ISSN:0975-9646

Access Control Policy: A Framework to Enforce Recommendations

Nada Essaouini , Anas Abou El Kalam , Abdellah Ait Ouahman

Cadi Ayyad University, ENSA of Marrakesh – OSCARS Laboratory
Avenue Abdelkarim Khattabi, Guéliz Marrakech, Maroc

***Abstract*— Access control policies are generally modelled using permission, prohibition, and obligation rules. However, this does not cover all possible scenarios as several applications have recommendation rules. In this paper, we provide a formal framework to express and to enforce recommendations. More precisely, our framework allows to express recommendation rules that become requirements over time. Furthermore, we give the specification of the policy controller behavior in charge of evaluating such a policy. Basically, in our formalization, a recommendation is asso ciated with three conditions. The -first one triggers the recommendation: when this condition is true, a notification is sent to the user to carry out an action satisfying the recommendation. The second condition is the recall deadline that determines when the next notification will be sent if the user has not perform the access satifying the recommendation. The third condition determines when a recommendation could become a requirement.**

***Keywords*— Information systems security, access control policy, temporary logic of actions**

I. INTRODUCTION

As organizations depend on their information systems (IS) and as theses IS are more and more vulnerable and open to the internet, security has become essential and unavoidable.

Insecurity has in fact been proved costly in case of incidents, -malfunctions or -failure.

To overcome such incurred risks, in a specific organizational context, we generally identify what needs to be protected, quantify the corresponding issue, formulate security goals and identify, arbitrate and implement adapted parades at the right maintained level. This goes primarily through the formulation and implementation of the security policy within an organization.

Basically, a security policy is developed in three areas: physical, administrative and logical. The first specifies the physical aspects and environments of the system to protect. (e.g., procedures and mechanisms taken to overcome thefts and physical disasters). The second describes the organizational procedures (e.g., separation of duties). The third refers to the logical access control, which is based on a triple service of identification, authentication and authorization. In fact, before using the system, any user must identify himself (identification) and prove its identity (authentication). Once the relationship is established, legitimate actions that the user can do are determined by authorization policy (also known as access control policy).

Basically, this policy specifies who ha access to what, when and in which condition? It of course must be consistent and must comply with laws and regulations.

In general, the rules of security policy are specified in terms of permissions (e.g., every attending physician has the right to access medical records of his patients) and prohibitions (e.g., physicians do not have the right to delete diagnosis already established), but also in terms of obligations (e.g., physicians are obliged to keep medical records for the period determined by law). However, in most information systems, we find guidance and rules in the form of recommendations as in the General Security Referential [1]. This document and its annexes have been drafted jointly by the National Security Information Systems Agency of France and -the General Headquarters for the Modernization of the State of France in a legal framework. It defines a set of safety rules imposed on the administrative authorities in securing their information systems and also provides a good practice in the security of information systems that administrative authorities should apply. These good practices are typically formulated as recommendations. Let us take the example of the document entitled: “rules and recommendations regarding the selection and design of cryptographic mechanisms” [2] (that supplements the annex of the general security referential). In this document, obligations are preceded by the word “Rule” (e.g., RuleAuthenticityVerification stipulates that before using a key in an application system, its authenticity must be verified by a safety mechanism according to the repository) ; while recommendations are preceded by the word “Recom” (e.g., RecomEndUtilisationReason stipulates that it is recommended that a key management architecture handles different causes of end of life of a key separately). Actually, recommendations are present is most of international and European regulation such as the recommendations of the UN General Assembly [3], recommendations of Europe Council [4] [5], directives of the European Parliament [6], etc. Similarly, in the field of critical infrastructure, organizations such as the European Council [7], International Governance Risk Council(IGRC) [8], North-American Electric Reliability Council(NERC), etc. specify a number of recommendations to protect infrastructures (e.g., electric network) [9].

These rules of recommendations should be considered as a healthy approach against potential weaknesses, but in no case they are mandatory. Besides that, in addition to theses "classical recommendations", we distinguish another kind of recommendations that - under a condition of time - become an obligation. Basically, this kind of reformulated recommendation rules is not imposed by the security policy in the current time, despite its utility to improve system security, because their application at the time of completion of the document may be binding or costly. These rules could actually be considered as a sort of bridge between the security level selected in the current time and the target one at a given time. As examples of this kind of rules extracted from [2] we can cite:

RuleSymKey-1. The minimum size of symmetric keys used until 2020 is 100 bits.

RuleSymKey-2. The minimum size of symmetric keys to be used beyond 2020 is 128 bits.

RecomSymKey-1. The recommended minimum size of symmetric keys is 128 bits.

It is clear that the obligation to use a symmetric key minimum size of 128 bits in 2020 (RuleSymKey-2) is actually a recommendation in the current time (RecomSymKey-1).

In this work, we define a formal framework to express "classical" recommendation policies and we specify a policy controller in charge of evaluating such policies. Furthermore, we introduce the notion of recommendation becoming an obligation over time. To achieve our goals, we base our analysis on the Temporary Logic of Actions (TLA) [10].

This paper is organized as follows. A brief introduction of TLA is given in the next section. We give in Section 3, an overview of the core language defined in [11] to express access control, usage control and obligation policies, and to specify the policy controller behavior in charge of evaluating such policies. In Section 4, we extend this model to express recommendation rules -in an access control policy. Moreover, we extend the specification of the controller in charge of evaluating such a policy proposed in [11]. In Section 5, we introduce the concept of recommendation that is transformed into obligation and we extend the specification of the controller policy. Section 6 describes the behavior of the controller security policy to reflect the updated security policy. In Section 7 and 8, we describe related works we draw up conclusions .

II. Introduction to TLA

As our formal framework will be based on the Temporal Logic of Actions (TLA), we briefly present it in this section. TLA was introduced by Leslie Lamport in 1991, inspired from [10] and [12]. TLA is our formal framework.

Let *Var* be a countable infinite set of *variables*. The value of each variable will be interpreted as an element of *Val*, set of values.

A *state* is an assignment of values to variables that is, a mapping from the set *Var* of variable names to the collection *Val* of values. Thus, a state *s* assigns a value *s*(*x*) to a variable *x*. The collection of all possible states is denoted *St*. We write *s*□*x*□ to denote *s*(*x*). Hence, we consider □*x*□□of the variable *x* as a mapping from states to values.

Besides, a trace is defined as an infinite sequence of states. We denote $\langle \sigma_0 , \sigma_1 , \sigma_2 ,...\rangle$ sequence of states $\sigma_0 , \sigma_1 ,...$

A *predicate* (or a state predicate) is a boolean expression built from variables and constant symbols. The meaning □*P*□□of a predicate *P* is a mapping from states to booleans, so for every state *s*, *s*□*P*□ is equal to true or false. We say that *s* satisfies *P* if and only if *s*□*P*□ equals true .

An *action* is a boolean valued expression formed from variables, primed variables, and constant symbols. A predicate is actually a special case of action. An action represents a relation between old states and new states, where the unprimed variables refer to the old state and primed variables refer to the new state. Formally, the meaning □*A*□ of an action *A* is a relation between two states, a function that assigns a boolean *s*□*A*□*t* to a pair of states (*s*, *t*). We define *s*□*A*□*t* by considering *s* to be the "old state" and *t* the "new state", so *s*□*A*□*t*□is obtained from *A* by replacing each unprimed variable *v* by *s*□*v*□□and each primed variable *v*' by *t*□*v*□:

s□A□t ≜ A(□ 'v': s□v□ /v, t □v□□/v')

A temporal formula is built from elementary formulas using boolean operator and the unary operator □, read always.

Other new operators can be defined : ◊F ≜ ¬ □¬F

The basic TLA formulas are predicates and formulas of the form: □□A□$_f$, where □A□$_f$ ≜ A □ (f' = f)

Stuttering is a series of states where some variables retain the same value.

For a series of states $\langle s_0 , s_1 , s_2 ,...,s_n \rangle$, for A, B actions, TLA defines the following modalities:

		$\langle s_0 , s_1,..., s_n\rangle$	□A□	≜	s_0 □A□ s_1
Next	◯A	$\langle s_0 , s_1,..., s_n\rangle$	□◯A	≜	s_1□A□s_2
Always	□A	$\langle s_0 , s_1..., s_n \rangle$	□ □□A□	≜	□i □ [0..n] s_i□A□s_{i+1}
Eventually	◊A	$\langle s_0 , s_1..., s_n \rangle$	□◊A□	≜	□i □ [0..n] s_i□A□s_{i+1}

Using the first-order logic, a TLA formula is defined by the following grammar:

Formula $F ::= A \mid \neg F \mid \bigcirc F \mid \Box F \mid \Diamond F \mid F \Box F \mid F \Box F \mid F \rightarrow$

A system, a program or an algorithm are specified by giving all the allowed behaviors of the system. By expressing the specification as a TLA formula, a system is specified by the formula corresponding to allowed behaviors.

III. A Framework to Enforce Access Control, Usage Control and Obligation

An Information System (IS) is an organized set of resources (hardware, software, personnel, data and procedures) allowing to regroup, classify, process and make information accessible in a given environment. Three main elements of the IS are important in our context: the *user*, an active entity that interacts with the IS; the *policy controller* in charge of evaluating the user's requests according to the access control policy (to prevent unauthorized access to a resource of SI); and finally, the *executive manager* of the IS in charge of performing the corresponding actions on data, when it is allowed by the policy controller. Sometimes, the policy controller is called a Policy Decision Point (PDP) and the executive manager is called a Policy Enforcement Point (PEP) or a reference monitor.

In the remainder of this section, we give an overview of the formal framework used in [11] to enforce access control, usage control and obligations. This framework was based on the Temporal Logic of Actions [10] and used to express both contextual permissions and obligations.

Basically, a permission is associated with two conditions, the start condition that must be true just when the access request is evaluated, and the ongoing condition that must be always satisfied while the access is in progress. The concept of cancellation actions was introduced to allow users to cancel access in progress.

An obligation is a mandatory access that must be performed (e.g., by users or by the system). It is associated with two conditions as well: the *raise condition* to trigger the obligation, and the *deadline condition* to determine when the obligation is violated. Furthermore, the concept of non persistent obligation was introduced. An obligation is persistent when the access is mandatory even if its raise condition is no longer true once the obligation was raised and before the deadline expires.

The specification of the framework begun with specifying message types and message interactions between users and the policy controller to the one hand, and message interaction between policy controller and the executive manager on the the other hand:

```
TYPE MessageType   ≜   accessRequest | accessEnd
                       | accessGrant
                       | accessDeny | accessRevoke
                       | cancellationRequest | cancellationDeny
                       | cancellationGrant
                       | obligationNotification
                       | penalty | obligationCancel
```

"AccessRequest" messages represent the access requests sent by users.

"AccessEnd" messages are sent by the executive manager to the policy controller to notify the end of the action corresponding to the access request.

"AccessGrant" messages are sent by policy controller (to the executive manager) when the request is allowed to notify that the relevant action can be executed.

Messages of type "accessDeny" are sent by the policy controller to the user when access is not allowed.

Messages of type "accessRevoke" are sent by the policy controller to notify the executive manager that access should be aborted.

Messages of type "cancellationRequest" are sent by users to cancel ongoing access before end or before a possible revocation.

Message of type "cancellationDeny " are sent by the policy controller to users when their request to cancel an access is denied.

Messages of type "cancellationGrant" are sent by the policy controller to the executive manager to abort the current access.

Message of type "obligationNotification" are sent to users by the policy controller when an obligation is triggered so as the user can perform the access satisfying the obligation.

Messages of type "penalty" are sent by the policy controller to the executive manger when the obligation deadline has expired and the user did not perform the mandatory access at the right time in order to satisfy the obligation.

Every message contains a description of the access. We call Target a triplet composed of: (1) who wants to access, (2) which object is requested and (3) which action is requested.

The target is denoted α and defined as follows:

TYPE Target $\triangleq$ [Subject $\times$ Action $\times$ Object]

Subject, Action and Object are nominal types.

A Message is then formally defined as a couple with a message type and a target as follows:

TYPE Message $\triangleq$ [MessageType $\times$ Target]

To model interactions between users and the executive manager through the network by sending and receiving messages, the network is considered as a finite set of messages that is denoted Φ. Two predicates *snd* and *rcv* are defined like this:

```
PREDICATE  rcv   ≜   [Message → Boolean ]
           snd   ≜   [Message → Boolean ]
```

When a message is sent, Φ will contain the message in the next state, and when the message is received, we are in another state where that message is removed from Φ.

The axiomatic is defined as follows:

AXIOM $\Phi, \Phi' = \Phi \; \square \; \{\langle \text{messageType}, \alpha \rangle\} \vdash \text{snd}\,(\langle \text{messageType}, \alpha \rangle)$

$\Phi \; \square \; \{\langle \text{messageType}, \alpha \rangle\}, \Phi' = \Phi \vdash$

$\text{rcv}\,(\langle \text{messageType}, \alpha \rangle)$

A. Policy Expression

In this subsection, we base our analysis on the formal framework proposed in [11] to express the access control, usage control and obligation policies and how to decide whether a given request is authorised or not by the policy. We first consider the following definitions:

- A *system state* is considered to be a finite set of attributes.
- An *attribute* is a pair of a tag attribute and its corresponding value.
- A *condition* is then a comparison between the expected value of the attribute in the policy and the actual value of the attribute given by the system state when the request is evaluated. To compare two attribute values, a finite set of binary operator is considered.
-
- An *atomic condition* is a comparison between an attribute and its expected value or a comparison between two attributes. A condition can also be a conjunction of atomic conditions.
- The *environment*, denoted Σ is defined as the finite set of attributes describing the state of the system and the finite set of binary operators.

More formally, Attributes, Operators and Conditions are defined as follows:

TYPE	Attribute	≜	[Tag × Value]
	Operator	≜	[Value × Value → Boolean]
	Condition	≜	[Operator × Tag × Value]
			| [Operator × Tag × Tag]
			| [Condition □ Condition]

The predicate *isSatisfied* checks if a given condition is satisfied at the time of the request evaluation. For instance, □ α: Target, □ c: Condition, Σ ⊢ isSatsfied (α,c) means that, according to the environment Σ , the condition c corresponding to the access α is satisfied.

PREDICATE *isSatisfied* [Target × Condition → Boolean]

With atomic condition, the predicate *isSatisfied* is true if the comparison operator is true for a given attribute *tag*(*s*) matching existing *attribute*(*s*) in the system environment. In the case where two tags are presented, the predicate is true if the two tags correspond to existing tags in the system environment Σ and the comparison between their two values is true. With the conjunction of atomic conditions, the predicate is true if each condition is satisfied.

The axiomatic is defined as follows:

AXIOM $\Sigma \; \square \; \{\langle tag_1 , value_1\rangle\} \vdash \text{isSatisfied}\,(\alpha, \langle \; operator, tag_1 , value_2\rangle)$

$\leftrightarrow \Sigma \vdash operator(value_1, value_2)$

$\Sigma \; \square \; \{\langle tag_1 , value_1\rangle, \langle tag_2 , value_2\rangle\} \vdash \quad \text{isSatisfied}\,(\alpha, \langle operator, tag_1, tag_2\rangle)$

$\leftrightarrow \Sigma \vdash operator(value_1, value_2)$

$\Sigma \vdash \text{isSatisfied}(\alpha, c_1 \; \square \; c_2)$
$\leftrightarrow \vdash \quad \text{isSatisfied}(\alpha, c_1) \; \square \; \Sigma \vdash \text{isSatisfied}(\alpha, c_2)$

The access control policy is considered as a finite set of rules, denoted Γ. Each policy rule consists of five parameters:

- a *label* of type *RuleType* indicating whether the rule is a permission or obligation,
- a *target* of type Target,
- *tow conditions* of type Condition, and finally
- a *boolean* of type Option.

The meaning of the parameters differs depending on whether the rule is a permission or an obligation. Basically, in case of a permission rule:

- the first condition must be verified at the time of the request evaluation before access is granted.
- The second condition must be always true , until the end of the action execution corresponding to access request.
- The parameter of type Option is a boolean indicating if an access in progress could be aborted or not.

Besides that, in an obligation rule:

- the first condition is that which triggers the obligation,
- the second condition corresponds to deadline accorded to the user to perform the access satisfying obligation.
- The boolean of type option specifies whether the obligation is persistent or not.

TYPE	Option	≜	Boolean
	RuleType	≜	“permission” | “obligation”
	Rule	≜	[RuleType × Target × Condition × Condition × Option]
	Policy	≜	{Rule}

The predicate *isPermited* checks if there is a rule of type “permission” in access control policy corresponding to the given request. □ *α*: Target, □ *sc*, *oc*: Condition, Γ ⊢ isPermitted (*α*, *sc*, *oc*) means that following the security policy Γ, the access *α* may be granted if the condition *sc* is satisfied and while *oc* is true.

Predicate *isPermited* ≜ [Target ×Condition ×Condition →Boolean]

The *isPermitted* corresponding axiomatic is defined as follows:

```
Axiom   Γ □ {⟨"permission", α, startCondition, ongoingCondition,
            Option⟩}⊢ isPermitted(α, startCondition, ongoingCondition)
```

The predicate *isCancelable* is true if access could be aborted according to the policy:

```
PREDICATE   isCancelable ≜            {[Target → Boolean]}
```

The corresponding axiomatic:

```
AXIOM    Γ □ {⟨"permission", α, sc, oc, TRUE⟩} ⊢ isCancelable(α)
```

Similarly, the predicate *isObligated* is true if there is a rule of type "obligation" in access control policy which corresponds to the given target and conditions :

```
PREDICATE  isObligated ≜         [Target ×Condition ×Condition
           →Boolean]
```

The corresponding axiomatic is as follows:

```
AXIOM Γ □ {⟨"obligation", α, raiseCondition, deadlineCondition, Option ⟩}
          ⊢ isObligated (α, raiseCondition, deadlineCondition)
```

An obligation is persistent if the corresponding option is true. The predicate used to verify whether an obligation is persistent or not is *isPersistent*; it defined by:

```
PREDICATE  isPersistent  ≜  [Target → Boolean]
```

The corresponding axiomatic is as follows:

```
AXIOM Γ □{⟨"obligation", α, raiseCondition, deadlineCondition, True ⟩}
          ⊢ isPersistent(α)
```

B. Policy Interpretation

Based on the TLA formalism, different actions specifying the behavior of the policy controller in charge of evaluating access requests are defined. The access request evaluation depends on the policy and the environment.

Basically, the TLA action "*Request*" is triggered when a message of type "*accessRequest*" is received. The policy controller first checks that there is a corresponding rule (in the policy Γ) matching the message target and conditions using the *isPermitted* predicate. If such a rule matches, the corresponding "*startCondition*" condition is then evaluated according to the environment Σ using the "*isSatisfied*" predicate. If both of these predicates are true, the access is granted and an "*accessGrant*" message is finally sent to the executive manager.

Note that in order to enable the monitoring of authorized access in progress and to ensure usage control, each ongoing access *α* is saved with its condition "*ongoingCondition*" in a "*ongoingAccess*" finite set. The *ongoingAcess* variable is defined as follows:

```
VARIABLE   ongoingAccess  ≜  {[Target × Condition]}
```

An obligation is triggered when the "*raiseCondition*" condition is satisfied according to environment Σ . The policy controller must check all the time if the "*deadlineCondition*" condition corresponding to the ongoing obligation is satisfied. In the case of not persistent obligation , the policy controller must also check if the condition that triggered the obligation is still satisfied. For this, the "*ongoingObligation*" variable is introduced; it is a finite set containing obligations target and their corresponding conditions: "*deadlineCondition*" and "*raiseCondition*".

The variable *ongoingObligation* is defined as follows:

```
VARIABLE   ongoingObligation  ≜  {[Target × Condition × Condition]}
```

When an obligation is triggered, the user could send the access request to satisfy the ongoing obligation. In this case, there is no need to check if the access is permitted because it is mandatory. The access is thus allowed and the obligation is removed from the set *ongoingObligation*.

The Request action is defined as follows:

```
Action  Request  ≜
        □ α : Target |
        Φ, Φ'  ⊢  rcv (⟨"requestAccess", α ⟩)
        □ If  □ raiseCondition, deadlineCondition : Condition |
             ⟨α, raiseCondition, deadlineCondition⟩     □ ongoingObligation
          Then   ⟨α,raiseCondition,ongoingObligation⟩              ∉
          ongoingObligation'
               □ Φ, Φ' ⊢     snd ( ⟨"grantAccess", α ⟩)
          Else If  □ startCondition, ongoingCondition : Condition |
                   Γ ⊢  isPermitted (α, startCondition, ongoingCondition)
                      □ Σ  ⊢ isSatisfied (α, startCondition)
          Then  ⟨ α, ongoingCondition ⟩ □ ongoingAccess'
                 □ Φ, Φ' snd ( ⟨"grantAccess", α⟩)
          Else  Φ, Φ'  ⊢ snd( ⟨"denyAccess", α ⟩)
```

The policy controller must ensure that all *ongoingCondition* conditions associated with the current access are satisfied. When one of them is no longer satisfied, access is revoked and a "*accessRevoke*" message is sent to the executive manager; Subsequently, the corresponding access is removed from the *ongoingAccess* set.

The action *CheckOngoingAccess* is thus introduced and is defined as follows:

```
Action   CheckOngoingAccess  ≜

          □ α : Target |
□ ongoingCondition : Condition |
⟨α, ongoingCondition⟩    □ ongoingAccess
□ If Σ ¬ isSatisfied (α, ongoingCondition)
   Then ⟨α, ongoingCondition⟩    ∉ ongoingAccess'
        □ Φ, Φ' ⊢ snd( ⟨"accessRevoke", α 𝓘)
```

When a cancellation request is received, the policy controller obviously checks if the current access is cancellable according to the security policy; if it is the case, the policy controller removes the ongoing access from the *ongoingAccess* set.

The action Cancel is thus defined as follows:

```
Action   Cancel ≜
         □ α : Target |
         Φ, Φ' ⊢ rcv (⟨"cancellationRequest", α ⟩)
         □  If  □ ongoingCondition : Condition |
                ⟨α, ongoingCondition⟩    □ ongoingAccess
                □ Γ ⊢  isCancelable (α)
            Then  Φ, Φ' ⊢ snd( ⟨"cancellationGrant", α ⟩)
                  □    ⟨α, ongoingCondition⟩    ∉ ongoingAccess'
            Else Φ, Φ' ⊢ snd( ⟨"cancellationDeny", α ⟩    )
```

Note that the *raiseObligation* action, defined below, allows the policy controller to check if there are obligations that have been triggered, i.e., when their associated *raiseCondition* conditions are satisfied according to the system environment Σ. If an obligation is triggered, the policy controller sends the "*obligationNotification*" notification message to the user to perform the appropriate access requirement, and the corresponding target and *raiseCondition* and *deadlineCondition* conditions are registered in the *ongoingObligation* set.

```
Action   RaiseObligation  ≜
         □ α : Target,
         □ raiseCondition, deadlineCondition : Condition |
         Γ ⊢  isObligated (α, raiseCondition, deadlineCondition)
         □ Σ ⊢  isSatisfied (α, raiseCondition)
         □ ⟨α, raiseCondition, deadlineCondition⟩     ∉ ongoingObligation
         □ ⟨α, raiseCondition, deadlineCondition ⟩    □ ongoingObligation'
         □ Φ, Φ' ⊢ snd ( ⟨"notifyObligation", α ⟩    )
```

It is worth noting that the policy controller must repeatedly check if each *deadlineCondition* condition associated with a persistent ongoing obligation is satisfied; when one of them is satisfied, the "penalty" message is sent to the executive manager and the corresponding obligation is removed from the *ongoingObligation* set. In the case of non persistent obligation, the policy controller must check if *raiseCondition~~i~~* condition is still satisfied; if not, a "*cancelObligation*" message is sent and this obligation is removed from the *ongoingObligation* set. The *CheckOngoingObligation* action is thus introduced and is defined as follows:

```
Action   CheckOngoingObligation ≜
         □ α : Target,
         □ raiseCondition, deadlineCondition : Condition |
         ⟨α, raiseCondition, deadlineCondition ⟩          □
         OngoingObligation
         □ If  Σ ⊢ ¬ isSatisfied (α, raiseCondition)
              □ Σ ⊢ ¬ isPersistant(α, raiseCondition)
           Then     ⟨α, raiseCondition,deadlineCondition⟩
           ∉ongoingObligation'
                 □ Φ, Φ' ⊢ snd (⟨"CancelObligation", α ⟩    )
           Else If  Σ ⊢  isSatisfied (α, deadlineCondition)
           Then     ⟨α, raiseCondition,deadlineCondition⟩
           ∉ongoingObligation'
                 □ Φ, Φ' ⊢ snd (⟨"penalty", α ⟩    )
```

When the executive manager completes an action corresponding to an access request, an "*accessEnd* " message is sent to the policy controller; consequently, the corresponding target and *ongoingCondition* condition must be removed from the *ongoingAccess* set. The End action is defined as follows:

```
Action   End ≜
         □ α: Target |
         Φ, Φ' ⊢ rcv (⟨"accessEnd", α ⟩    )
         □ If  □ ongoingCondition: Condition |
               ⟨α, ongoingCondition ⟩            □ OngoingAccess
           Then    ⟨α,ongoingCondition ⟩    ∉☐ongoingAccess'
```

Finally the *Init* action and the policy controller behavior are defined as follows :

```
Action   Init ≜
         ongoingAccess = □
         □ ongoingObligation = □

specification  PolicyController ≜
               Init
               □ ☐ [Request □ End □ CheckOngoingAccess □ Cancel
                   □ RaiseObligation
                   □ CheckOngoingObligation]<ongoingAccess,ongoingObligation>
```

IV. Recommendations

The framework proposed in the previous section can be used to specify security requirements of many applications such as Digital Right Management, P2P or Web Service applications. However it is not rich enough to cover security requirements in form of recommendations while this access modality is constantly present in several security policies (as explained in the introduction). In fact, we often find guidance in the form of recommendations as in the Recommendations on collective cross-border management of copyright and related rights for legitimate online music services published by the Commission of the European Communities [13]. This document specifies several recommendations like:

– "*Collective rights managers should give reasonable notice to each other and commercial users of changes in the repertoire they represent".*
– "*Upon payment of the royalties, collective rights managers should specify all the right-holders they represent, the deductions made for purposes other than for the management services provided".*

We can cite many other example, but due to space limitation we can clearly state that recommendations are present in many current and emergent applications. Consequently, we propose extending the proposed framework to express recommendations rules in access control policy: moreover, we extend the specification of the controller in charge of evaluating such policy.

Basically, we consider recommendations as modalities that could advise the subjects (e.g., users) to do certain actions. When a recommendation is triggered, the policy controller notifies the user to perform the access in order to satisfy the recommendation. Note that by contrast to obligations, if the user does not perform the access, no penalty is applied on him. Moreover, after a given deadline, if the user does not perform the access satisfying the recommendation, the policy controller generally sends a reminder of the recommendation to the user.

To illustrate theses notions, let us consider another example: assume that for authentication to access the IS, certificates are used with the RSA algorithm. The access control policy may specify some recommendations on the size of RSA modules and size of RSA public exponents. Actually, the general security document [2] issued by the National Agency for the security of information systems in France already introduce this kind of recommendations by stipulating that: *"It is recommended for any application, to use public exponents strictly greater than 2^{16} = 65536". We can thus perfectly imagine that i*f the user's certificate contains an RSA key with exponent is less than 2^{16} (condition that triggers the recommendation), a notification is sent to the user to change its key. The notification is re-sent if the user does not change its key on a fixed time interval.

We thus extend the former formal framework by adding a new message interaction called: "*recommendationNotification*".

TYPE *MessageType* ≜ *"accessRequest"* | *"accessEnd"*
| *"accessGrant"*
| *"accessDeny"* | *"accessRevoke"*
| *"cancellationRequest"*
| *"cancellationDeny"*
| *"cancellationGrant"*
| *"obligationNotification"*
| *"penalty* | *obligationCancel"*
| *"recommendationNotification"*

A. *Specification of the security policy*

We believe that a recommendation rule is associated with two conditions. The first one is called *"raiseCondition*". The recommendation is triggered when this condition is satisfied. The second condition is called "*deadlineCondition*". When the recommendation is triggered, if the subject (e.g., user) does not perform the action satisfying the recommendation and if the condition "*deadlineCondition*" is satisfied, a reminder of the recommendation is sent to the user. The type of our security policy rules is redefined as follows:

TYPE *Permission* ≜ *"permission"*
Obligation ≜ *"obligation"*

Recommendation ≜ *"recommendation"*

TYPE Rule ≜ [*Permission* × *Target* × *Condition* × *Condition* × *option*] |
[*Obligation* × *Target* × *Condition* × *Condition* × *option*] |
[*Recommendation* × *Target* × *Condition* × *Condition*]

The predicate *isRecommended* and the corresponding axiomatic are defined as follows:

PREDICATE *isRecommended* ≜ [Target × Condition → Boolean]

AXIOM Γ □ {⟨"recommendation ", α, raiseCondition ⟩}

⊢ isRecommended (α, raiseCondition, deadlineCondition)

B. *Policy Interpretation*

A recommendation is triggered when the condition "*raiseCondition*" is satisfied for a given target. The policy controller must check if the condition "*deadlineCondition*" associated with the current recommendation is satisfied. We consequently introduce the "*ongoingRecommendation*" variable, a finite set containing the target of the current recommendation with the corresponding "*deadlineCondition*" condition. Hence, the *ongoingRecommendation* is a finite set of ongoing recommendations with their associated recall conditions waiting to be satisfied.

VARIABLE *ongoingRecommendation* ≜ {[Target × Condition]}

The action that triggers the recommendation is defined below:

```
ACTION  RaiseRecommendation ≜
        □ α: Target,
        □ raiseCondition, deadlineCondition: Condition
        Γ ⊢ isRecommanded (α, raiseCondition, deadlineCondition)

        □ ⟨α, deadlineCondition⟩   ∉ ongoingRecommendation

        □ ⟨α, deadlineCondition⟩ □ ongoingRecommendation'

        □ Φ, Φ' ⊢ snd (⟨"recommendationNotification", α ⟩)
```

The *checkOngoingRecommendation* action checks if for an ongoing recommendation, the *deadlineCondition* condition is satisfied; if it is the case, the recommendation is removed from the *ongoingRecommendation* set. If for the target α, the condition which triggered the recommendation is always true, the *raiseRecommandation* action will send a reminder to the user to perform the query that satisfy the recommendation.

```
ACTION  CheckOngoingRecommendation ≜
        □ α: Target,

        □ deadlineCondition: Condition |

        ⟨α, deadlineCondition⟩ □ ongoingRecommendation

        □ if Σ ⊢ isSatisfied (⟨α, deadlineCondition⟩)

            Then ⟨α, deadlineCondition⟩   ∉ ongoingRecommendation'
```

The action Request is redefined as follows:

```
ACTION  Request ≜
        □ α : Target |
        Φ, Φ' ⊢ rcv (⟨"accessRequest", α ⟩)

        □ If □ raiseCondition, deadlineCondition : Condition |
              ⟨α, raiseCondition, deadlineCondition⟩   □

              ongoingObligation
            Then ⟨α,raiseCondition, deadlineCondition⟩   ∉

                  ongoingObligation'
                  □ Φ, Φ' ⊢    snd ( ⟨"accessGrant", α ⟩)
            Else If  □ deadlineCondition: Condition |

                  ⟨α, deadlineCondition⟩   □ ongoingRecommendation

            Then ⟨α, deadlineCondition⟩∉ ongoingRecommendation'

                  □ Φ, Φ' ⊢    snd ( ⟨"accessGrant", α ⟩)

            Else If  □ startCondition, ongoingCondition: Condition |

                  Γ ⊢ isPermitted (α, startCondition, ongoingCondition)

                  □ Σ ⊢ isSatisfied (α, startCondition)

            Then ⟨ α, ongoingCondition ⟩ □ ongoingAccess'

                  □ Φ, Φ' snd ( ⟨"accessGrant", α⟩)

            Else  Φ, Φ' ⊢ snd( ⟨"accessDeny", α ⟩)
```

Finally, the *Init* action as well as the behavior of the policy controller are re-defined as follows:

```
Action  Init ≜
        ongoingAccess = □
        □ ongoingObligation = □
        □ ongoingRecommandation = □

specification  PolicyController ≜
               Init
               □ □ [Request □ End □ CheckOngoingAccess □ Cancel
                    □ RaiseRecommandation
                    □  CheckOngoingRecommendation
                    □ CheckOngoingObligation
                    □
                    RaiseObligation]<ongoingAccess,ongoingObligation,ongoingRecommendation>
```

V. From Recommendation to Obligation

In numerous contexts, some recommendation rules are mandatory rules. These recommendations could not be obligations upon the completion of the document, because their applications could not be practical for the user, or expensive at the time (even if their application could be strongly recommended). Several concrete and real example may confirm this vision and this form of recommendations. For instance, the Policy Certification Type document "Authentication Server" issued by the National Security Information Systems[14] stipulates that: "*Requirements, common to all levels and specific to a given level, specified in this certification policy type must be fully respected by the providers of electronic certificates except for the following: in this certification policy type, a number of recommendations are formulated. Providers of electronic certificates are encouraged to also respect them now because these recommendations which are not mandatory in this version of this document will become it later*". In this example, a recommendation clearly becomes a requirement when a transition condition is satisfied.

For example: let us take the following three rules of the RSA module:

Rule-1:The minimum size of the module must be 2048 bits, for use not to exceed the year 2020.

Rule-2: For use beyond 2020, the minimum size of the module must be 4096 bits

Rule-3: It is recommended to use a 4096-bit key

In other words, if the user RSA key has with a module length less than 2048 bits (condition that triggers the obligation), the user is forced to change its key, assuming in a one month deadline. If by contrast the RSA key module is 2048 bit (condition that triggers the recommendation), the user is notified to change the key; and this notification is resent every month until he satisfies the recommendation. In 2020 (the condition transforming recommendation to obligation), the recall recommendation is stopped and if the user has not yet performed the access satisfying the recommendation, he will be penalised, as recommendation is becoming an obligation in this case. The second rule could then be deleted and the third rule could be a recommendation which becomes an obligation in 2020.

A. *Specification of the security policy*

In order to specify a recommendation that becomes an obligation, we suggest using the "*transitCondition*" condition. We assume that the time to recall a recommendation is the same as the one allowed to perform an obligation.

The policy rules thus are re-defined as follows:

TYPE *Permission* ≜ *"permission"*
Obligation ≜ *"obligation"*
Recommendation ≜ *"recommendation"*

TYPE Rule ≜ [*Permission* × *Target* × *Condition* × *Condition* × *option*] |
[*Obligation* × *Target* × *Condition* × *Condition* × *option*] |
[*Recommendation* × *Target* × *Condition* × *Condition* × *Condition* × *option*]

The *isRecommanded* predicate is also re-defined as follows:

PREDICATE *isRecommended* ≜ [Target ×Condition × Condition × Condition → Boolean]

Consequently, the corresponding axiomatic is re-defined as follows:

AXIOM Γ □ {⟨"recommendation ", α, raiseCondition, deadlineCondition, transitCondition, option⟩}
⊢ isRecommended (α, raiseCondition, deadlineCondition transitCondition, option)

The *isPersistent* axiomatic is also re-defined as follows:

AXIOM Γ □ {⟨"obligation", α, raiseCondition, deadlineCondition, True⟩}
□ {⟨"recommendation", α, raiseCondition, deadlineCondition, TransitCondition, True⟩}
⊢ isPersistent(α)

B. *Policy Interpretation*

With the notion of recommendation that becomes obligation, a recommendation is triggered if the "*raiseCondition*" condition is satisfied and the "*transitCondition*" condition is not satisfied. The *raiseRecommandation* action is subsequently redefined as follows:

ACTION *RaiseRecommendation* ≜
□ α : Target,
□ raiseCondition, recallCondition, transitCondition: Condition
Γ ⊢ *isRecommanded* (*α, raiseCondition, recallCondition, transitCondition*)
□ Σ ⊢ isSatisfied (α, raiseCondition)
□ Σ ⊢ ¬ isSatisfied (α, transitCondition)
□ ⟨α, *deadlineCondition*⟩ ∉ *ongoingRecommendation*
□ ⟨α, *deadlineCondition*⟩ □ *ongoingRecommendation'*
□ Φ, Φ' ⊢ snd (⟨"*recommendationNotification*", α ⟩)

Note that the *RecommendationToObligation* action is used to checks: if for a recommendation rule, the *transitCondition* condition is satisfied; if yes, the target and the corresponding *raiseCondition* and *deadlineCondition* conditions are then recorded in *ongoingObligation* set:

ACTION *RecommendationToObligation* ≜
□ α : Target,
□ raiseCondition, deadlineCondition, transitCondition:Condition|
Γ ⊢ isRecommanded (α, raiseCondition, deadlineCondition, transitCondition)
□ Σ ⊢ isSatisfied (α, raiseCondition □ transitCondition)
□ *if* ⟨α, *deadlineCondition*⟩ □ *ongoingRecommendation*
Then ⟨α,*deadlineCondition*⟩∉*ongoingRecommendation'*
□ ⟨α, raiseCondition □ transitCondition, deadlineCondition⟩ ∉ ongoingObligation
□ ⟨α, raiseCondition □transitCondition, deadlineCondition ⟩ □ ongoingObligation'
□ Φ, Φ' ⊢ snd (⟨"obligation*Notification*", α ⟩)

The behavior of policy controller is consequently redefined as follows:

Action Init ≜
ongoingAccess = □
□ ongoingObligation = □
□ ongoingRecommandation = □

specification PolicyController ≜
Init
□ ☐ [Request □ End □ CheckOngoingAccess □ Cancel
□ RaiseRecommendation
□ RecommendationToObligation
□ CheckOngoingRecommendation
□ CheckOngoingObligation
□ RaiseObligation]$_{<ongoingAccess,ongoingObligation,ongoingRecommendation>}$

VI. Policy Update

As a security policy is usually dynamic (not static), the policy controller behavior should consider and handle the frequent updates of the security policy. In this section, we extend our formal framework mechanism to handle the policy updates, so that the access control policy can be updated not only when ongoing obligations are checked, but also when ongoing recommendation are checked. As in [11], we consider every update of the security policy as regular access request; for example "*admin*", "*update*", "The policy" authorized by the policy controller according to the security policy. The update is effective when this access is complete, i.e., when “*accessEnd*”, “*admin*”, “*update*”, “*The policy*” is received by the policy controller. Note that the *CheckUpdate* action (defined bellow) is responsible for checking if there is an update of the security policy; if yes, the *UdpdateOngoingAccess*, *UpdateOngoingObligation* and *UpdateOngoingRecommendation* actions are then updating the ongoing access, ongoing obligations and ongoing recommendations sets respectively.

```
ACTION  CheckUpdate ≜
        □ admin: Subject, □ α: Target |
        α = ⟨admin, “update ”, “ThePolicy”⟩

        □  Φ, Φ'  ⊢ rcv(⟨“endAccess”, α ⟩)

        □ UpdateOngoingAccess
        □ UpdateOngoingObligation
        □ UpdateOngoingRecommendation
```

Actually, the *UpdateOngoingAccess* action allows the update of *OngoingAccess* set containing the target and the corresponding condition of the access in progress. For each target and the corresponding condition in the *OngoingAccess* set, the action checks if there is a corresponding permission rule in the new policy:

- if the rule exists and the condition recorded in *OngoingAccess* set is different from the condition in the new security policy, the condition recorded in *OngoingAccess* set is replaced by the new condition in the new security policy.
- If there is no rule in the new security policy corresponding to the target and related condition, i.e., the ongoing access is no longer allowed in the new security policy, the ongoing access is revoked.

Note that the previously defined *CheckOngoingAccess* action checks if new conditions in *ongoingAccess* are satisfied.

```
ACTION  UpdateOngoingAccess ≜
        □ α: Target,
        □ ongoingCondition 1 : Condition |
        ⟨α, ongoingCondition 1 ⟩ □ ongoingAccess

        □ If □ startCondition 2 , ongoingCondition 2 : Condition |

              Γ ⊢   isPermitted(α, startCondition 2 , ongoingCondition
              2)
              □ ongoingCondition 1 ≠ ongoingCondition 2
          Then ⟨α, ongoingCondition 1⟩ ∉ ongoingAccess'

                □ ⟨α, ongoingCondition 2 ⟩  □ ongoingAccess'

          Else ⟨α, ongoingCondition 1 ⟩ ∉ ongoingAccess'

                □ Φ, Φ' snd( “revokeAccess”, α )
```

Besides that, for the update of the ongoing obligations, the *UpdateOngoingObligation* action goes through all the target, and related conditions. If obligation rules corresponding to the target still exist in the new security policy, the action updates conditions in *ongoingObligation* set by conditions in the new security policy if they are different. If there is no corresponding obligation rule in the new security policy, we check if there is a corresponding recommendation rule. If that is the case, it is naturally a recommendation that becomes an obligation; consequently, the action updates the condition that triggered the obligation (conjunction of the condition which triggers the recommendation and the condition that transforms the recommendation to an obligation) by the new value (conjunction of the new condition which triggers the recommendation and the new condition that transforms the recommendation to an obligation in the new security policy) and the recall condition by the new one in the new security policy if it is different. If there is no corresponding recommendation rule, then the obligation is aborted.

Note that new conditions are checked by the *CheckOngoingObligation* action in the case of obligations as well as in the case of recommendations turned into obligations.

```
ACTION  UpdateOngoingObligation ≜
        □ α: Target,
        □ deadlineCondition 1 : Condition |
        ⟨ α, raiseCondition 1 , deadlineCondition 1 ⟩ □ ongoingObligation

        □  If □ raiseCondition 2 , deadlineCondition 2 : Condition |
              Γ ⊢ isObligated(α, raiseCondition 2 ,
                                  deadlineCondition 2 )
              □ (raiseCondition 1 ≠ raiseCondition 2
                □ deadlineCondition 1 ≠ deadlineCondition 2 )

           Then ⟨ α, raiseCondition 1 , deadlineCondition 1 ⟩ ∉
                ongoingObligation'
                □ ⟨ α, raiseCondition 2 , deadlineCondition 2 ⟩ □
                  ongoingObligation'
           Else If □ raiseCondition 2 , deadlineCondition 2 ,
                  transitCondition: Condition |
                Γ ⊢  isRecommended(α, raiseCondition 2 ,
                                   deadlineCondition 2 ,
                                   transitCondition)
                □ (raiseCondition 1 ≠ (raiseCondition 2 □
                                   transitCondition)
                  □ deadlineCondition 1 ≠ deadlineCondition 2 )
           Then ⟨ α, raiseCondition 1 , deadlineCondition 1⟩ ∉
                ongoingObligation'
                □ ⟨ α, raiseCondition 2 □ transitCondition,
                  deadlineCondition 2 ⟩ □ ongoingObligation

           Else ⟨ α, raiseCondition 1 , deadlineCondition 1⟩ ∉
                ongoingObligation'
                □ Φ, Φ' ⊢ snd(⟨ “cancelObligation”, α ⟩)
```

The update of the recommendations is done by the *UpdateOngoingRecommendation* action that goes through all elements of the o*ngoingRecommendation* set. If there is a recommendation rule in the new access control policy corresponding to the current recommendation, then we update the deadline condition corresponding to the ongoing recommendation by the new deadline condition recorded in the new policy if they are different. If no corresponding recommendation rule in the new security policy, the recommendation is aborted.

```
ACTION   UpdateOngoingRecommendation ≜
         □ α: Target,
         □ recallCondition ₁ : Condition |
         ⟨ α, recallCondition ₁ ⟩ □ ongoingRecommendation
         □  If □ raiseCondition ₂ , recallCondition, transitCondition:
                    Condition |
               Γ ⊢ isRecommended (α, raiseCondition ₂ ,
                                      recallCondition, transitCondition)
               □ recallCondition ₁ ≠ recallCondition ₂ )
             Then ⟨ α, recallCondition ₁ ⟩ ∉ ongoingRecommendation'
                  □ ⟨ α, recallCondition ₂ ⟩ □ ongoingRecommendation'
             Else ⟨ α, recallCondition ₁ ⟩  ∉ ongoingRecommendation'
```

The final behavior of the controller of the policy is given as fllows:

```
Action   Init ≜
       ongoingAccess = □
       □ ongoingObligation = □
       □ ongoingRecommandation = □

SPECIFICATION PolicyController ≜
              Init
              □ ☐ [Request □ End □ CheckOngoingAccess □ Cancel
                 □ RaiseRecommandation
                 □ RecommendationToObligation
                 □  CheckOngoingRecommendation
                 □ CheckOngoingObligation
                 □ CheckUpdate
                 □RaiseObligation]<ongoingAccess,ongoingObligation,ongoingRecommendation>
```

VII. Related Work

In the literature, most of traditional security models are unfortunately static. They in fact respond to user requests just by yes or no. Recently, some interesting works on access control framework that model obligations was proposed [15] [16] [17] [18]. The formalization of the obligation is different from one model to another. In XACML for example, obligations are all operations that must be filled in conjunction with the application of the authorization decision. In [15] and [19], there is a difference between provisions and obligations: provisions are actions or conditions that must be satisfied before an access decision is made, while obligations are actions that must be satisfied by the users or the system after the access decision is made. The specification language of obligations in [18] distinguishes between usages formulas and obligations formulas: usage formulas concern operations on the data that must be protected, while obligations formulas are conditions on the use of data. An obligation formulas become an obligation once the data received by the user and the later agrees to the conditions. Besides that, while in UCON $_{ABC}$ [20], obligations rules must be satisfied before granting access. In [21], obligations are actions that regular users of the IS must perform . In [11], a permission is associated with two conditions, the first must be true at the time of the request evaluation, and the second must always be true as long as access is in progress. It also introduce a concept to give the user the right to abandon an access in progress. Moreover, the obligation in [11] is associated with two conditions: the first one triggers the obligation. The second condition determines when the obligation should be considered as violated. If the user did not perform the access satisfying the obligation before this condition becomes true, a penalty is applied on the user.

However, none of these models expresses recommendations. Up to our knowledge, the only works introducing recommendations are proposed in [22] and [23], In [22], we based our work on the deontic logic and we suggested a logical framework for modeling recommendations. In this model, a rule is a requirement in the current state of the system if all states connected directly to it satisfy the rule. A rule is permitted if at least one state directly connected to the current state satisfies the rule. A rule is a recommendation if a majority of states connected to the current state satisfy the rule. Besides that, in [23], we introduce the notion of probability of occurrence to model recommendations. Basically, a rule is recommended if the possibility of occurrence of this rule in the possible executions of the system is greater than 0.5; not recommended if its probability of occurrence is less than 0.5; mandatory if its probability of occurrence is 1 and finally prohibited if its probability of occurrence is zero.

In this paper, our model is able to express recommendations that become obligations. In addition, we provide a framework to implement recommendations by specifying the behavior of the policy controller in charge of evaluating access request according to the access security policy. A recommendation in our model is associated with three conditions:

– the first is the one that triggers the recommendation. When this condition is true, a notification is sent to the user to perform the access in order to satisfy the recommendation.
– The second condition is the recall deadline that determines when the next notification will be sent if the user has not performed the access, and finally
– The third condition is the one that determines when a recommendation could become a requirement.

VIII. Conclusion

In many emerging applications, several regulations are in the form of recommendations and guidelines. Of course, these guidelines should be reflected in the security policy, both in the specification and in the implementation phase. Modeling of the recommendations is thus a new challenge in the security policy and models.

In this paper, we establish a formal framework to implement recommendations in general and, in particular, recommendations that become obligations over time. We base our work and greatly extend the specification defined in [11] to express recommendations in access control policy. Using TLA, we extend the specification of the policy controller in charge of evaluating such a policy. This could be used to specify the security policy that is based on requirements information systems for a qualification or certification policy documents. In these documents, there are rules of obligations that must be met in full and a number of recommendations, they suggest, to be respected, indicating that it will become mandatory in subsequent version of this documents.

REFERENCES

[1] Référentiel général de sécurité version 1.0, 6 mai 2010.

[2] Référentiel général de sécurité version 1.0, annex b1, mécanismes cryptographiques, règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques, version 1.20, Janvier 2010.

[3] Resolution a/res/45/ general assembly of united nations, guidelines for the regulation of computerized personal data files, December 1990.

[4] Recommendation of the communication of health information in hospitals, european health committee cdsp (92)8, council of europe, strasbourg, June 1992.

[5] Recommendations of the council of europe, r(97)5, on the protection of medical data banks, council of europe, strasbourg, February 1997.

[6] Directive 95/46/ec of the european parliament and of the council of 24 october 1995, on the protection of individuals with regard to the processing of personal data, October 1995.

[7] European council, bangemann report recommendations to the ec, May 1994.

[8] International risk governance council, critical infrastructures at risk: Securing the european electric power system, 2007.

[9] North american electric reliability council, urgent action standard 1200, 2003.

[10] Leslie Lamport. The temporal logic of actions. In ACM Transactions on Programming Languages and Systems, 16(3):872-923, May 1994.

[11] T. Sans, F Cuppens, and N. Cuppens-Boulahia. A framework to enforce access control, usage control and obligations. Annals of telecommunications : Security in The Digital World, November-December 2007.

[12] Denis Roegel. Etude de la sémantique de programmes parallèles réels en TLA. PhD thesis, Université Henri Poincaré – Nancy 1, 7 novembre 1996.

[13] Commission recommendation on collective cross-border management of copyright and related rights for legitimate online music services, 18 May 2005.

[14] Référentiel général de sécurité version 1.0, annexe a9, politique de certification type, authentification serveur version 2.3, 11 février 2010.

[15] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Obligation monitoring in policy management. In International Workshop, Policies for Distributed Systems and Networks (Policy 2002), Montery, California, USA, June 2002.

[16] N. Demeanor, N. Delay, E. Lupus, and M. Sloan. The ponder policy specification language. In International Workshop Policy, Bristol, UK, 2001.

[17] Q. Ni, E. Bertino, and J. Lobo. An obligation model bridging access control policies and privacy policies. In 13th ACM SACMAT, Estes Park, CO, USA, June 2008.

[18] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In 12th European Symposium On Research In Computer Security (ESORICS), Dresden, Germany, September 2007.

[19] Manuel Hilty, David Basin, and Er Pretschner. On obligations. In In: Proc. ESORICS. (2005) 98–117, pages 98–117, 2005.

[20] J. Park and R. Sandhu. The uconabc usage control model. In ACM Transactions on Information and System Security, 7(1), February 2004.

[21] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: A security model with non atomic actions and deadlines. In In 18th IEEE Computer Security Foundations Workshop (CSFW), Aix en Provence, France, June 2005

[22] A. Abou El Kalam and P.Balbiani. A policy language for modelling recommendations. IFIP Advances in Information and Communication Technology, 297(ISBN 978-3-642-01243-3):176, 2009.

[23] A. Abou El Kalam. A research challenge in modeling access control policies: Modeling recommendations. In RCIS, pages 263–270, 2008.