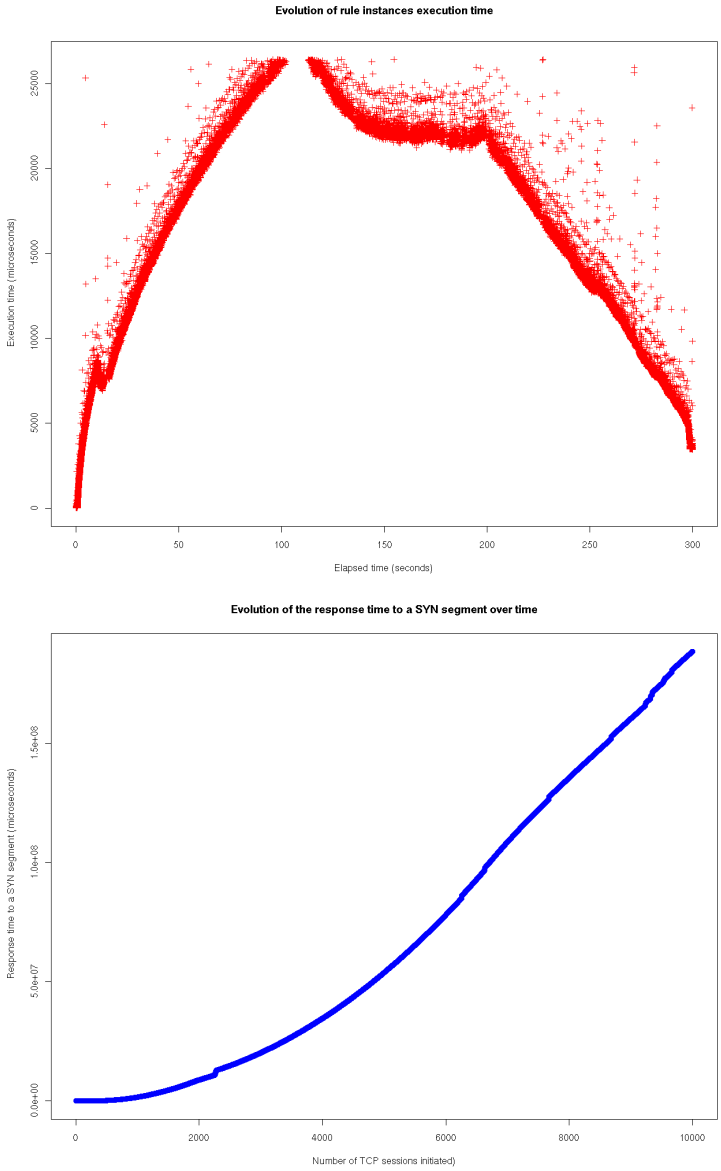


Figure 7.6: $HT_{no\text{-}opt}$ - $ASAX_{no\text{-}opt}$ - fast test: Evolution of T_{ASAX} and response time

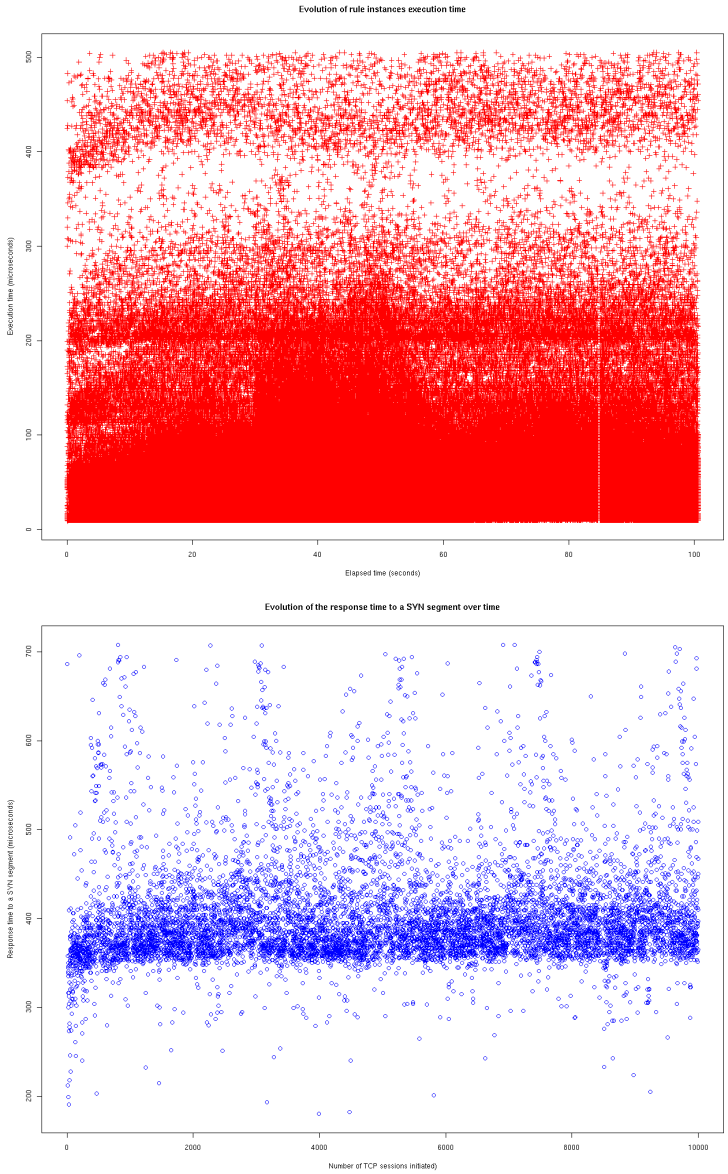


Figure 7.7: HT_{opt} - $ASAX_{opt}$ - slow test: Evolution of T_{ASAX} and response time

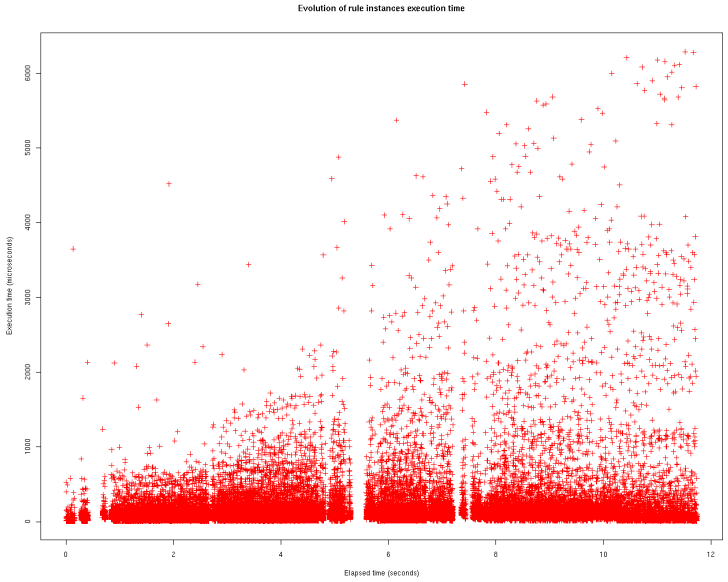


Figure 7.8: HT_{opt} - $ASAX_{opt}$ - fast test: Evolution of T_{ASAX}

7.6 Conclusion

In this chapter, we have designed an optimization technique to evaluate parametric conditions efficiently. This technique uses hash tables to select rule instances for which an evaluated parametric condition is true, in constant time. However, this technique cannot be applied to all kinds of parametric conditions and it is not optimal when a group of parametric conditions is used to select relevant events. Consequently, rules must be specifically (re)written to benefit from the optimization technique.

Assessing our technique with the Darpa benchmark does not highlight the benefits of our optimizations because few rule instances are active. Thus, we have assessed our optimization with the stateful honeytank, which involves thousands of active rule instances. We have shown that, in this case, our technique is very rewarding. The stateful honeytank is able to simulate thousands of hosts while being capable to cope with a high rate of incoming traffic. More precisely, the execution time necessary to treat each network packet is constant regardless the number of simulated hosts.

Part IV

Conclusion

Chapter 8

Conclusion and future work

In this PhD thesis, we have addressed the problem of optimizing a programming system that applies a large set of "rules" to each element of a data (or "event") stream. The rules are executed in non determinate order (or even in parallel) and they are created and killed dynamically. We have proposed specific data structures and algorithms to solve the problem. We have implemented our method inside ASAX and we have conducted experiments showing the usefulness of our techniques. In this conclusion, we raise a few questions relative to the generality of our approach: is it easy to integrate our techniques into other programming systems and what are the limits and possible improvements to what we propose ?

8.1 Applicability to other systems

The applicability of the optimization methods presented in our work much depends on the software architecture of the system under consideration. In the ASAX case, the applicability is straightforward because of the simple and modular architecture of ASAX. Also, if a system already enjoys specific optimizations, they may conflict with ours. This suggests that some rewriting, actually simplification, of an original system implementation should be necessary before applying our techniques.

We further discuss these issues for STAT and Bro, the two systems presented in Chapter 3.

8.1.1 Optimizing STAT

Remind that the STAT approach consists in testing and firing transitions to follow attack scenarios progressing through signatures formalized by state transition diagrams (see Section 3.4). The STAT core applies the mechanisms to match scenario signatures against an event flow.

The STAT core uses the concept of *event subscription* to test relevant

transitions only. It consists in associating outgoing transitions from the current state of all scenario instances with a specific event type, called *event spec*. When an event is treated by the STAT core, the event spec of the event is matched against a database and the associated transitions are retrieved (see Section3.4 again). This concept is similar to our optimization approach; it avoids the testing of the event spec for all transitions to be potentially fired individually.

The concept of event subscription does not fully optimize the execution of the scenarios however. Assertions of transitions and assertions of state are evaluated individually for each subscription, resulting in a potential redundancy. Our work should benefit to STAT-based IDSs by solving this problem. This situation is similar to retrieving relevant rule instances to execute in ASAX. A transition has to be fired (executed) if its event spec equals to the event spec in the current event, and if its assertion is verified, and if the assertion of the destination state is verified. Thus, for all defined transitions, a local decision tree can be built from the expression:

> Event spec = event spec in the event AND assertion of the transition AND assertion of the destination state.

where leaves are decisions determining if the corresponding transition fires. Then, local decision trees are merged involving all our optimization techniques. The resulting global decision tree, when evaluated, returns the transitions to fire.

8.1.2 Optimizing Bro

The relevance of integrating our optimizations into Bro must be discussed for its two layers: the event engine and the policy interpreter.

Optimizing the event engine

Concerning the event engine, depending on the analyzed packet headers, this layer selects a subset of analyzers to apply on the packet. These analyzers perform one task among the following:

1. The analyzer performs nothing if the packet does not bring any relevant information.

2. The analyzer performs a side effect action such as modifying a connection state and/or generating an event.

The question is now to determine whether the execution of this subset may be optimized. Analyzers are not executed more than one time for a single packet. Thus, we cannot optimize the redundancy of many "instances" of the same analyzer as we did for rules in ASAX.

We could optimize the redundancy of condition testing over all analyzers of the subset however. We have two solutions: either (a) we build a global decision tree for all analyzers implemented in Bro or (b) we dynamically build a global decision for each monitored connection. We prefer the second solution for the following reason: few analyzers are executed on a single packet. Thus, a global decision tree for all analyzers would be too big and would entail useless condition evaluations of unapplied analyzers. On the contrary, dynamically building a global decision tree for each novel connection would result in a small tree since fewer analyzers are involved. Moreover, we could use caching to build trees for connections of the same type.

Optimizing the policy interpreter

The policy interpreter cannot be optimized by our optimization approach. In ASAX, our optimizations are efficient because all rules test the same data, i.e., an NADF record. In Bro, there is no concept of global data for all event handlers. Each event has its own environment defined by its arguments. As a result, there is no common condition to optimize. This is an example of an existing optimization conflicting with ours. Note that the lack of global data for event handlers means that some data is duplicated. Thus another approach similar to us could be more appropriate.

Implementing the Bro model in ASAX

The analysis process of Bro is implemented optimally by instantiating its model for the network domain and by integrating specific techniques to manage network packets efficiently. Thus, optimizing the existing architecture of Bro with our work does not provide a significant gain. To use our optimizations, it is possible to implement the Bro model in RUSSEL. In that situation, optimizing the ASAX execution should exhibit a performance similar to Bro. Moreover, the user would enjoy a more modular architecture and would have the possibility to mix Bro-based analyses with other kinds of analyses.

To implement the model of Bro in RUSSEL, we can adapt the work of Nicolas Vanderavero [65] to follow the traffic of distinct connections. The event engine can be implemented by a set of rules, each rule standing for a specific protocol analyzer. Instances of “analyzer” rules are triggered for each monitored connection. Implementing the signature engine can be done in a straightforward way [14]. Each event handler is implemented by a distinct rule, which is triggered by an “analyzer” rule.

In Bro, information about states is shared by several protocol analyzers monitoring the same connection and by event handlers. In ASAX, this information would be duplicated and hold by parameters of the instances

needing the information. We would use global frozen variables to make instances communicate between them in order to notify an information update. Then, each instance updates its own version when necessary.

8.2 Limitations and possible improvements to our work

The techniques proposed in this thesis are applicable and/or optimal only under some conditions.

1. The language to optimize must be modular and must have a clear notion of "rule" (or thread, or process...), which must be easily amenable to static analysis. The static analysis techniques used in this thesis are simple and straightforward. More powerful techniques exist but static analysis is not the subject of this thesis.

2. The evaluation of the global tree may entail useless work when not all rules are active, i.e., when some rules do not have at least one active instance.

3. The evaluation of parametric conditions can be inefficient when the condition applies to parameter values shared by a large number of instances. Ideally, only parametric conditions on identifying values should be included in the global tree.

Some of the above limitations have been experienced by Nicolas Vanderavero in his work on honeytanks [65]. They have been overcome, in his context, by introducing a few programming patterns preventing inefficiencies. Future work would address the definition of improved programming constructs to guide the compiler for choosing the best optimizations. More powerful static analysis techniques could also be used to the same aim. Finally, it should be investigated how to make the global tree evolve dynamically when not all rules are active.

Bibliography

[1] Bro Web Site. http://www.bro-ids.org.

[2] TCPdump/libpcap Web Site. http://www.tcpdump.org.

[3] B. Delcourt A. van Lamsweerde, A. Dardenne and F. Dubisy. The kaos project: Knowledge acquisition in automated specification of software. In *Proceedings AAAI Spring Symposium Series, Stanford University, American Association for Artificial Intelligence*, pages 59–62, March 1991.

[4] R. De Landtsheer A. van Lamsweerde, S. Brohez and D. Janssens. From system goals to intruder anti-goals: Attack generation and resolution for security requirements engineering. In *Proceedings of the RE03 Workshop on Requirements for High Assurance Systems (RHAS03), Monterey (CA)*, pages 49–56, September 2003.

[5] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 40–45, New York, NY, USA, 1990. ACM.

[6] Simon Brohez and Yann Grégoire. ORKA: Obstacles Recognition with KAOS based on ASAX = Utilisation d'ASAX pour la Détection d'Obstacles. Master's thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, 2002.

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[8] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[9] Françoise Gérard. Définition et Implémentation d'un Langage Déclaratif pour l'Analyse d'Audit Trails. Master's thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgium, 1998.

[10] D. Curry and H. Debar. Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition, June 2002. Internet Draft.

[11] MIT Lincoln Laboratory - DARPA Intrusion Detection Evaluation. URL. http://www.ll.mit.edu/IST/ideval/index.html.

[12] Hervé Debar and Benjamin Morin. Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, *Recent Advances in Intrusion Detection (RAID 2002)*, LNCS 2516, pages 177–198, Zurich, Switzerland, October 2002. Springer-Verlag.

[13] S. Debray, S. Kannan, and M. Paithane. Weighted decision trees. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 654–668, Washington, USA, 1992. The MIT Press.

[14] Jérôme Devigne. Transposition d'une configuration snort en règles asax: principes de traduction et comparaison expérimentale d'efficacité. Master's thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2007.

[15] Edsger W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[16] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[17] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational experiences with high-volume network intrusion detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 2–11, New York, NY, USA, 2004. ACM.

[18] R. O. Duda, P. E. Hart, and N. J. Nilsson. Subjective bayesian methods for rule-based inference systems. In *Readings in uncertain reasoning*, pages 274–281, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[19] Steven T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer. Statl: an attack language for state-based intrusion detection. *J. Comput. Secur.*, 10(1-2):71–103, 2002.

[20] Marc Dacier Fabien Pouget and Hervé Debar. Honeypots, a practical mean to validate malicious fault assumptions. In *PRDC'04, 10th International symposium Pacific Rim dependable computing Conference, Tahiti, French Polynesia*, March 2004.

[21] Charles L. Forgy. *OPS5 User's Manual*. Departement of Computer Science, Crnegie-Mellon University, 1981.

[22] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial intelligence*, volume 19, pages 17–37. 1982.

[23] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 348–356, New York, NY, USA, 1987. ACM.

[24] Naji Habra, Baudouin Le Charlier, and Abdelaziz Mounji. Advanced Security Audit Trail Analysis on Unix (ASAX also called SAT-X). Implementation Design of the NADF Evaluator. Technical report, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgium, September 1994.

[25] Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. ASAX: Software Architecture and Rule-Based Language for Universal Audit Trail Analysis. In *European Symposium on Research in Computer Security (ESORICS'92)*, Toulouse, France, November 1992. Springer-Verlag.

[26] Peter E. Hart and Richard O. Duda. Prospector–a computer based consultation system for mineral exploration. Technical Report 155, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1977. Presented at the Taita Hills Conference on Standards for Computer Applications in Resource Studies, Kenya, Nov 8-15, 1977. To be published in International Association for Mathematical Geology, v 10, no 5-6.

[27] K. Ilgun, A. Kemmerer, and P.A. Porras. State Transition Analysis: A Rule-based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[28] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[29] S. Kliger and E. Shapiro. A decision tree compilation algorithm for fcp(—, :, ?). In *Fifth Int. Conf. on Logic Programming*, pages 1315–1336, Seattle, August 1988. MIT Press.

[30] Shmuel Kliger and Ehud Shapiro. From decision trees to decision graphs. In *Proceedings of the 1990 North American conference on Logic programming*, pages 97–116, Cambridge, MA, USA, 1990. MIT Press.

[31] M. Korsloot and E. Tick. Compilation techniques for nondeterminate flat concurrent logic programming languages. In K. Furukawa, editor, *Logic Programming: Proc. of the Eighth International Conference*, pages 457–471. MIT Press, Cambridge, MA, 1991.

[32] Christopher Kruegel and Thomas Toth. Automatic Rule Clustering for improved, signature based Intrusion Detection. Technical report, University of California, Santa Barbara, USA, 2002.

[33] Christopher Kruegel and Thomas Toth. Using Decision Trees to Improve Signature-Based Intrusion Detection. In Giovanni Vigna, Erland Jonsson, and Christopher Kruegel, editors, *Recent Advances in Intrusion Detection (RAID 2003)*, volume 2820 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2003.

[34] S. Kumar and E.H Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th national Computer Security Conference*, pages 11–21, 1994.

[35] Josu Kuri, Gonzalo Navarro, Ludovic Mé, and Laurent Heye. A Pattern Matching Based Filter for Audit Reduction and Fast Detection of Potential Intrusions. In Hervé Debar, Ludovic Mé, and S. Felix Wu, editors, *Recent Advances in Intrusion Detection (RAID 2000)*, LNCS 1907, pages 17–27, Toulouse, France, October 2000. Springer-Verlag.

[36] Ulf Lindqvist and Phillip A. Porras. Detecting computer and network misuse through the production-based expert system toolset (p-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, 1999.

[37] Richard Lippman, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA Off-Line Intrusion Detection Evaluation. Technical report, Lincoln Laboratory MIT, 2000.

[38] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, volume 2, pages 12–26. Lincoln Laboratory MIT, 2000.

[39] Xavier Martin and Baudouin Le Charlier. Experimental evaluation of some optimizations for rule-based intrusion detection. Technical report,

Université Catholique de Louvain - Faculté des Sciences Appliquées - Département d'Ingénierie Informatique, April 2007.

[40] Xavier Martin and Baudouin Le Charlier. Optimizing a rule-based intrusion detection language to handle a large number of signatures. In *Proceedings of the 5th Conference on Security and Network Architectures (SAR 2006)*, June 2006.

[41] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[42] John McHugh. The 1998 Lincoln Laboratory IDS Evaluation: A Critique. In Herv Debar, Ludovic M, and S. Felix Wu, editors, *Recent Advances in Intrusion Detection (RAID 2000)*, LNCS 1907, pages 145–161, Toulouse, France, October 2000. Springer-Verlag.

[43] Sebastian Schmerl Michael Meier and Hartmut Koenig. Improving the efficiency of misuse detection. In Springer Berlin / Heidelberg, editor, *Intrusion and Malware Detection and Vulnerability Assessment*, volume 3548/2005, pages 188–205, June 2005.

[44] C. Michel and L. Mé. ADeLe: An Attack Description Language for Knowledge-Based Intrusion Detection. In *Proc. of the 16th International Conference on Information Security*, 2001.

[45] Bernard M. E. Moret. Decision trees and diagrams. *ACM Comput. Surv.*, 14(4):593–623, 1982.

[46] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* PhD thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgium, September 1997.

[47] Abdelaziz Mounji, Baudouin Le Charlier, and Isabelle Mathieu. *ASAX: Advanced Sequential File Analysis product (manual)*, November 1993.

[48] Peter G. Neumann and Phillip A. Porras. Experience with EMERALD to Date. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, 1999.

[49] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[50] Jean-Philippe Pouzol and Mireille Ducassé. From declarative Signatures to Misuse IDS. In Wenke Lee, Ludovic M, and Andreas Wespi, editors, *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS 2212, Davis, CA, USA, October 2001. IRISA/INSA de Rennes, Springer-verlag.

[51] Jean-Philippe Pouzol and Mireille Ducassé. Formal Specification of intrusion Signatures and Detection Rules. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW) 2002*. IRISA/INSA de Rennes, 2002.

[52] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.

[53] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of USENIX LISA'99*, 1999.

[54] Muriel Roger and Jean Goubault-Larrecq. Log Auditing through Model-Checking. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236, Cape Breton, Nova Scotia, Canada, 2001. IEEE Comp. Soc. Press.

[55] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[56] Snort IDS website. http://www.snort.org.

[57] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[58] Lance Spitzner. *Honeypots*. Addison-Wesley, September 2002.

[59] Anurag Srivastava, Eui-Hong Han, Vipin Kumar, and Vineet Singh. Parallel formulations of decision-tree classification algorithms. *Data Mining and Knowledge Discovery*, 3(3):237–261, 1999.

[60] Stat website. http://www.cs.ucsb.edu/~seclab/projects/stat/index.html.

[61] SunSoft. Solaris SHIELD Basic Security Module, October 1993.

[62] Eric Totel, Bernard Vivinis, and Ludovic Mé. A language driven ids for event and alert correlation. In *Proceedings ot the 19th IFIP International Information Security Conference*, pages 209–224, 2004.

[63] Paul E. Utgoff. Incremental induction of decision trees. *Mach. Learn.*, 4(2):161–186, 1989.

[64] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *RAID*, volume 4637 of *Lecture Notes in Computer Science*, pages 107–126. Springer, 2007.

[65] Nicolas Vanderavero. *A Lightweight Framework to Build Honeytanks.* PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 2007.

[66] Nicolas Vanderavero, Xavier Brouckaert, Olivier Bonaventure, and Baudouin Le Charlier. The HoneyTank : a scalable approach to collect malicious Internet traffic. International Infrastructure Survivability Workshop (IISW'04), December 2004.

[67] Nicolas Vanderavero and Baudouin Le Charlier. Towards a more stateful and accurate HoneyTank. In *Proceedings of the 5th Conference on Security and Network Architectures (SAR 2006)*, June 2006.

[68] G. Vigna, S. Eckmann, and R. Kemmerer. The stat tool suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, USA, January 2000. IEEE Computer Society Press.

[69] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *ESEC / SIGSOFT FSE*, pages 88–97, 2003.

[71] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

Appendix

Appendix A

RUSSEL rules of the Darpa benchmark

This appendix lists all RUSSEL modules composing the Darpa benchmark used for our experiments. We first give a short description of used external user-made functions in a separated section.

A.1 Description of the external functions

Type-related functions

- `strToInt(buf: string)`:
  returns the conversion of the content of `buf` into a signed integer. (Recall that all files in NADF records are encoded by strings of bytes. Integer values thus need to be converted by this function to be manipulated).

- `stringlen(buf: string)`:
  returns the length of the string of bytes `buf`.

Unix-related functions

- `is_suid(file_mode: string)`:
  returns 1 if the unix file mode `file_mode` contains the set-uid bit, 0 otherwise.

- `is_gid(file_mode: string)`:
  returns 1 if the unix file mode `file_mode` contains the set-gid bit, 0 otherwise.

- `is_IROTH(file_mode: string)`:
  returns 1 if the unix file mode `file_mode` contains the "readable for others" bit, 0 otherwise.

- `is_IWOTH(file_mode: string)`:
  returns 1 if the unix file mode `file_mode` contains the "writable for others" bit, 0 otherwise.
- `is_IRGRP(file_mode: string)`:
  returns 1 if the unix file mode `file_mode` contains the "readable for the group", 0 otherwise.
- `is_IWGRP(file_mode: string)`:
  returns 1 if the unix file mode `file_mode` contains the "writable for group", 0 otherwise.
- `contains_non_ascii(buf: string)`:
  returns 1 if the string of bytes `buf` contains non-ascii characters, 0 otherwise.
- `shell_script(path: string)`:
  returns 1 if the file denoted by the path `path` is a shell script, 0 otherwise.
- `set_process_info(pid: integer, gid: integer, auid: integer, path: string)`:
  stores information of a process in a data base inside the external module. The information is composed of the tuple <PID, GID, audit user ID, path of the executed binary> (respectively parameters `pid`, `gid`, `auid` and `path`). Storing a process whose PID is already in the database overwrites the associated information with the parameters passed to the function.
- `delete_process_info(pid: integer)`:
  removes information about the process indentified by `pid` from the databse inside the external module.

Policy-related functions

USTAT uses an external file whose content defines files and directories of which accesses are restricted. USTAT scenarios uses these definitions to perform policy verification. The following functions are a translation of the routines used by USTAT.

- `init_policy`:
  initiates a database containing the policy information defined by the external file. It returns 1 if the initialization is successful, 0 otherwise.
- `policy_member(file: string, list:string)`:
  returns 1 if the file `file` is present in the list (defined by the external file) identified by `list`, 0 otherwise. For example, the condition `policy_member(path_0_path, 'RESTRICTED_READ_OBJECTS') =`

`1` checks if the path defined in the current event is present in the list identified by the string `'RESTRICTED_READ_OBJECTS'` in the external list.

- `policy_member_dir(file: string, dir: string)`:
  returns 1 if the file `file` is present in at least one directory in a list (defined by the external file) identified by `dir`, 0 otherwise.
- `policy_member_dir_sub(file: string, dir: string)`:
  returns 1 if the file `file` is present in at least one subdirectory of any directory defined in a list (defined by the external file) identified by `dir`, 0 otherwise.
- `policy_in_user_set(auid: integer, set: string)`:
  returns 1 if the user `auid` exists in a set (defined by the external file) identified by the string `set`, 0 otherwise.

A.2 Rules

Listing A.1: unix_buf_overflow.asa

```
1  global internal max_len : integer;
2
3  init_action;
4  begin
5    max_len := 128;
6    trigger off for_next exec1
7  end;
8
9  rule exec1;
10 begin
11   if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
         23) and /* AUE_EXECVE or AUE_EXEC */
12     strToInt(return32_proc_value) != -1 and
13     (is_suid(attr_file_mode) = 1 or not present attr_token_ID) and
14     strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
15     stringlen(exec_args_env_args) > max_len and
16     contains_non_ascii(exec_args_env_args) = 1
17     --> trigger off for_current alert
18   fi;
19   trigger off for_next exec1
20 end;
21
22 rule alert;
23 begin
24   println('user: ', strToInt(subject32_audit_ID),
25     ' - potential buffer overflow in ', path_0_path);
26   println(stringlen(exec_args_env_args))
27 end.
```

Listing A.2: unix_buf_overflow.asa

```
1  init_action;
2  trigger off for_next exec1;
3
4  rule exec1;
5  begin
6    if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
         23) and /* AUE_EXECVE or AUE_EXEC */
7      strToInt(return32_proc_value) != -1 and
8      is_suid(attr_file_mode) = 1 and
9      strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
10     match('^..*/usr/bin/su$', path_0_path) = 0
11     --> trigger off for_next exec2(strToInt(subject32_PID),
12                                    strToInt(subject32_audit_ID),
13                                    path_0_path)
14   fi;
15   trigger off for_next exec1
16 end;
17
18 rule exec2(pid:integer; auid:integer; binary:string);
19 begin
20   if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
         23) and /* AUE_EXECVE or AUE_EXEC */
21     strToInt(return32_proc_value) != -1 and
22     strToInt(subject32_PID) = pid and
23     strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
24     (match('^..*/usr/bin/sh$', path_0_path) = 1 or
25       match('^..*/usr/bin/ksh$', path_0_path) = 1)
26     --> begin
27       trigger off for_current alert(auid, binary);
28       trigger off for_next exec2(pid, auid, binary)
29     end;
30
31     strToInt(header32_event_ID) = 1 and /* AUE_EXIT */
32     strToInt(subject32_PID) = pid
33     --> skip;
34
35     true --> trigger off for_next exec2(pid, auid, binary)
36   fi
37 end;
38
39 rule alert(auid:integer; binary:string);
40 println('Buffer To Root Shell: User ', auid, ' Path: ',binary).
```

Listing A.3: unix_enable_suid_sgid.asa

```
1  init_action;
2  trigger off for_next mod_perm;
3
4  rule mod_perm;
5  begin
6    if strToInt(header32_event_ID) = 39 and /* AUE_FCHMOD */
7      strToInt(return32_proc_value) != -1 and
8      present attr_token_ID and is_suid(attr_file_mode) = 1 and
```

```
9       is_gid(attr_file_mode) = 1
10      --> trigger off for_current alert
11    fi;
12    trigger off for_next mod_perm
13  end;
14
15  rule alert;
16  println(strToInt(subject32_audit_ID),': setuid/setgid enabled on file ',
17          path_0_path).
```

Listing A.4: unix_execute_exit.asa

```
1   init_action;
2   begin
3     trigger off for_next exec;
4     trigger off for_next exit
5   end;
6
7   rule exec;
8   begin
9     if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
          23) and /* AUE_EXECVE or AUE_EXEC */
10      strToInt(return32_proc_value) != -1
11      --> set_process_info(strToInt(subject32_PID),
12                           strToInt(subject32_group_ID),
13                           strToInt(subject32_audit_ID),
14                           path_0_path)
15    fi;
16    trigger off for_next exec
17  end;
18
19  rule exit;
20  begin
21    if strToInt(header32_event_ID) = 1 /* AUE_EXIT */
22      --> delete_process_info(strToInt(subject32_PID))
23    fi;
24    trigger off for_next exit
25  end.
```

Listing A.5: unix_ftp_write.asa

```
1   init_action;
2   trigger off for_next create_file;
3
4   rule create_file;
5   begin
6     if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
          79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
7       strToInt(return32_proc_value) != -1 and
8       strToInt(subject32_user_ID) != 0 and
9       present attr_token_ID
10      and strToInt(attr_owner_UID) != strToInt(subject32_audit_ID)
11      --> trigger off for_next login(strToInt(attr_filein_ID),path_0_path)
12    fi;
```

```
13    trigger off for_next create_file
14  end;
15
16  rule login(inode:integer; file:string);
17  begin
18    if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
          23) and /* AUE_EXECVE or AUE_EXEC */
19      strToInt(return32_proc_value) != -1 and
20      match('^../login$', path_0_path) = 1
21      --> trigger off for_next read_rhosts(strToInt(subject32_PID),
22                                           inode, file);
23
24      true --> trigger off for_next login(inode, file)
25    fi
26  end;
27
28  rule read_rhosts(pid:integer; inode:integer; file:string);
29  begin
30    if
31      (strToInt(header32_event_ID) = 72 or /* AUE_OPEN_R */
32       strToInt(header32_event_ID) = 73 or /* AUE_OPEN_RC */
33       strToInt(header32_event_ID) = 74 or /* AUE_OPEN_RT */
34       strToInt(header32_event_ID) = 75) and /* AUE_OPEN_RTC */
35      strToInt(return32_proc_value) != -1 and
36      strToInt(subject32_PID) = pid and
37      present attr_token_ID and strToInt(attr_filein_ID) = inode
38      --> trigger off for_current alert(file);
39
40      strToInt(header32_event_ID) = 1 and /* AUE_EXIT */
41      strToInt(subject32_PID) = pid
42      --> trigger off for_next login(inode, file);
43
44      true --> trigger off for_next read_rhosts(pid, inode, file)
45    fi
46  end;
47
48  rule alert(file:string);
49  println('ftpwrite attack: ', file).
```

Listing A.6: unix_lpd_delete.asa

```
1   init_action;
2   if init_policy = 1
3     --> trigger off for_next exec;
4     true --> println('rule unix_lpd_delete skipped')
5   fi;
6
7   rule exec;
8   begin
9     if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
          23) and /* AUE_EXECVE or AUE_EXEC */
10      strToInt(return32_proc_value) != -1 and
11      match('^../usr/ucb/lpr$', path_0_path) = 1
12      --> trigger off for_next read(strToInt(subject32_audit_ID),
```

```
13                                      strToInt(subject32_PID),
14                                      path_0_path)
15    fi;
16    trigger off for_next exec
17  end;
18
19  rule read(auid:integer; pid:integer; path:string);
20  begin
21    if
22      (strToInt(header32_event_ID) = 72 or /* AUE_OPEN_R */
23       strToInt(header32_event_ID) = 73 or /* AUE_OPEN_RC */
24       strToInt(header32_event_ID) = 74 or /* AUE_OPEN_RT */
25       strToInt(header32_event_ID) = 75) and /* AUE_OPEN_RTC */
26      strToInt(return32_proc_value) != -1 and
27      strToInt(attr_owner_UID) != strToInt(subject32_audit_ID) and
28      strToInt(subject32_PID) = pid and
29      strToInt(subject32_user_ID) = 0 and
30      policy_member_dir(path_0_path, 'LPD_SPOOL_DIRS') = 1
31      --> trigger off for_next delete(auid, pid, path)
32    fi;
33    trigger off for_next read(auid, pid, path)
34  end;
35
36  rule delete(auid:integer; pid:integer; path:string);
37  begin
38    if strToInt(header32_event_ID) = 6 and /* AUE_UNLINK */
39      strToInt(return32_proc_value) != -1 and
40      strToInt(attr_owner_UID) != strToInt(subject32_audit_ID) and
41      strToInt(subject32_PID) = pid and
42      strToInt(subject32_user_ID) = 0
43      --> trigger off for_current alert(auid, path)
44    fi;
45    trigger off for_next delete(auid, pid, path)
46  end;
47
48  rule alert(auid:integer; path:string);
49  println(auid,': ',path,' deleted illegally using lpr -r').
```

Listing A.7: unix_mail_spool_hack.asa

```
1   init_action;
2   trigger off for_next mod_perm;
3
4   rule mod_perm;
5   begin
6     if strToInt(header32_event_ID) = 39 and /* AUE_FCHMOD */
7       policy_member_dir(path_0_path, 'MAIL_SPOOL_DIRS') = 1 and
8       present attr_token_ID and
9       is_IROTH(attr_file_mode) = 1 and
10      is_IWOTH(attr_file_mode) = 1 and
11      is_IRGRP(attr_file_mode) = 1 and
12      is_IWGRP(attr_file_mode) = 1
13      --> trigger off for_current alert
14    fi;
```

```
15    trigger off for_next mod_perm
16  end;
17
18  rule alert;
19  println(strToInt(subject32_audit_ID), ': mail file ', path_0_path,
20          ' had its permissions changed').
```

Listing A.8: unix_non_owner_write.asa

```
1   init_action;
2   trigger off for_next write;
3
4   rule write;
5   begin
6     if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
          79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
7       strToInt(return32_proc_value) != -1 and
8       present attr_token_ID and
9       strToInt(subject32_user_ID) != strToInt(attr_owner_UID) and
10      strToInt(subject32_user_ID) != 0
11      --> trigger off for_current alert
12    fi;
13    trigger off for_next write
14  end;
15
16  rule alert;
17  println(strToInt(subject32_user_ID), ': ', path_0_path,
18          ' written by user ', strToInt(subject32_user_ID),
19          ' owner ', strToInt(attr_token_ID)).
```

Listing A.9: unix_restricted_dir_write.asa

```
1   init_action;
2   if init_policy = 1
3     --> trigger off for_next create;
4     true --> println('rule unix_restricted_dir_write skipped')
5   fi;
6
7   rule create;
8   begin
9     if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
          79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
10      strToInt(return32_proc_value) != -1 and
11      policy_member_dir(path_0_path, 'RESTRICTED_WRITE_DIRECTORIES') = 1
12      --> trigger off for_current alert
13    fi;
14    trigger off for_next create
15  end;
16
17  rule alert;
18  println(strToInt(subject32_audit_ID), ': ', path_0_path,
19          ' created by user ', strToInt(subject32_audit_ID),
20          ', process ', strToInt(subject32_PID)).
```

Listing A.10: unix_restricted_read.asa

```
1  init_action;
2  if init_policy = 1
3    --> trigger off for_next read;
4    true --> println('rule unix_restricted_read skipped')
5  fi;
6
7  rule read;
8  begin
9    if
10     (strToInt(header32_event_ID) = 72 or /* AUE_OPEN_R */
11      strToInt(header32_event_ID) = 73 or /* AUE_OPEN_RC */
12      strToInt(header32_event_ID) = 74 or /* AUE_OPEN_RT */
13      strToInt(header32_event_ID) = 75) and /* AUE_OPEN_RTC */
14     strToInt(return32_proc_value) != -1 and
15     policy_member(path_0_path, 'RESTRICTED_READ_OBJECTS') = 1
16     --> trigger off for_next alert
17   fi;
18   trigger off for_next read
19 end;
20
21 rule alert;
22 println(strToInt(subject32_audit_ID), ': ', path_0_path,
23                  ' read by user ', strToInt(subject32_audit_ID)).
```

Listing A.11: unix_restricted_write.asa

```
1  init_action;
2  if init_policy = 1
3    --> trigger off for_next write;
4    true --> println('rule unix_restricted_write skipped')
5  fi;
6
7  rule write;
8  begin
9    if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
       79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
10     strToInt(return32_proc_value) != -1 and
11     policy_member(path_0_path, 'RESTRICTED_WRITE_CONFIG_FILES') = 1
12     --> trigger off for_next alert
13   fi;
14   trigger off for_next write
15 end;
16
17 rule alert;
18 println(strToInt(subject32_audit_ID), ': ', path_0_path,
19         ' written by user ', strToInt(subject32_audit_ID)).
```

Listing A.12: unix_secret_remote.asa

```
1  init_action;
2  var ip_local:string;
3  begin
```

```
4    if init_policy = 1
5    --> begin
6      ip_local := IP_AsciiToBinary('130.104.229.21');
7      trigger off for_next access1(ip_local);
8      trigger off for_next access2(ip_local);
9      trigger off for_next access3(ip_local)
10   end;
11   true --> println('rule unix_secret_remote skipped')
12   fi
13 end;
14
15 rule access1(ip:string);
16 begin
17   if
18     (strToInt(header32_event_ID) = 72 or /* AUE_OPEN_R */
19      strToInt(header32_event_ID) = 73 or /* AUE_OPEN_RC */
20      strToInt(header32_event_ID) = 74 or /* AUE_OPEN_RT */
21      strToInt(header32_event_ID) = 75) and /* AUE_OPEN_RTC */
22     strToInt(return32_proc_value) != -1 and
23     subject32_machine_ID = ip and
24     policy_in_user_set(strToInt(subject32_audit_ID), 'DARPA_SECRET') = 1
25     and policy_member_dir_sub(path_0_path, 'DARPA_SECRET') = 1
26     --> trigger off for_current alert
27   fi;
28   trigger off for_next access1(ip)
29 end;
30
31 rule access2(ip:string);
32 begin
33   if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
       79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
34     strToInt(return32_proc_value) != -1 and
35     subject32_machine_ID = ip and
36     policy_in_user_set(strToInt(subject32_audit_ID), 'DARPA_SECRET') = 1
37     and policy_member_dir_sub(path_0_path, 'DARPA_SECRET') = 1
38     --> trigger off for_current alert
39   fi;
40   trigger off for_next access2(ip)
41 end;
42
43 rule access3(ip:string);
44 begin
45   if strToInt(header32_event_ID) = 6 and /* AUE_UNLINK */
46     strToInt(return32_proc_value) != -1 and
47     subject32_machine_ID = ip and
48     policy_in_user_set(strToInt(subject32_audit_ID), 'DARPA_SECRET') = 1
49     and policy_member_dir_sub(path_0_path, 'DARPA_SECRET') = 1
50     --> trigger off for_current alert
51   fi;
52   trigger off for_next access3(ip)
53 end;
54
55 rule alert;
56 println(strToInt(subject32_audit_ID), ': unencrypted access of ',
```

```
57          path_0_path, ' by user', strToInt(subject32_audit_ID), ' from ',
58          IP_BinaryToAscii(subject32_machine_ID)).
```

Listing A.13: unix_secret_unauth.asa

```
1  init_action;
2  if init_policy = 1
3    --> begin
4      trigger off for_next access1;
5      trigger off for_next access2;
6      trigger off for_next access3
7    end;
8    true --> println('rule unix_secret_remote skipped')
9  fi;
10
11 rule access1;
12 begin
13   if
14     (strToInt(header32_event_ID) = 72 or /* AUE_OPEN_R */
15      strToInt(header32_event_ID) = 73 or /* AUE_OPEN_RC */
16      strToInt(header32_event_ID) = 74 or /* AUE_OPEN_RT */
17      strToInt(header32_event_ID) = 75) and /* AUE_OPEN_RTC */
18     strToInt(return32_proc_value) != -1 and
19     policy_in_user_set(strToInt(subject32_audit_ID), 'DARPA_SECRET') = 0
20     and policy_member_dir_sub(path_0_path, 'DARPA_SECRET') = 1
21     --> trigger off for_current alert
22   fi;
23   trigger off for_next access1
24 end;
25
26 rule access2;
27 begin
28   if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
       79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
29     strToInt(return32_proc_value) != -1 and
30     policy_in_user_set(strToInt(subject32_audit_ID), 'DARPA_SECRET') = 0
31     and policy_member_dir_sub(path_0_path, 'DARPA_SECRET') = 1
32     --> trigger off for_current alert
33   fi;
34   trigger off for_next access2
35 end;
36
37 rule access3;
38 begin
39   if strToInt(header32_event_ID) = 6 and /* AUE_UNLINK */
40     strToInt(return32_proc_value) != -1 and
41     policy_in_user_set(strToInt(subject32_audit_ID), 'DARPA_SECRET') = 0
42     and policy_member_dir_sub(path_0_path, 'DARPA_SECRET') = 1
43     --> trigger off for_current alert
44   fi;
45   trigger off for_next access3
46 end;
47
48 rule alert;
```

```
49 println(strToInt(subject32_audit_ID), ': access of ', path_0_path,
50         ' by unauthorized user', strToInt(subject32_audit_ID)).
```

Listing A.14: unix_sh_minus_hack.asa

```
1  init_action;
2  trigger off for_next hardlink;
3
4  rule hardlink;
5  begin
6    if strToInt(header32_event_ID) = 5 and /* AUE_LINK */
7      strToInt(return32_proc_value) != -1 and
8      present attr_token_ID and
9      is_suid(attr_file_mode) = 1 and
10     is_IXGRP(attr_file_mode) = 1 and
11     is_IXOTH(attr_file_mode) = 1 and
12     match('^..*/-*$', path_0_path) = 1 and
13     shell_script(path_0_path) = 1
14     --> begin
15       trigger off for_next exec(strToInt(subject32_audit_ID),
             path_1_path);
16       trigger off for_current alert1(strToInt(subject32_audit_ID),
17                                      path_0_path, path_1_path)
18     end
19   fi;
20   trigger off for_next hardlink
21 end;
22
23 rule exec(auid:integer; target_path:string);
24 begin
25   if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
         23) and /* AUE_EXECVE or AUE_EXEC */
26     strToInt(return32_proc_value) != -1 and
27     path_0_path = target_path and
28     strToInt(subject32_user_ID) = auid
29     --> trigger off for_next alert(auid, target_path)
30   fi;
31   trigger off for_next exec(auid, target_path)
32 end;
33
34 rule alert1(auid:integer; source_path:string; target_path:string);
35 println(auid, ': suspicious link created from setuid file ',
36         path_0_path, ' to ', path_1_path);
37
38 rule alert(auid:integer; target_path:string);
39 println(auid, ': suid sheel invoked interactively via ', target_path).
```

Listing A.15: unix_suid_shell.asa

```
1  init_action;
2  trigger off for_next exec1;
3
4  rule exec1;
5  begin
```

```
6     if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
          23) and /* AUE_EXECVE or AUE_EXEC */
7       strToInt(return32_proc_value) != -1 and
8       strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
9       (present attr_token_ID and is_suid(attr_file_mode) = 1) and
10      shell_script(path_0_path) = 1
11      --> trigger off for_next read(strToInt(subject32_audit_ID),
12                                    strToInt(subject32_user_ID),
13                                    strToInt(subject32_PID),
14                                    path_0_path)
15    fi;
16    trigger off for_next exec1
17  end;
18
19  rule read(auid_dest:integer; euid:integer; pid_exec:integer; source_path
        :string);
20  begin
21    if
22      (strToInt(header32_event_ID) = 72 or /* AUE_OPEN_R */
23       strToInt(header32_event_ID) = 73 or /* AUE_OPEN_RC */
24       strToInt(header32_event_ID) = 74 or /* AUE_OPEN_RT */
25       strToInt(header32_event_ID) = 75) and /* AUE_OPEN_RTC */
26      strToInt(return32_proc_value) != -1 and
27      strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
28      strToInt(subject32_user_ID) != euid
29      --> trigger off for_next exec2(auid_dest, pid_exec,
30                                     strToInt(subject32_PID), source_path)
31    fi;
32    trigger off for_next read(auid_dest, euid, pid_exec, source_path)
33  end;
34
35  rule exec2(auid_dest:integer; pid_exec:integer; pid_read:integer;
        source_path:string);
36  begin
37    if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
          23) and /* AUE_EXECVE or AUE_EXEC */
38      strToInt(return32_proc_value) != -1 and
39      strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
40      strToInt(subject32_PID) = pid_read and
41      match('^../usr/bin/sh$', path_0_path) = 1
42      --> trigger off for_current alert(auid_dest,
43                                        strToInt(subject32_audit_ID),
44                                        source_path);
45
46      strToInt(header32_event_ID) = 1 and /* AUE_EXIT */
47      strToInt(subject32_PID) = pid_exec
48      --> skip;
49
50      true --> trigger off for_next exec2(auid_dest, pid_exec,
51                                          pid_read, source_path)
52    fi
53  end;
54
55  rule alert(auid_dest:integer; auid_src:integer; source_path: string);
```

```
56 println('User ', auid_src, ' created shell via ', source_path,
57         ' with ', auid_dest, ' privileges').
```

Listing A.16: unix_suid_shell_script.asa

```
1  init_action;
2  trigger off for_next exec;
3
4  rule exec;
5  begin
6    if (strToInt(header32_event_ID) = 7 or strToInt(header32_event_ID) =
         23) and /* AUE_EXECVE or AUE_EXEC */
7      strToInt(return32_proc_value) != -1 and
8      (present attr_token_ID and is_suid(attr_file_mode) = 1) and
9      strToInt(attr_owner_UID) != strToInt(subject32_audit_ID) and
10     strToInt(subject32_user_ID) != strToInt(subject32_audit_ID) and
11     shell_script(path_0_path) = 1
12     --> trigger off for_current alert
13   fi;
14   trigger off for_next exec
15 end;
16
17 rule alert;
18 println(strToInt(subject32_audit_ID),': executing setuid shell script ',
19         path_0_path).
```

Listing A.17: unix_system_program_write.asa

```
1  init_action;
2  if init_policy = 1
3    --> trigger off for_next write;
4    true --> println('rule unix_system_program_write skipped')
5  fi;
6
7  rule write;
8  begin
9    if (strToInt(header32_event_ID) = 78 or strToInt(header32_event_ID) =
         79) and /* AUE_OPEN_WT or AUE_OPEN_WTC */
10     strToInt(return32_proc_value) != -1 and
11     strToInt(attr_owner_UID) != strToInt(subject32_audit_ID) and
12     strToInt(subject32_audit_ID) != 0 and
13     is_IXOTH(attr_file_mode) = 1 and
14     policy_member(path_0_path, 'RESTRICTED_WRITE_EXECUTABLE_FILES') = 1
15     --> trigger off for_next alert
16   fi;
17   trigger off for_next write
18 end;
19
20 rule alert;
21 println(strToInt(subject32_audit_ID), ': ', path_0_path,
22         ' written by user ', strToInt(subject32_audit_ID),
23         ' process ', strToInt(subject32_PID)).
```

Listing A.18: unix_unauth_delete.asa

```
1  init_action;
2  trigger off for_next delete;
3
4  rule delete;
5  begin
6    if strToInt(header32_event_ID) = 6 and /* AUE_UNLINK */
7      strToInt(return32_proc_value) != -1 and
8      strToInt(attr_owner_UID) != strToInt(subject32_audit_ID) and
9      strToInt(subject32_user_ID) != 0
10     --> trigger off for_current alert
11   fi;
12   trigger off for_next delete
13 end;
14
15 rule alert;
16 println(strToInt(subject32_audit_ID), ': ', path_0_path,
17         ' deleted by user ', strToInt(subject32_audit_ID),
18         ' owned by ', strToInt(attr_owner_UID)).
```