ABSTRACT
Over the last few years there has been a rapid development of technologies such as ubiquitous computing and distributed multi-agent systems. As a consequence an increasing need to share information securely in a distributed dynamic environment has arisen. Risk-aware access control (RAAC) has recently shown promise as an approach to addressing this need of flexible and dynamical access control requirements. Additionally, OASIS proposed XACML as a new standard XML-based language for writing access control policies, requests and responses. The standard specification also defines reference architecture for implementing an XACML based system. Despite the fact that XACML is designed to support various access control models, we believe it doesn’t provide a natural way for defining RAAC policies. In this paper we propose an approach that uses standard XACML features to implement RAAC. In particular, we abstract core components of RAAC relevant to risk assessment and risk mitigation, and illustrate how to define XACML policies to implement these components. We also propose a modular architecture for the XACML obligations service to handle both system and user obligations, which are typically used as risk mitigation methods in RAAC.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Access controls

Keywords
XACML, RAAC, access control

1. INTRODUCTION
Recently, the scientific community has seen a growing interest in technologies such as ubiquitous computing and multi-agent systems. These technologies are characterized by an intensive exchange of information between different subjects that possibly act in distributed and dynamic environments. Traditional access control models have been shown to be not flexible enough to ensure a secure but efficient sharing of information and resources in these kind of situations in which different agents could act in different domains and be subject to different access control policies. Moreover traditional access control systems, based on static policies, are too rigid to handle the exceptional or emergence situation in which such policies should be relaxed to some extent in order to proceed with business functionality.

Inspired by addressing this increasing need to share information in dynamic environments, risk-aware access control (RAAC) has been the subject of considerable research in recent years [2, 4, 5, 6, 7, 10]. The core idea of RAAC is to develop an authorisation decision function that is able to make decisions based on dynamic risk analysis: how much risk is incurred by allowing access (which runs the risk of information disclosure or modification) or denying access (which runs the risk of preventing business objectives from being realized). From the safety perspective, the system usually employs some risk mitigation methods to account for and reduce the risk that is incurred by granting access. Typically, there are two types of mitigation methods that have been proposed [5]: either the system automatically enforces a risk mitigation action (e.g. audit all granted or denied access) or the system requires the requesting user to perform a risk mitigation action (e.g. explain the purpose of the access request in writing within 12 hours). We refer to this two types of actions as system obligations and user obligations respectively.

Meanwhile, OASIS has introduced a standard XML-based language, namely eXtensible Access Control Markup Language (XACML) [13], for the specification and evaluation of access control policies. It is designed to provide a standard policy exchange format, and to support fine-grained authorisation policies that are implementation independent. It supports the implementation of distributed access control systems, in which decisions made by different policy decision points can be combined together to obtain a final decision regarding a request to access distributed resources.

Over the last decade, XACML has attracted considerable interest from industry and the research community. Despite the enthusiasm for RAAC and XACML, the use of XACML to implement RAAC has been less widely studied. In this paper we aim to develop a simple and widely applicable approach to build RAAC using XACML. Our main contributions can be summarized as follows:

- On the basis of the RAAC models developed by Chen et al. [5], we summarize a basic model that abstracts common system components relevant to risk assessment and mitigation. These components can be naturally integrated into existing access control models, making them risk-aware.
• We propose an approach that uses XACML policies and features to implement elements of the basic model. We introduce our approach on the top of the RBAC profile of XACML [12], which means that, for those who implement RBAC using this profile, it would be straightforward to adapt our approach to incorporate risk into their RBAC solution.
• We extend the basic model to introduce the concept of budget, which is used to account for failure of users to discharge their obligations. In comparison with the work of Chen et al. [5], we take a more restrictive approach that requires users to pay budget for risky access, which only be returned when the associated obligations are fulfilled by the user.
• We examine the use of XACML to implement the extended RAAC model. In particular, in order to fully support the operational semantics of the extended model in XACML, we propose a concrete form of the obligations service for understanding and handling both system and user obligations.

2. BACKGROUND

In this section, we present an abstract model for risk-aware access control, based on earlier work by Chen et al. [5], and then introduce a running example to illustrate the model. We also provide a brief overview of XACML.

2.1 An Abstract Model for RAAC

Let \( P \) be a set of permissions. A permission represents an action-object pair for which a subject may be authorised. Let \( S \) be a set of subjects. A subject represents an active entity in a system that may request access to resources and it could consist in a human user or a software agent. Let \( C \) be a set of contexts. We model an access request as a tuple \( \langle s, p, c \rangle \), where \( s \in S \), \( p \in P \) and \( c \in C \). In general, the risk of granting a permission to a subject in a particular context can be interpreted as the likelihood of the misuse of the permission by the subject. Determining the likelihood of misuse depends on various factors such as the security attributes of subject (e.g. trustworthiness, roles or access history), the value of the resource, the context (e.g. device, location or current time) from which the subject is requesting, etc. Let \( \Sigma \) denote a set of states, and \( \mathcal{K} = \{ k \in \mathbb{R} : 0 \leq k \leq 1 \} \) denote a risk domain. We define a risk function \( \text{risk}: Q \times \Sigma \rightarrow \mathcal{K} \) that takes as input an access request \( q = \langle s, p, c \rangle \in Q \) and the current system state \( \sigma \in \Sigma \), and returns the risk \( k \in \mathcal{K} \) associated with the request. There are a number of ways of explicitly defining the risk function depending on system requirements and a concrete access control model. These are domain-dependent, and thus outside the scope of this paper.

From the system’s perspective, we need to determine a risk threshold that the system is willing to accept when granting access requests, and what kind of risk mitigation should be put in place if risky access is allowed. We define risk thresholds and risk mitigation strategies on a per-permission basis. We write \( [k_0, O^*_0], [k_1, O^*_1], \ldots, [k_{n-1}, O^*] \), \( [k_n, O^*_n] \) \], where \( 0 = k_0 < k_1 < \cdots < k_n \leq 1 \) and \( O^*_i \subseteq O^* \). Let \( M \) denote a set of risk mitigation strategies. We define a function \( \mu: P \rightarrow M \), where \( \mu(p) \) denotes the risk mitigation strategy associated with permission \( p \). Informally, a risk mitigation strategy \( \mu(p) \) for \( p \in P \) specifies that obligations \( O^*_i \) will be executed if the risk of granting \( p \) is within the interval \([k_i, k_{i+1}]\). Note that a special case of our approach is to define a single risk mitigation strategy that is applicable to all permissions, and this is the approach advocated in Cheng et al.’s work [6].

Formally, given a request \( q = \langle s, p, c \rangle \) and a system state \( \sigma \), we define an authorisation function \( \text{auth} \) as,

\[
\text{auth}(q, \sigma) = \begin{cases} 
(\text{allow}, O^*_i) & \text{if } k_{i} \leq \text{risk}(q, \sigma) \leq k_{i+1}, i \in [1, n) \\
(\text{deny}, O^*) & \text{otherwise.}
\end{cases}
\]

In other words, the request \( \langle s, p, c \rangle \) is permitted but the system must enforce obligations \( O^*_i \) if the risk of allowing \( \langle s, p, c \rangle \) belongs to \([k_i, k_{i+1}]\), and the request \( \langle s, p, c \rangle \) is denied but the system must perform \( O^* \) if the risk is greater than or equal to \( k_n \).

We believe that these risk-based features can be naturally integrated into existing access control models, making them become risk-aware. In role-based access control, for example, we may introduce risk assessment on user-role activation. In this case, a subject \( s \in S \) is regarded as a user or a session, and a permission \( p \in P \) as an approval to activate a particular role. Of course, there exist other possible interpretations of subjects and permissions for RBAC or other access control models. In most cases, a permission is thought of as an approval to perform an operation on a protected resource, and this is the notion defined in the RBAC standard [1], whereas a subject could also be regarded as a role or even a security group.

In order to illustrate the features of RAAC, we introduce a concrete example for accessing patient records in an emergency situation. One evening, Alice is knocked unconscious in a car accident and is taken into the emergency department by an ambulance. The emergency doctor treating her, Bob, would like to view her summary care record (SCR) in order to find out whether there are any important factors to consider, such as any allergies to medications. However, Bob is not allowed to access the SCR via the current activated Doctor role. In this case, Bob attempts to activate EmergencyDoctor role, and the system determines whether to grant this request based on risk assessment. The risk computation depends on two factors associated with the request: the level of competence of Bob to activate this role, and the context (e.g. emergency situation) in which the request was submitted. Eventually, the system deems the risk is acceptable and allows Bob to activate the EmergencyDoctor role, thereby allowing him to access Alice’s SCR. Meanwhile, all those activities are noted in an audit trail, and result in an alert being automatically sent to a privacy officer.

2.2 XACML

XACML is an extensible, XML-encoded language that provides a standard format for authorisation policies and access control decision requests and responses. XACML 3.0 was approved as an OASIS standard on 22 January 2013 [13]. It includes a non-normative data flow model, shown in Figure 1, that describes the major components involved in pro-
Figure 1: The data-flow model for XACML

Users of an XACML-aware application submit requests to access resources made available through that application (step 1). The application includes a policy enforcement point (PEP) which intercepts all access requests. The request is forwarded to the context handler (step 2), which converts it into an XACML request context and sends it to a policy decision point (PDP) (step 3). The PDP evaluates the request context by querying the relevant XACML policies stored in a policy administration point (PAP) (step 4). If those policies refer to additional attributes that are not available in the request context, the PDP will request those attributes from the context handler (step 5), which obtains the relevant attributes from a policy information point (PIP) (steps 6-7). The PIP may be part of the application, such as a username/password file, or external to the application, such as an attribute authority. Then the context handler sends the requested attributes to the PDP (step 8). The PDP evaluates the policies and renders a response to the PEP, which is responsible for enforcing the authorisation decision and fulfilling the obligations (steps 9-12).

XACML uses three basic elements in constructing authorisation policies: <Rule>, <Policy> and <PolicySet>. A <Policy> is the smallest element that the PDP can evaluate, which mainly comprises a <Target>, one or more <Rule>s and a rule-combining algorithm. The <Target> defines a set of conditions that the attribute values in an access request must meet for the policy to apply to the request. A <Rule> comprises an optional <Target> and <Condition> elements and an Effect attribute. The <Condition> defines a Boolean expression that further restricts the applicability of the rule that already implied by the <Target> of the rule. The Effect of the rule that determines the outcome: either Permit or Deny. A <Rule> may also include obligation expressions that refer to operations that must be performed by the PEP in addition to enforcing the PDP’s decision. Note that these are system obligations, not user obligations. More than one rule defined by a policy may be relevant to a request, and so the rule-combining algorithm (Deny-overrides, for example), is used to combine outcomes of these rules into a single decision. <Policy>s may be grouped in <PolicySet>s, each of which uses a policy-combining algorithm that determines how the results of evaluating the policies should be combined.

3. ENCODING RAAC USING XACML

In this section we present an approach to implementing the features of RAAC using XACML. In order to set a context for illustrating our approach, we describe our risk-aware policies based on the XACML RBAC profile [12].

3.1 Risk Mitigation Policies

The XACML RBAC profile (RB-XACML) is designed to address the core and hierarchy components of RBAC. It mainly defines three generic XACML policies: a Role <PolicySet>, a Permission <PolicySet> and a Role Assignment <Policy> or <PolicySet>. A Role <PolicySet> associates a role identifier with a single Permission <PolicySet> using a <PolicySetIdReference> element. A Permission <PolicySet> is used to define a set of permissions, and such a <PolicySet> may reference another Permission <PolicySet> to implement role inheritance. To implement the emergency example described in Sect. 2.1, we can simply define a Role <PolicySet> for EmergencyDoctor which references a Permission <PolicySet>. The Permission <PolicySet> specifies the permission for reading Alice’s SCR, and references a Permission <PolicySet> associated with the normal Doctor role, thereby simulating role inheritance.

RB-XACML states that “a role attribute for a given user is a valid assignment at the time the access decision is requested, and the assignment of role attributes to users . . . is outside the scope of the XACML PDP” [12]. Instead, role enabling authorities (REAs) are used to determine the values of a user’s role attributes. One possible suggestion is that the REA might act as a separate PDP and use a Role Assignment <PolicySet> to determine whether a user can enable a particular role. In order to comply with RB-XACML, we believe that it is most natural to define risk assessment and risk mitigation in conjunction with Role Assignment <PolicySet> to implement risk-aware RBAC using XACML. A pseudo Role Assignment <PolicySet> is shown below, which comprises a <Target> element (lines 02-09) and a <PolicyIdRef> element (line 10). The <Target> determines the <PolicySet> is only applicable to subjects who has a particular attribute (their email name is in the “nhs.com” namespace). It also restricts the resource and action attributes in the request to be EmergencyDoctor role and EnableRole respectively. The <PolicyIdRef> points to a Risk Mitigation <Policy> that further prevents subjects from enabling the EmergencyDoctor by assessing the risk of their requests.

We define a risk mitigation strategy in a Risk Mitigation <Policy> that is treated as a first-class entity. In other word, any Risk Mitigation <Policy> can be referenced in any Role Assignment <PolicySet> without re-specifying the risk mitigation strategy. A Risk Mitigation <Policy> for the emergency example is shown below. This <Policy> consists of a <VariableDefinition> element and two <Rule> elements. Note that the <Target> element in this <Policy> is empty, in which case it is implied by the <Target> of the Role Assignment <PolicySet>. The <VariableDefinition> (lines 03-05) is used to define a risk threshold for the mitiga-
tion strategy, which essentially splits the risk domain \([0, 1]\) into two risk intervals \([0, 0.7]\) and \([0.7, 1]\). Clearly, we can define an arbitrary number of such \(<\text{VariableDefinition}>\) elements (risk thresholds) to have more fine-grained risk intervals. Specifying the risk thresholds in these variables instead of hard-coding them in the rule conditions provides a flexible way to update and maintain the risk mitigation policy. Specifically, a policy administrator only needs to change those variable definitions in order to change the risk thresholds for existing risk intervals.

Now it becomes very natural to write different rules that refer to these variables to implement a risk mitigation strategy. The first \(<\text{Rule}>\) (lines 06-20) has \textit{Permit} as its effect when the condition is satisfied (lines 07-23); that is, the risk value for the access request lies in the interval \([0, 0.7]\). Note that the \(<\text{AttributeDesignator}>\) element is used to retrieve a risk value for the access request, and the returned value must meet the specified criteria such as within the \textit{access-risk} category and being issued by a trusted authority (lines 10-11). We describe how this mechanism works in the next section. Similarly, the second \(<\text{Rule}>\) (lines 27-37) has \textit{Deny} as its effect if the risk value lies in the interval \([0.7, 1]\). Additionally, both rules contain \(<\text{ObligationExpressions}>\) (lines 24-25 and line 36) which are evaluated into obligations by the PDP.

It can be seen that we can define two or more \(<\text{Rule}>\)s in the Risk Mitigation \(<\text{Policy}>\), each of which corresponds to checking a risk interval. For the sake of readability, we arrange the rules in an order with respect to the risk intervals from low to high (\([0, 0.7]\) to \([0.7, 1]\), for example). This naturally leads us to use the \textit{first-applicable} algorithm \([13, \text{ Appendix C}]\) for combining the results of rules in the Risk Mitigation \(<\text{Policy}>\). This algorithm forces the evaluation of the rules in the order as listed in the policy, and ensures that for a particular rule, if its target and condition are evaluated to \textit{True}, then the result for the policy is the effect of the rule (\textit{Permit} or \textit{Deny}). For example, if the \(<\text{Rule}>\) in lines 06-26 evaluates to \textit{Permit}, then the second \(<\text{Rule}>\) is not evaluated, and a value of \textit{Permit} is returned for the \(<\text{Policy}>\) (lines 01-38).

### 3.2 Risk Assessment

Recall that, given an access request, a Role Assignment \(<\text{PolicySet}>\) is evaluated to \textit{Permit} for the request only if the conditions defined in its target are met by the request and its associated Risk Mitigation \(<\text{Policy}>\) evaluates to \textit{Permit} (which means the risk of granting this request lies in an acceptable risk interval). Let us now look at how to use XACML to compute the risk associated with an access request in more detail. As we mentioned, the risk calculation generally depends on various factors associated with the entities appeared in the request. Since XACML itself supports the use of attributes when constructing request contexts and policies, it is natural to express these factors as attributes and choose an suitable XACML function to combine these attributes into a risk value in the rule condition. As a generic solution, however, the XACML predefined functions are limited; it is also not clear whether XACML accommodates the definition of an arbitrary new function, such as the complex formula used to compute risk in multi-level security \([6]\). Instead we propose a method in which the risk calculation is conducted in the PIP. As shown in the previous section, we introduce a special attribute, namely \textit{risk}, under the \textit{access-risk} category and require that the values for this attribute are issued by a special trusted authority. When evaluating the Risk Mitigation \(<\text{Policy}>\), the PDP is instructed to request values for this risk attribute in the request context from the context handler. The context handler may retrieve this risk value from the PIP and then supply the required values into the request context. This suggests that the PIP should be able to compute the risk value at run-time when requested by the context handler, and this is compliant with the requirement of RAAC regarding dynamic risk analysis.

We explored this approach by implementing our medical emergency example based on the Balana\(^2\). The Balana is an open source Java implementation of XACML.
ana implementation provides interfaces that allow us to extend the PIP to perform risk retrieval and risk calculation in a modular way. This was done by extending the AttributeFinder module with three additional modules, each of which is responsible for finding attributes relevant to a particular category (subject, resource or environment). This RiskAttributeFinder module is responsible for finding a risk value corresponding to the access-risk category. It may obtain these values by querying an external system (anomaly detection system, for example) (step w) or a risk assessment module built inside the PIP (step 1). In the later case, a generic RiskCalculator module is used to connect the RiskAttributeFinder module with the other three modules. Specifically, the RiskAttributeFinder invokes the RiskCalculator, supplying attributes obtained from the request context (typically, the subject-id and the resource-id). On the basis of this information, the RiskCalculator obtains additional attributes (subject, resource and environment) that are needed for the risk computation from the three standard modules (steps 2a-2c), and computes a risk value according to a specific method (step 3). In our implementation we instantiate the RiskCalculator module with a method that accumulates risk factors (competence and environmental threat) into a single value.

Although we illustrate our approach to implement RAAC in the RB-XACML setting, it is self-contained and can be employed on any existing XACML applications. In particular, all risk-based features are embodied in the Risk Mitigation <Policy> that can be referenced in other policies making them risk-aware. When the PDP evaluates the Risk Mitigation <Policy>, it sees the risk assessment process as normal attribute retrieval, the risk attribute value being obtained from the PIP. This separation of risk assessment (PIP) and risk-aware policy evaluation (PDP) conforms with the spirit of the XACML standard for developing distributed authorisation systems.

4. EXTENDING THE MODEL

In many situations, risk mitigation actions are imposed on requesting users in order to manage the risk of allowing them to access resources. However, unlike system obligations, user obligations may go unfulfilled, simply because the user has either deliberated failed or forgotten to take obliged actions. In this context, from the point of view of safety, the system must introduce mechanisms to incentivise users to fulfill obligations, and account for the risk incurred.

Formally, we define a function $b : S \rightarrow [0, 1]$, where $b(s)$ denotes the amount of budget for subject $s$. The budget of $s$ is used to pay deposits for granting risky accesses, and these deposits will only be returned if $s$ fulfills obligations resulting from such accesses. We model a user obligation $o^s$ as a pair $(a, l)$, where $a$ is an action and $l$ is a symbolic temporal interval (for example, Instantly, 10Hours) during which $a$ must be performed. We write $d \in [0, 1]$ to denote an amount of deposit that the requesting user is required to pay for some risky access to be granted. Then we extend a risk mitigation strategy to be a list $[(k_0, O_0^s; \emptyset, 0), (k_1, O_1^s, O_1^s, d_1), \ldots, (k_n, O_n^s, \emptyset, 0)]$, where $0 = k_0 < k_1 < \cdots < k_n$, $k_n < 1$, $O^s_1 \subseteq O^s_0 \subseteq \ldots \subseteq O^s_n \subseteq \emptyset^*$ and $0 \leq d_1 < \cdots < d_{n-1} \leq 1$. Given a request $q = (s, p, c)$ and a system state $\sigma$, we define an extended authorisation function $\text{auth}_s$ as,

$$\begin{align*}
\text{auth}_s(q, \sigma) = \begin{cases} 
(\text{allow}, O_0^s, \emptyset, 0) & \text{if risk}(q, \sigma) < k_1, \\
(\text{bud}, O_i^s, O_i^s, d_i) & \text{if risk}(q, \sigma) \in [k_i, k_{i+1}), \\
(\text{deny}, O_n^s, \emptyset, 0) & \text{otherwise},
\end{cases}
\end{align*}$$

where the budget checker function $\text{bud}$ is defined as,

$$\text{bud}(q, O_i^s, O_i^s, d_i) = \begin{cases} 
(\text{allow}, O_i^s, O_i^s, d_i) & \text{if } b(s) \geq d_i, \\
(\text{deny}, O_i^s, \emptyset, d_i) & \text{otherwise}.
\end{cases}$$

Informally, the semantics for how $\text{auth}_s$ works is very close to $\text{auth}$ as explained in Sect 2.1, so we omit its explanation here. It is, however, important to explain the basic ideas behind the $\text{bud}$ function. Essentially, specifying deposits in a risk mitigation strategy provides a means of sanctioning subjects, as well as restricting the number of risky accesses to be granted to subjects. If subject $s$, for example, has sufficient budget to pay deposit $d_i$, then $d_i$ will be deducted from $s$'s budget and $d_i$ will be returned when $O_i^s$ are fulfilled. If $s$'s budget is less than the required deposit $d_i$ because of failures to fulfill prior obligations, or outstanding obligations to be fulfilled, then $s$ is denied access to the resource, even though the risk of allowing the access would be acceptable in other circumstances.

There are a few operations that the PEP is required to execute when receiving a response from the $\text{auth}$ function. In fact, given a request $(s, p, c)$, the actions that the PEP has to perform vary depending on one of four possible results being returned:

- $\{\text{allow}, O_i^s, \emptyset, 0\}$ and $\{\text{deny}, O_i^s, \emptyset, 0\}$: The PEP is required to perform system obligation $O_i^s$ in conjunction with enforcing an authorisation decision (either “allow” or “deny”).

- $\{\text{allow}, O_i^s, \emptyset, d_i\}$: In addition to enforcing the “allow” decision and executing $O_i^s$, when $s$ accepts $O_i^s$, the PEP would be able to transform $O_i^s$ into a more concrete form whose execution the system can monitor.
At the same time, the PEP needs to deduct the deposit $d$ from $s$'s budget and return it when $O_f$ are fulfilled.

- (deny, $O_f$, $\emptyset$, $d$): In addition to enforcing the “deny” decision and executing $O_f$, the PEP needs to show a message dictating that $s$'s budget is not sufficient to pay deposit $d$ for gaining this access. This is to inform $s$ to seek a way to increase her budget, as the risk of granting this access is acceptable.

In the remainder of this section, we show how XACML can implement this extended model.

### 4.1 Complex Risk Mitigation Policies

To implement the extended model, we first look at how to extend the Risk Mitigation <Policy> to include the concept of a budget check. Recall that the Risk Mitigation <Policy> includes <Rule>s, each of which is used to check whether the risk lies in a specific interval. For some rules that have a Permit effect (which means the risk is acceptable), we now need to further split each of them into two cases: whether the requesting subject has sufficient budget or not. In order to provide greater flexibility and economy in writing such complex rules, we take an approach that, for every risk interval, two boolean variables are defined: one for the risk check and the other for the budget check. These boolean variables can be referenced many times when defining rules.

An example of a Complex Risk Mitigation <Policy> is shown below. The <VariableDefinition> (lines 03-06) define two risk thresholds that form three risk intervals: $[0, 0.2), [0.2, 0.7]$ and $[0.7, 1]$. The <VariableDefinition> (lines 07-08) introduces a budget value 0.3 that corresponds to the risk interval $[0.2, 0.7]$.

The <VariableDefinition> (lines 09-14) define three boolean variables, each of which holds a True value if the risk value for the request lies in the corresponding interval and False otherwise. Similarly, the boolean value in the <VariableDefinition> (lines 15-16) indicates whether the requesting subject’s budget is greater than 0.3. The budget of the subject is treated as an attribute, and its value is retrieved by the means of an <AttributeDesignator> with the access-subject category defined in the policy. The policy consists of four rules, each of which simply includes one or more references to the predefined <VariableDefinition> elements (risk check and budget check) as a condition to restrict its applicability. For example, the second rule (lines 20-46) is applicable (evaluates to Permit) if its referred two <VariableDefinition> elements (lines 23-26) are both True.

XACML allows every <Rule> to include a set of <ObligationExpression> elements, each of which represents an obligation. An <ObligationExpression> can include an arbitrary number of attribute assignments that forms the arguments of the action defined by the obligation. For example, the obligation expression with user:email id (lines 31-44) defines three attributes: the first attribute (lines 32-35) specifies who is obliged to fulfill the obligation; the second (lines 36-39) indicates the email address of the requestor’s line manager; and the third (lines 40-43) defines the temporal interval (1 day and 2 hours) during which the requestor

\[ \text{\textbf{In the extended model, there is no budget associated with the least and most risky intervals (0.2 and 0.7, 1), for example, hence we simply omit them here.}} \]

\[ \text{\textbf{has to perform the obligation. Note that the ObligationId attribute (line 31) identifies the action; that is, oblige the requestor to send an email. When evaluating this <ObligationExpression>, the PDP determines the values for requestor and emailId at runtime by the means of an <AttributeDesignator>, and sends the resulting obligation to the PEP in the response context. As stated in the XACML specification, the PEP itself has to know how to handle the obligation when receiving the response.}} \]

**Figure 5: Complex Risk Mitigation <Policy>**
4.2 Implementing the Obligations Service

We now propose an extended PEP to support obligation monitoring and enforcement as shown in Fig. 6. When the PEP receives a response that contains an authorisation decision and a list of obligations (system and user obligations), it passes these obligations to the obligation handler before enforcing the decision (step 1). The obligation handler classifies these obligations into two sets: system obligations and user obligations, and forwards system obligations to the obligation enforcement point (step 2). The obligation enforcement point performs the system obligations by invoking corresponding fulfillment module. The fulfillment modules are responsible for correctly enforcing the obligations, each of which is designed to support a particular type of action. For example, one module handles the deduction of a deposit from a requester’s budget, and the other is to execute logging and auditing. Once these system obligations are successfully performed, the PEP will be notified to proceed with the decision enforcement⁴ (steps 3-4b). Meanwhile, the obligation handler forwards user obligations to the fulfillment monitor (step 4c), which makes these obligations into active system states according to their temporal constraints and current system clock. Additionally, the fulfillment monitor maintains the list of pending user obligations. It checks whether any pending obligations is fulfilled by a user-initiated action, and if any have fulfilled, it notifies the obligation handler. It also monitors whether any pending obligations that have deadlines that have passed without appropriate actions performed, and notifies the obligation handler of those violations (steps 5-6). The fulfillment monitor consists of a number of monitor modules, each of which is implemented to be able to monitor a particular type of user obligation. In order to implement the control of budget in our extended model, we instruct the obligation handler, for each corresponding decision, to process the list of obligations in order \((o_0, o_1, \ldots, o_i, o_{i+1}, \ldots, o_n)\), where \(o_0\) is the system action that deducts deposit from the requester’s budget, while \(o_i\) is the system action that returns the deducted budget back to the requester’s account. This means that \(o_0\) will be only executed if the previous user obligations \((o_1, \ldots, o_{i-1})\) are successfully fulfilled.

In short, we propose an architectural structure for implementing the obligations service component defined in the XACML standard. Our proposed approach supports the enforcement of both system and user obligations in a generic manner without changing the standard XACML architecture and language.

5. DISCUSSION

There is a considerable body of work on risk-aware access control, much of it focusing on developing models for incorporating risk in multi-level security [6, 10] and role-based access control [2, 4]. Very little of that research is concerned with the authorisation architectural design that accommodates the awareness of risk, with the exception of the work of Chen et al. [3]. The latter work extends XACML with new XML languages and functional components to support risk-adaptive access control. In contrast, our approach to implementing RAAC is fully compliant with the XACML standard without introducing extra elements.

Another work that is most closely related to RAAC is due to Martínez-García et al. [9] who developed a distributed access control system for MedIGS, a multi-agent middleware for the medical data sharing between different hospitals. The goal was to maintain, for every different institution, its own security policies, providing at the same time a model for allowing the interoperability of the different security systems in a multi-agent environment. This is done by defining a scheme for attribute translation between the different institution’s domain, based on static policies specified offline by a domain’s authority. Every attribute translation is associated with a magnitude value in the range [0,1] that represents the degree of possession of the attribute. For every domain, two security thresholds define three region in which, according to its magnitude, an attribute conversion can be accepted, denied or submitted for validation through post-processing. We believe that, given the highly dynamic environment in which multi agents systems usually operate, there is need for a runtime evaluation for the magnitude that is able to adapt to continuously changing context and subjects. Therefore, our approach to risk computation on the runtime is complementary to this work on the translation of attributes.

The XACML standard treats a system obligation as an attribute assignment, and leaves the interpretation of these obligations to the PEP. Further, it does not provide support for user obligations. In the XACML technical committee, there is some work, called “Obligation Families” [11], which attempts to define additional mechanisms for obligation processing and enforcement, but this is very preliminary and is not reflected in the current XACML standard. Additionally, Li et al. [8] recently introduce a comprehensive specification and processing model for obligations by extending XACML specification and architecture. Certainly this work is complementary to our approach to handling obligations, but we take a different view that, for someone who is committed to existing XACML architecture, it would be straightforward to use our approach for supporting user obligations and incorporating the notion of risk.

6. CONCLUSION

In this paper we have explored the use of XACML as a means of implementing risk-aware access control. We have

---

⁴This is consistent with the XACML specification: an authorisation decision can only be enforced if the PEP can understand and discharge the returned obligations [13, Section 2]
provided a simple and flexible way to encode risk mitigation strategies by risk mitigation <Policy>, and illustrated how these risk mitigation policies can be naturally integrated into the XACML RBAC profile, making it risk-aware. We then discussed our approach to utilizing the PIP for risk attribute retrieval and risk calculation. Finally, since obligations are essential elements in RAAC for mitigating the risk incurred by granting access, we proposed extensions to the XACML obligations service that would permit efficient enforcement of the both system and user obligations.

7. REFERENCES


