Formal specification of intrusion signatures and detection rules

Jean-Philippe Pouzol, Mireille Ducassé
IRISA / INSA de Rennes
Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{pouzol,ducasse}@irisa.fr

Abstract

1 Misuse intrusion detection systems detect signatures of attack scenarios. Existing systems are split into two categories: transition-based and declarative. In the transition-based systems what are the significant traces of attacks is hidden behind how they should be detected. This means that writing a signature is a very heavy task. In the declarative systems the signatures only contain what are the significant traces of attacks and an algorithm addresses how they should be detected. Writing signatures is thus much easier. However, the algorithm is a black box, and the security officer has no control over it.

In this article, we propose to refine the declarative approach. We formally specify the algorithm in two stages: firstly we classify the signature instances, secondly we give a detection rule set which detects in an audit trail a representative of each class. The rules are formally specified with “parsing schemata”, a high level formalism used to specify grammar parsers. The algorithm defined by the rules is proved sound and complete. With our approach, the what (signatures) and the how (detection algorithm) are still cleanly separated, but the security officer can possibly parameterize the detection by choosing a class for each signature.

1. Introduction

Misuse intrusion detection systems (IDS in the following) detect signatures of known attack scenarios. They can, among other classifications, be divided into two classes. They can match patterns on single events, or they can look for combinations of events. In single event IDS, even if the analyzer is very optimized and uses sophisticated data structures, the search algorithm is easy to understand from an abstract point of view: each event is compared with each known signature. Therefore, specifying signatures for single event IDS is quite simple. For example, signatures for the network IDS Snort are conjunctions of regular expressions that must be matched by the contents of a network packet [11].

Existing multi-event IDS do not have a uniform abstract algorithm if only because they do not all propose the same operators to combine events. These systems can be further split into two categories: transition-based and declarative. Examples of transition-based IDS are IDIOT a colored Petri nets simulator [6], STAT which manipulates state-transition diagrams [18], P-BEST (used in Emerald) a general purpose expert system [8], and ASAX which contains a specific rule based system [4]. In the transition-based systems what are the significant traces of attacks is hidden behind how they should be detected. This means that writing signatures is a very tricky task.

Recent work have proposed declarative solutions with higher level languages. For example, Lin et al. have developed MuSigs [7]; Roger and Goubault-Larrecq use linear time temporal logic [12]; Uppuluri and Sekkar defined “regular expressions for events” [17]; Gérard has proposed to compile a declarative language into ASAX signatures [3]; and we have presented a first approach to compile a declarative language into several of the transition-based IDS cited above [10]. In the declarative systems, the signatures only contain what are the significant traces of attacks and an algorithm addresses how they should be detected. Writing signatures is thus much easier. However, the algorithm is a blackbox, and the security officer has no control over it. This can become a real problem. Indeed, these systems detect all instances of attacks. Attackers can then easily choke the IDS by automatically generating hundreds of thousand incomplete instances of few attacks.

In this article, we propose to refine the declarative approach. We formally specify the algorithm in two stages: firstly we classify the signature instances, secondly we give a set of detection rules which detects in an audit trail a representative of each class. The rules are formally specified with “parsing schemata”, a high level formalism used to
specify grammar parsers [14]. The algorithm defined by the rules is proved sound and complete. With our approach, the what (signatures) and the how (detection algorithm) are still cleanly separated, but the security officer can possibly parameterize the detection by choosing a class for each signature.

The contribution of this article is twofold. Firstly, less instances of signatures are tracked, the IDS is therefore more resistant to choking attacks. Secondly, the detection algorithm is specified in a high-level formal way which makes it easy to understand and reason about, while essential operational features are nevertheless made explicit.

In the following, section 2 describes how to specify signatures with sequences and conjunctions of events correlated with logical variables. Section 3 provides the declarative semantics of signatures. Section 4 specifies class instances. Section 5 describes a detection algorithm that finds representatives of relevant instance classes. Section 6 sketches the soundness and completeness proofs. The remaining of the paper reviews related work and concludes.

2. Specification of Signatures

Intrusion signatures describe combinations of events that reflect a malicious activity. We call “filter” a signature on a single event. More complex signatures can be built in a compositional way. In this paper, we consider two constructs: the sequence and the conjunction. This section briefly presents how we abstract the log file, then the syntax of the language and a concrete example of specification are given.

Definition 1 (Event) An event is a collection of values identified by field names. We represent an event as a set of pairs (field name, value). We assume that in an event, a field name belongs to only one pair.

Example 1 The SUN BSM audit system [16] produces one event per system call performed on a Solaris machine. Here is an example of a partial event produced when a user with the user ID 42 executes the program /bin/su

\[ E_1 = \{ \text{eventID} = \text{EXEC}, \text{userId} = 42, \text{path} = \text{/bin/su} \} \]

Definition 2 (Trail) A trail is a totally ordered sequence of events. Given a trail \( T \), we note \( T[i] \) the \( i \)th event in the trail.

Now we define single event signatures (i.e. filters).

Definition 3 (Filter) A filter is a set of constraints between event field names, constant values and variable names. In this paper, we consider that constraints involving variable names can only be equality constraints, but this could be extended (in particular, [10] presents how non equality constraints with variables can be handled).

Example 2 The following filter matches the execution of any program not performed by the super-user. In this filter, the variables ProgName and User are respectively constrained to be equal to the path of the executed program and to the identifier of the user who performed the action.

\[ F_1 = \{ \text{eventID} = \text{EXEC}, \text{userId} \neq 0, \text{user} = \text{User, path} = \text{ProgName} \} \]

In the case of a signature with several filters, we distinguish two types of constraints: temporal constraints and valuation constraints. The temporal constraints describe the order in which events must be found in the trail. They are specified using the temporal connectors. Two connectors are considered in the paper: the sequence (i.e. total order) and the conjunction (i.e. partial order). The valuation constraints force some fields of several events to have the same value to be symptomatic of an intrusion. Such constraints are expressed using logical variables (as in Prolog). We propose here an abstract syntax for signatures.

Definition 4 (Signature) A signature is defined by a 5-tuple \((V, F, N_T, S, P)\) where \(V\) is a set of variables, \(F\) is a set of filters that use variables in \(V\), \(N_T\) is a set of non terminal elements, \(S \in N_T \) is the axiom, \(P\) is a set of production rules \( N_T \rightarrow \text{Prod} \), where \( P \in \text{Prod} \) can be:

- Filter\((F)\) where \( F \in F \)
- Seq\((A,B)\) where \( (A,B) \in N_T \times N_T \)
- And\((A,B)\) where \( (A,B) \in N_T \times N_T \)

Several signatures rules can be summed up into a single one. For example,

\[ N_1 \rightarrow \text{Seq}(N_{2A}[X], N_{2B}[Y]) \quad F[Z] = \{\ldots\} \]

is equivalent to

\[ F_X = F \text{ where occurrences of } Z \text{ are substituted by } X \]
\[ N_{2A} \rightarrow \text{Filter}(F_X) \]
\[ N_{2B} \rightarrow \text{Filter}(F_Y) \]
\[ F_Y = F \text{ where occurrences of } Z \text{ are substituted by } Y \]
### Notation

In the remaining of this document, we note \( \mathcal{V}(S) \) the set of variables \( \mathcal{V} \) present in the definition of the signature \( S \).

### Example 3

A bug in the \texttt{ff.core} program distributed with Solaris 2.6 allows any user to rename any file (even if unauthorized by the file system protection) with the restriction that the target name must be in the same directory than the original file [2]. Renaming a file \( f1 \) to \( f2 \) using this bug requires two operations and can be done as follows:

(a) `ln -fs f1 /vol/rmt/diskette0`

(b) `ff.core -r /vol/rmt/diskette0 f2`

The scenario described in [5] uses this vulnerability to create a back-door on a machine. The scenario consists of several steps: (1) rename the file \texttt{/usr/bin/sh} to \texttt{/usr/bin/admintool}; (2) rename the file \texttt{/usr/sbin/swamtool} to \texttt{/usr/sbin/in.rlogin.d}; (3) connect on the \texttt{rlogin} port of the machine. Note that steps 1 and 2 can be in reverse order, or performed in parallel. The key point in this attack is that \texttt{swamtool} is a link to \texttt{admintool}. Then, when a connection is opened on the machine, the \texttt{in.rlogin} daemon (usually running as \texttt{root}) executes the program \texttt{in.rlogin.d}, which, in these conditions, a link to \texttt{admintool}, which is in fact \texttt{sh}. Then, anybody can obtain a root shell on the \texttt{rlogin} port of the machine.

Figure 1 presents a signature for this scenario. The grammar axiom \( \text{Sig} \) specifies that the renaming operations (non-terminal \texttt{Move}) have to be performed before the execution of \texttt{in.rlogin.d} by the super-user (non-terminal \texttt{Login}). The non-terminal \texttt{Move} specifies an interleaving of the two renaming operations (non-terminal \texttt{Mv}[p, u]). Finally, the non-terminal \texttt{Mv}[p, u] is a sequence of two filters: the link creation and the execution of \texttt{ff.core} by the same user. Note that this signature does not take into account the name of the target file in the arguments of the \texttt{EXEC} system call in \texttt{ff.core}. It is generally considered that auditing these arguments is too heavy to be viable. However, with a more precise study of the exploit, it may be possible to refine the signature. Indeed, tracing the system calls generated by \texttt{ff.core} shows that the target name is used in a \texttt{RENAME} system call. Then the signature can be extended by looking for such a system call performed with the same process ID than \texttt{ff.core}.

In this signature, we specify that the two commands required to rename a file must be performed by the same user. Moreover, using the two variables \( U_1 \) and \( U_2 \), we also specify that the two renaming operations can be done by distinct users. This could be easily changed by using a single variable to specify that all operations must be done by the same user, or by using four distinct variables to handle the case where four users make a coordinate attack.

### 3. Semantics of Signatures

This section presents the meaning of a signature by formally defining what is an instance of this signature in a trail. We first distinguish concrete instances from their abstraction used in the semantics, then we give semantics rules.

An instance of a signature is a collection of events that fulfill the constraints in filters (b) with respect to the correlation specified by the logical variables (c) and are in a correct order according to the temporal constraints. Such a collection is called a concrete instance of the signature and is noted \( \{p_1, ..., p_n\} \) where all \( p_i \) are positions in the trail.

The proposed semantics abstracts concrete instances into a triplet \((\theta, i, j)\) where \( \theta \) is a signature valuation (see defini-
Definition 5 (Signature constraints) We call “signature constraint” a set of constraints that force some properties on the possible values of the variables used in a signature. Note that, as opposed to filters, a signature constraint cannot contain any reference to a field name since it is a property that may concern several events.

Example 4 Here is a signature constraint for the signature in figure 1

$$
\theta = \begin{cases} 
U_1 = 42, \\
U_2 \neq 21, U_2 \neq 33, U_2 \neq 17 
\end{cases}
$$

Definition 6 (Signature valuation) A signature constraint is said to be a “signature valuation” if it forces a unique value for each variable that appears in the signature.

Example 5 Here is a valuation of the signature in figure 1

$$
\theta = \{U_1 = 42, U_2 = 28\}
$$

Note that the constraint in example 4 is not a valuation of this signature.

We can now define the meaning of matching an event and a filter, this will be the base case of our inductive definition of the semantics.

Definition 7 (Event matching) We define the predicate $\text{match}(E, F, \theta)$ where $E$ is an event, $F$ is a filter, and $\theta$ a valuation of $F$. This predicate holds if and only if the constraint set $(F_E \cup \theta)$ (where $F_E$ is obtained by replacing field names in $F$ according to $E$) admits at least one solution. This predicate does not hold if an event field name used in $F$ is not present in $E$.

The semantics of the language is given by means of relation $\models$. Given a signature $S$, a trail $T$ and an abstract instance $I = (\theta, i, j)$, $I$ is an instance of $S$ in $T$ if and only if $T, \theta, i, j \models S$. The relation $\models$ is inductively defined on the structure of the signatures as shown in figure 2.

1. **Filter**: an instance of $\text{Filter}(F)$ exists in $T$ at the position $i$ if and only if the predicate $\text{match}$ holds for this event.

2. **Sequence**: an instance of a sequence exists in $T$ between positions $i$ and $j$ if an instance of $A$ starts at position $i$, if an instance of $B$ ends at position $j$, and if these two instances do not overlap.

3. **Conjunction**: an instance of a conjunction exists in $T$ between positions $i$ and $j$ if there exists instances of both parts of the conjunction, if one starts on $i$, and if one ends on $j$.

$$
T, \theta, i, j \models \text{Filter}(F) \iff (j = i + 1) \\
\wedge \text{match}(T[i], F, \theta)
$$

$$
T, \theta, i, j \models \text{Seq}(A, B) \iff \exists a, b \ (T, \theta, i, a \models A) \\
\wedge (T, \theta, b, j \models B) \\
\wedge (i < a \leq b < j)
$$

$$
T, \theta, i, j \models \text{And}(A, B) \iff \exists a, b, c, d \ (T, \theta, a, b \models A) \\
\wedge (T, \theta, c, d \models B) \\
\wedge (i < a < b < j) \\
\wedge (i < c < d < j) \\
\wedge (j = \text{min}(a, c)) \\
\wedge (j = \text{max}(b, d))
$$

Figure 2. Semantics of signatures

4. Specification of Signature Instances

Many approaches to perform misuse intrusion detection have striven to be both sound and complete with respect to the signature specifications. While soundness is obviously required, we argue that completeness is not always needed, in some cases it can even become a weakness of the IDS. Trying to report all instances of a signature raises the capabilities of a malicious user to directly attack the IDS. Indeed, IDS must be able to check an audit trail “on the fly” (mainly because log files are too large to be entirely stored) and they cannot backtrack during the search. As a consequence, they have to look for several instances of the same signature in parallel and this can be exploited to put the IDS in stressful conditions. For example, consider the primary $\text{ff.core}$ exploit and the sequence of two operations needed to achieve it. The first command is a link creation. Suppose that an attacker creates several times the same link. There will exist in the trail as many instances of the intermediate signature as the number of times the attacker has reiterated the operation. While he does not execute $\text{ff.core}$, there is no instance of the complete signature, and therefore no alert is raised by the IDS. Writing few lines of program, this attacker can create several hundreds of thousand unachieved instances without any alarm, and then saturate the memory of the machine that hosts the IDS in few minutes. We call such a deny-of-service attack against the IDS a “choke attack”.

Reporting all instances of a signature is thus not always needed. To overcome the problem mentioned above, we propose an approach to specify what instances are relevant for the detection. These specifications are expressed as equivalence relations between instances of each signature. Once this classification is done, the IDS can report only particular instances in each class.

When signatures are expressed using a declarative language with logical variables, it is possible to group in-
instances according to the variable valuations. In this paper, given a signature, an equivalence relation is specified by choosing an element in the lattice of all the subsets of the variables of the signature. Two instances are said to be equivalent if they contain the same values for the variables in this subset.

Consider a signature with a set of variables \( V = \{U_1, U_2, T_1, T_2, T_3, T_4, T_5\}\). Figure 3 shows a partial view of the lattice of all the subsets of \( V \). The lowest element of this lattice is the empty set; a new element noted \( \top \), greater than \( V \) added been added. Each element \( e \) in this lattice corresponds to an equivalence \( \mathcal{R}(e) \) between instances.

- \( \mathcal{R}(\{\}\) = \( \lambda I_1 \cdot \lambda I_2 \cdot \text{true} \) All instances are in the same class.
- \( \mathcal{R}(\top) = \lambda I_1 \cdot \lambda I_2 \cdot \text{false} \) Each instance is in its own class.
- For any other element \( e = \{V_1, ..., V_n\} \) of the lattice,

\[
\mathcal{R}(e) = \lambda I_1 \cdot \lambda I_2 \cdot \left( I_1(V_1) = I_2(V_1) \land \ldots \land I_1(V_n) = I_2(V_n) \right)
\]

where \( I(V) \) is the value of variable \( V \) in the instance \( I \). Here, instances are grouped according to the valuations of variables present in \( e \).

Whereas the aim of alert correlation is to reduce a posteriori the number of reported alerts, the motivation to select instances in our context is to provide a criterion that will be used by the IDS to prune search paths on the fly. If the algorithm does not perform this pruning, it may not resist the choking attack described above.

The IDS will ignore events that would produce instances with a valuation equivalent to another one already seen during the search. However, performing such a pruning with any equivalence class in the lattice could, in some cases, make the IDS miss some relevant instances.

To exhibit the correct equivalence relations, we must first notice that two motivations can lead to use of variables in signatures. They can be used in several filter to correlate events. We call such variables “correlation variables”. They can also be used to report data when an alert is raised and then appear only once in a signature. Such variables are called “information variables”. For example, in the signature \( \text{Sig} \) proposed in section 2, the variables \( U_1 \) and \( U_2 \) are correlation variables, and there is no information variable. This signature can be refined by adding in each filter a variable \( T_i \) to store the time stamp of each event in the instance (notice that this gives means to reconstruct a concrete instance from an abstract instance). All these new variables are information variables.

An element in the lattice can be selected to build an equivalence relation only if it is greater than the subset of all correlation variables. Choosing an element that does not contain all correlation variables will let the algorithm ignore some events that may be part of relevant instances. If we have choose \( \mathcal{R}(\{\}\) to perform the search, two attacker can deceive the IDS if one starts the scenario, does not complete it, and if the other then restarts from the beginning but this time achieves the attack. In this case, the IDS will only follow the first path and will miss the second attack that is the real one.

Even if the whole lattice cannot be considered when choosing a subset of variables, this approach gives some flexibility. The partial view of the lattice in figure 3 actually corresponds to the signature \( \text{Sig} \) extended with the information variables \( T_i \). In this lattice, only elements greater than \( \{U_1, U_2\} \) can be chosen. Suppose that the algorithm is designed to report a single instance for each class, here are several examples with different equivalence relations:

- \( \mathcal{R}(\top) \): since each instance is in its own class, the search will be exhaustive and all instances will be reported. This choice does not resist the choking attack depicted in the beginning of the section.

- \( \mathcal{R}(\{U_1, U_2, T_1, T_2, T_3, T_4, T_5\}) \): In this particular case, since there is a variable \( T_i \) in each filter, and since two timestamps cannot have the same value, choosing this equivalence relation will lead in this case to the same search as the one performed when choosing \( \mathcal{R}(\top) \).

- \( \mathcal{R}(\{U_1, U_2\}) \): with this choice, the IDS will report only one instance for each pair of users \( (U_1, U_2) \) that perform the attack. The algorithm will better resist...
the choking attack than when performing an exhaustive search.

- \( R(\{U_1, U_2, T_3\}) \): this choice is more interesting. Indeed, even if we are mostly interested by a single instance for each pair of users (for example, to close their account), it may be important to know all the moments where the login has succeeded because this corresponds to the real intrusions. Considering that \( T_3 \) is the time stamp of the last event of the signature, adding this variable in the equivalence relation gives this information. Here again, the algorithm will resist to the choking attack described above.

Once the classification into equivalence classes is done, the next decision to take is to specify which instance in each class must be reported by the intrusion detection algorithm. Several answers can be proposed. Chakravarthy et al. studied several possible strategies in the context of composite event languages for active databases [1]: reporting the instance that starts first and ends first, or reporting the one that starts last among all the ones that end first, or reporting the shortest instance for each event that starts an instance. In this paper, we consider the first strategy: reporting the instance that starts first and ends first. Firstly, finding the instance that ends first is required when analyzing an infinite trail. Secondly, finding the one that starts first seems simpler because constraining further search is easier than cancelling previous results. In the remaining of the paper, this strategy is referred as the "First strategy".

We define the predicate \( \text{First}(S, \rho, T, \alpha, (i, j, \theta)) \) where \( S \) is a signature, \( \rho \) is an equivalence relation between instances of \( S \), \( T \) is a trail, \( \alpha \) is a position in \( T \) and \( (i, j, \theta) \) is an instance of \( S \). Given an instance \( I = (\theta, i, j) \) with \( \alpha \leq i \), this predicate holds if and only if, among all instances equivalent to \( I \) according to \( \rho \) that start after \( \alpha \), \( I \) is the one that starts first and ends first. Formally,

\[
\text{First}(S, \rho, T, \alpha, (i, j, \theta)) = \\
(\alpha \leq i < j) \\
\land \ (T, \theta, i, j \models S) \\
\land \ \forall (x, y, \theta') \left( \\
\left( \begin{array}{c}
\alpha \leq x < y \\
\land T, \theta, x, y \models S \\
\land \rho(\theta', x, y, (\theta, i, j))
\end{array} \right) \Rightarrow (x \geq i \land y \geq j) \right)
\]

### 5. "First" Algorithm

We propose an algorithm to find instances of multi-event signatures with logical variables. Each signature is assigned an equivalence relation between instances, and the algorithm implements the \( \text{First} \) strategy described in the previous section. This algorithm is described with a formalism called \textit{parsing schemata}.

This formalism specifies algorithms using a set of deduction rules. It gives a formal framework to describe the algorithm and to formally prove some of its properties. It also provides an intermediate level of description (i.e. it is more precise than the declarative specification of signatures, but many cumbersome implementation details are hidden). Finally, this description is modular: one does not need to know the whole specification to understand how is performed the search for a particular construct of the language.

Section 5.1 presents a short introduction to the formalism and section 5.2 details the algorithm. The proofs of soundness and completeness are exposed in section 6.

#### 5.1. Parsing Schemata

"Parsing schemata", a high-level framework to describe parsing algorithms, has been defined by Sikkel [14]. It is the continuation of the parsing-as-deduction approach initially proposed by Pereira and Warren in [9] and extended by Shieber et al [13]. Within this framework, parsing algorithms are described as a set of deduction steps. The hypotheses and conclusions of these steps are elements called \textit{parsing items} that represent partial or complete parsing trees. The deduction generally starts with an item representing an empty parsing tree and the deduction ends when an item representing a complete parsing tree of the axiom of the grammar is produced. This framework has been used to formalize different strategies to parse context free grammars [15] and has been extended to more complex languages like unification or natural language grammars [13].

The first step when formalizing an algorithm in this framework is to define the domain of items. Our algorithm uses “dotted” items of the following form: \( [i, \alpha \bullet \beta, j] \theta \) where \( (i, j) \) are positions in the trail, \( \alpha \bullet \beta \) is the right hand side of a grammar production where \( \bullet \) has been inserted, and \( \theta \) is a signature constraint. Here are intuitive meanings for several examples:

\[
[i, \bullet S, i] \theta \quad \text{: the algorithm looks for an instance of } S \text{ from position } i \text{ under the constraint } \theta
\]

\[
[i, \text{Seq}(A \bullet B), j] \theta \quad \text{: an instance of } A \text{ has been found between } i \text{ and } j \text{ under the constraint } \theta. \text{ B still needs to be found to produce an instance of Seq}(AB)
\]

\[
[i, \text{And}(AB)\bullet, j] \theta \quad \text{: an instance of And}(AB) \text{ has been found between } i \text{ and } j \text{ with the valuation } \theta.
\]

The deduction steps are described by several set transformation rules. Given a set of items, these rules can add
new items and/or retract existing ones. Applying a rule only once in all the grammar rules. This can also be obtained with simple rewriting rules. Finally, since equivalence relations can differ from one signature to another, all variables in the signature constraint with the values in the expanded. Secondly, a non-terminal element must be used to match the filter, the algorithm goes one event forward in the trail.

Figure 4. Deduction rules for filters

\[
\begin{align*}
\text{Filter}_1: & \quad \theta \left[ i, \text{Filter}_\rho(F), i \right] \theta \\
& \quad \{ \text{match} (T[i], F, \theta) = \text{false} \}
\end{align*}
\]

\[
\begin{align*}
\text{Filter}_2: & \quad \theta \left[ i, \text{Filter}_\rho(F), i + 1 \right] \theta \\
& \quad \{ \text{match} (T[i], F, \theta) = \text{true} \}
\end{align*}
\]

5.2. Description of the algorithm

Preliminary assumptions The algorithm makes several assumptions on the specifications. Firstly, signatures that use the [ ] notation presented in definition 4 have to be expanded. Secondly, a non-terminal element must be used once only in all the grammar rules. This can also be obtained with simple rewriting rules. Finally, since equivalence relations can differ from one signature to another, all filters must be labeled with the equivalence relation associated to the signature (a filter is then noted: Filter_\rho(F) where \rho is the equivalence relation).

Before defining the whole set of deduction rules of our algorithm, we introduce several operators that are used when an event matches a filter. The Propag operator unifies the variables in the signature constraint with the values in the event.

Definition 8 (Propag operator) Given an event E, a filter F and a signature constraint \theta, we define Propag(E, F, \theta) to be a new signature constraint obtained by (1) copying F into F', and removing all constraints with no variable in F', (2) substituting all field names in F' according to E, and (3) making the union F' \cup \theta.

Since this operator is only used when the predicate match holds, the field names substitution in F is always possible.

The Restrict operator creates a new constraint that will force the algorithm to prune some paths in the search. When the relation is \mathcal{R}(\{ \}), all instances are in the same class, then as soon as a sub-signature is found, the research of this component is stopped. When the relation is \mathcal{R}(\mathcal{T}), all instances are in different classes (i.e. the search is exhaustive), then we add no constraint to prune paths. When the relation is \mathcal{R}(\{V_1, V_n\}) further sought instances must have a valuation different for at least one variable V_i.

Definition 9 (Restrict operator) Given a signature \mathcal{S}, a valuation \theta of \mathcal{S} and an equivalence relation \rho, we define Restrict(\rho, \theta) to be a constraint computed as follows:

\[
\text{Restrict}(\rho, \theta) = \begin{cases} \rho \text{ in } \mathcal{R}(\{ \}) : \text{return FALSE} \\
\mathcal{R}(\mathcal{T}) : \text{return TRUE} \\
\mathcal{R}(\{V_1, \ldots, V_n\}) : \text{return } \bigvee_{i=1}^{n} (V_i \neq \theta[V_i]) \end{cases}
\]

where TRUE (resp. FALSE) is a constraint that is always (resp. never) satisfiable, and where \theta[V_i] is the value assigned to V_i in \theta.

The last operator compares signature constraints.

Definition 10 (Constraint comparison (\succ_\mathcal{S})) Given a signature \mathcal{S} and two signature constraints \theta_1 and \theta_2, we define the total order \succ_\mathcal{S} as follows: \theta_1 \succ_\mathcal{S} \theta_2 if and only if the set of possible values for each element of Var(\mathcal{S}) described by \theta_1 includes the one described by \theta_2.

Figure 4 shows the two deduction rules for filters. The antecedent of both these rules is the item [i, Filter_\rho(F), i] \theta corresponding to a search of an event that matches filter F from position i under constraints in \theta.

- Rule Filter_1 specifies that, if event T[i] cannot be use to match the filter, the algorithm cannot prune the path in the trail.
- Rule Filter_2 handles the other case. Here, the rule produces two items. The first one memorizes that an instance of F is found at position i, with the valuation \theta_1. The constraint attached to this item is the result of applying Propag to take into account that some variables can be instantiated here. The second created item starts the search of a new instance of F in the remaining part of the trail. This new search can be more constrained than the one that has produced this item according to the result provided by Restrict.
Figure 5 shows the deduction rules for the sequence and the conjunction constructs. Three rules encode the search of the sequence.

- **Rule Seq1.** This rule starts the search of the first part of the sequence with item \([i, \bullet A, i] \theta\).

- **Rule Seq2.** Once an instance of the first part of a sequence is found, the rule Seq2 replaces item \([i, \bullet A, j] \theta\) with the item \([i, Seq(A \bullet B), j] \theta\) to memorize that one half of the sequence has been found. An item \([j, \bullet B, j] \theta\) is added to start the search of the second half of the sequence from the position where the instance of A ends.

- **Rule Seq3.** When an item corresponding to an instance of the second part of a sequence is deduced, the algorithm must match it with an item corresponding to the first part of the sequence. This explains the two side conditions \(j \leq k\) (the first part must end before the second one starts) and \(\theta_1 \supseteq (Seq(AB)) \theta_2\) (the constraint of the second part must refine the constraint of the first part). Note that this rule does not remove the item \([i, Seq(A \bullet B), j] \theta_1\) because it may be needed in the future if another item \([x, B \bullet y] \theta^\prime\) with \(j \leq x\) and \(\theta_1 \supseteq \theta^\prime\) is deduced. Suppose that signature Seq(AB) uses two variables \(X\) and \(Y\) and that the search is performed with \(R(\{X, Y\})\). If the item \(\delta = [i, Seq(A \bullet B), j] \{X = 1\}\) has been deduced, and if there exists two instances of \(B\) with \(\theta_p = \{X = 1, Y = 1\}\) and \(\theta_q = \{X = 1, Y = 2\}\) after position \(j\) in the trail, \(\delta\) has to be paired with both \([x_p, B \bullet y_p] \theta_p\), \([x_q, B \bullet y_q] \theta_q\) to produce the two instances of Seq(AB).

Figure 5. Deduction rules for sequence and conjunction

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| Seq1 | \[i, \bullet Seq(A \bullet B), i] \theta \rightarrow [i, \bullet A, i] \theta \]
| Seq2 | \[i, Seq(A \bullet B), j] \theta \rightarrow [j, \bullet B, j] \theta \{S \rightarrow Seq(A \bullet B)\} \]
| Seq3 | \[i, Seq(A \bullet B), j] \theta_1, [k, B \bullet l] \theta_2 \{j \leq k\} \rightarrow [i, Seq(A \bullet B), l] \theta_2 \{\theta_1 \supseteq (Seq(AB)) \theta_2\} \]

And 1 | \[i, \bullet And(A \bullet B), i] \theta \rightarrow [i, \bullet A, i] \theta \]

And 2 | \[j, \bullet A, k] \theta_1, [l, B \bullet m] \theta_2 \rightarrow [min(j, l), \bullet And(A \bullet B) \bullet, max(k, m)] \theta_12 \{S \rightarrow And(A \bullet B)\} \}

\(\theta_{12} = (\theta_1 \cup \theta_2)\) is satisfiable

Figure 6. Deduction rules to fold/unfold grammar rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| Predict | \[i, \bullet A, i] \theta \rightarrow [i, \bullet A, i] \theta \{A \rightarrow \alpha\} \]
| Reduce | \[i, \bullet A, j] \theta \rightarrow [i, \bullet A, \bullet, j] \theta \{A \rightarrow \alpha\} \]

The rules for the conjunction are similar to the ones for the sequence.

- **Rule And1.** This rule starts the search of both parts of the conjunction.

- **Rule And2.** When two parts of a conjunction are found, if their respective constraints are compatible, a new item is created to notify that an instance of the conjunction is found.

We finally add the two rules depicted in figure 6 to fold/unfold grammar rules when the dot is just before or just after a non-terminal element of the grammar. Note that these rules are obvious because we have taken care to rewrite signatures such that a non-terminal cannot appear twice in a signature.

### 6. Soundness and completeness

This section gives a sketch of the proof that the algorithm is both sound and complete with respect to the declarative semantics of the language extended with the search strategy.
Before setting forth the theorems, one must notice that, since non-terminal elements are not shared between signa-
tures, if the theorems of soundness and completeness are proved for the search of a single signature, they are also
valid if the algorithm looks for instances of several signa-
tures in parallel.

**Theorem 1 (Soundness theorem)** Given a trail \( T \) and a
signature \( S \) where all filters are labeled with an equiva-
lence relation \( \rho \), when starting the search at position \( i \) in
\( T \), if the item \( [j, S \bullet, k] \theta \) is produced, then there exists in \( T \)
an instance of \( S \) between \( j \) and \( k \) with valuation \( \theta \). More-
over, this instance is the one that corresponds to the First
strategy.

\[\forall (i, j, k, \theta, \theta'). \begin{cases} i \leq j \\
\theta' \equiv S \theta \\
[i, S, i] \theta \rightarrow^T \Gamma, [j, S, k] \theta \\
(T, \theta, j, k \vdash S) \land \text{First}(S, \rho, T, i, j, k, \theta) \end{cases} \]

**Theorem 2 (Completeness theorem)** Given a trail \( T \) and a
signature \( S \) where all filters are labeled with an equiva-
lence relation \( \rho \), if (a) an instance of \( S \) with a valuation \( \theta \)
exists between \( j \) and \( k \), and if (b) this instance is the one that corresponds to the First strategy, and if (c) the algorithm
starts the search from a position \( i \) with a signature constraint \( \theta' \) more general than \( \theta \), then the algorithm will
deduce the item \( [j, S \bullet, k] \theta'' \) where \( \theta'' \) and \( \theta \) are the same valuation for the variables of \( S \).

\[\forall (i, j, k, \theta, \theta'). \begin{cases} i \leq j \\
\theta' \equiv S \theta \\
(T, \theta, j, k \vdash S) \land \text{First}(S, \rho, T, i, j, k, \theta) \end{cases} \]

**Remark on the completeness theorem** It can be proved that,
when the algorithm deduces an item of the form
\([i, S \bullet, j] \theta\), the constraint \( \theta \) is a valuation constraint for \( S \)
(i.e. \( \theta \) forces a unique possible value for each variable in \( S \)).
The proof of this property is not shown in the paper. It
can be obtained by induction on the structure of the signa-
ture. In the base case, when matching filters, the \text{Propag}
operator instantiates in the constraint all the variables used
in the filter. Therefore, in the completeness theorem, be-
cause \( \theta'' \) and \( \theta \) are valuations of \( S, \theta'' \equiv S \theta \) implies that \( \theta'' \)
and \( \theta \) respectively assign the same values to the variables of \( S \).

Since the semantics of the signature language and the
search algorithm are defined in an inductive way, all proofs
are done by induction on the depth of a signature, noted \( D \),
where:

\[ D(S) = \begin{cases} 1 & \text{if } S = \text{Filter}(F) \\
\max(D(A), D(B)) + 1 & \text{if } S = \text{Seq}(A, B) \text{ or And}(A, B) \end{cases} \]

### 6.1. Proof of soundness

We give here a sketch of the inductive proof on the depth
of the signature

**Case:** \( S = \text{Filter}(F) \) If an item \([j, \text{Filter}(F) \bullet, k] \theta \)
is deduced by rule \( \text{Filter}_2 \), then event \( T[j] \) matches filter \( F \),
and there exists \( \theta'' \) such that \( \theta = \text{Propag}(E, F, \theta'') \); we can then conclude that \( T, \theta, j, k \vdash \text{Filter}(F) \). We can also be
sure that this event is the first instance within its class from
position \( i \) in the trail. If another one had existed before in
the trail, the rule \( \text{Filter}_2 \) would have restricted the search
to the according to \( \rho \), and \([j, \text{Filter}(F) \bullet, k] \theta \) would not have been
deduced.

**Case:** \( S = \text{Seq}(A, B) \) Considering the deduction rules,
we deduce that there exists an instance of \( A \) between \( j \) and \( a \),
and an instance of \( B \) between \( b \) and \( k \). Considering the
semantics of the sequence, we conclude that there is an
instance of \( \text{Seq}(A, B) \) between \( j \) and \( k \). Moreover, we can also deduce from the induction hypothesis that this instance
of \( A \) is the first of its class from the position \( i \) in the trail.
And since the search of \( B \) cannot be started before this
instance of \( A \) is found, this instance of \( B \) is the first in its class
from position \( a \) in the trail. Then, we conclude that this
instance of \( \text{Seq}(A, B) \) is the first in its class from position \( i \) in the trail.

**Case:** \( S = \text{And}(A, B) \) The proof for the conjunction
is very close to the one for the sequence. An item
\([j, \text{And}(A, B) \bullet, k] \theta \) can be deduced, only if two items
\([a, A \bullet, b] \theta_1 \) and \([c, B \bullet, d] \theta_2 \) with \( i = \min(a, c), j = \max(c, d) \)
and \( \theta = \theta_1 \cup \theta_2 \) have been deduced before in
the sequence of deduction steps. Here again, applying twice the
induction hypothesis, we conclude that there exists an instance of \( \And(A \land B) \) between positions \( j \) and \( k \) in the trail, and this instance is the first of its class from position \( i \) in the trail.

### 6.2. Proof of completeness

The completeness proof is also an induction on the depth of the signature.

**Case \( S = \text{Filter}(F) \)** Suppose there exists \( T, \theta, j, k \models \text{Filter}(F) \) such that this is the first instance of its class from position \( i \). Since this instance is the first of its class, matching events with \( \text{Filter}(F) \) between position \( i \) and \( j \) cannot generate a constraint that is in contradiction with the event at position \( j \). Then the item \([j, \text{Filter}(F) \land, k] \theta \) will be produced by the algorithm.

**Case \( S = \text{Seq}(A \land B) \)** Suppose there exists \( T, \theta, j, k \models \text{Seq}(A \land B) \) such that this is the first instance of its class from position \( i \). Then, there exists an instance of \( A \) (resp. \( B \)) between position \( j \) and \( a \) (resp. \( b \) and \( k \)) with \( j < a \leq b < k \). It can be proved that these instances of \( A \) and \( B \) are respectively the first ones of their class. This is proved by reducing it to the absurd: if these instances of \( A \) and \( B \) are not the first instances of their respective classes, the instance of \( \text{Seq}(A \land B) \) cannot be the first of its class. Then we can apply twice the induction hypothesis and deduce that items \([j, A \land, a] \theta' \) and \([b, B \land, k] \theta \) with \( a \leq b \) and \( \theta' \succcurlyeq \theta \) will be deduced by the algorithm. Finally, we conclude that \([j, \text{Seq}(A \land B), k] \theta \) will be deduced by the algorithm.

**Case \( S = \And(A \land B) \)** The proof of completeness for the conjunction is the same as the one for the sequence. The existence of the instance of the conjunction implies the existence of instances of both parts of the conjunction. It can be proved that these two instances are the first ones of their respective classes. Applying the induction hypothesis twice let us conclude that the algorithm will produce the item corresponding to the instance of the conjunction.

### 7. Related work

Two approaches can be considered to build an IDS that can handle high level declarative specifications: developing a system that directly manipulates such specifications, or translating declarative expressions into signatures for existing IDS. These two ways have been studied in the literature.

Lin, et al. have described within the ARMD project a declarative language called MuSigs and have developed an algorithm based on relational algebra to perform the detection [7]. MuSigs provides the same syntactic constructs as the language described in this paper: the sequence and the conjunction. Roger and Goubault-Larrecq have proposed to express specifications using linear time temporal logic formulas with first order variables [12]; the detection is performed with an on-line model-checking algorithm.

Gérard has shown how to translate a declarative language named LaDAA into ASAX signatures [3]. This language provides a sequence, a conjunction and a disjunction. The translation is based on the construction of a non-deterministic finite automaton extended with variables. A similar approach has been studied by Uppuluri and Sekar in [17] where signatures are expressed using “regular expressions for events”. Finally we proposed [10] an extended version of the language described in this paper and have described a first approach to translate declarative specifications into signatures for several rule based IDS. This transformation uses, as intermediate virtual machine, a state-transition diagram close to the one used in STAT [18].

All these algorithms cannot be really compared because the expressiveness of the languages slightly differ from one proposition to another. However, all these languages have been proposed to hide operational considerations during signature specification, and they all come with an algorithm that can be ignored when writing signatures. As a consequence, no means to interact with the search algorithm is provided, and actually, all these algorithms perform an exhaustive search of all instances of the signatures. Then they may less resist to the choking attack described in section 4 than the algorithm we propose.

One must notice that Roger and Goubault-Larrecq, in order to improve the efficiency of their first proposal, have suggested in [12] an alternative to the exhaustive search. Firstly, they restrict the language and only consider flat “Wolper-style” formulas. Secondly, for each event that constitutes the beginning of an instance, their tool reports the shortest instance from this position. We argue that restricting the language, and choosing to report only a subset of all instances are two distinct choices that must be made separately. It seems that the authors have chosen this combination because the trade-off between expressiveness and efficiency is acceptable. In our approach, since we have cleanly separated these two considerations, we are able to prune paths in the search without restricting the expressiveness of the language. As an example, the signature depicted in section 2 cannot be expressed in the “Wolper-style” formulae, and we think it is a serious lack of expressiveness. Indeed, in many cases, the first steps of an attack scenario are not totally ordered, whereas the last event must be performed after all the others.
8. Conclusion

In this article we have described how to specify signatures with sequences and conjunctions of events correlated with logical variables. We have given a declarative semantics to these signatures. We have also introduced signature instance classes based on the valuation of variables of interest. We have given a formal description of a detection algorithm which is able to track only representatives of instance classes. It is therefore more resistant to choking attacks than most of the existing algorithms which track all signature instances. The parsing schemata used to specify the detection algorithm make it easy to understand and reason about it, while essential operational features are nevertheless made explicit.

Acknowledgments Many thanks to Marc Eluard for fruitful discussions, and to Jean Goubault-Larrecq for the additional information he has provided about his model-checking approach.

References