XACML and Risk-Aware Access Control

Liang Chen, Luca Gasparini and Timothy J. Norman

dot.rural Digital Economy Hub, University of Aberdeen, Aberdeen, U.K.*

Abstract. Risk-aware access control (RAAC) has shown promise as an approach to addressing the increasing need to share information securely in dynamic environments. For such models to realise their promise, however, principled, standard-based software engineering methods are essential. XACML is an XML-based OASIS standard for the specification and evaluation of access control policies. In this paper we explore the use of XACML as a means of implementing RAAC. We abstract core components of RAAC relevant to risk management, and show how these may be implemented using standard XACML features.

1 Introduction

Inspired by addressing the increasing need to share information in dynamic environments, risk-aware access control (RAAC) has been the subject of considerable research in recent years [1–6]. 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 (risks associated with undesirable information disclosure or modification, for example). RAAC systems are designed to be more permissive than traditional access control mechanisms, in the sense that more risky access is allowed provided that the risk is effectively managed. Typically, from the safety perspective, the system employs some risk mitigation methods to account for and reduce such risks.

OASIS has introduced a standard XML-based language, namely the eXtensible Access Control Markup Language (XACML) [7], for the specification and evaluation of access control policies. It offers a standard policy exchange format, and supports fine-grained authorisation policies that are implementation independent. 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 not been studied with the except of Chen et al. [8]. In this paper we present a simple and widely applicable approach to build RAAC using XACML. Our main contributions can be summarised as follows.

– On the basis of the RAAC models developed by Chen et al. [3], we present a model that abstracts common system components relevant to risk management. These components can be naturally integrated into existing access control models, making them risk-aware.

* This research is supported by the award made by the RCUK Digital Economy programme to the dot.rural Digital Economy Hub; award reference: EP/G066051/1.

Chen L., Gasparini L. and Norman T..
XACML and Risk-Aware Access Control.
DOI: 10.5220/0004609200660075
In *Proceedings of the 10th International Workshop on Security in Information Systems* (WOSIS-2013), pages 66-75
ISBN: 978-989-8565-64-8

– Exploring the RBAC profile of XACML [9], we illustrate how to define XACML policies to implement the core components of our abstract model. The benefit here is that, for those who implement RBAC using this profile, it would be straightforward to adopt our approach to incorporate risk management into their RBAC solution.
– We show how risk assessment may be implemented in the XACML policy information point. In particular, our approach supports risk assessment in a generic manner without requiring modification of the XACML standard.

2 Background

2.1 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 in January 2013 [7]. It includes a non-normative data flow model, shown in Fig. 1, that describes the major components involved in processing access requests.

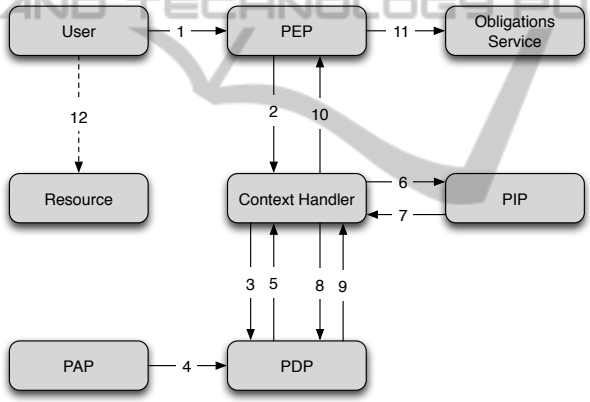

Fig. 1. The data-flow model for XACML [7].

Users of an XACML-aware application submit requests for resources through their 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 any obligations (steps 9-12). The possible decision values are: Permit, Deny, NotApplicable (no policies or rules are applicable to the access request), or Indeterminate (some error occurs during evaluation).

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>`[1] and `<Condition>` elements and an `Effect` attribute. The `<Condition>` further restricts the applicability of the rule already implied by the `<Target>` of the rule. The `Effect` of the rule 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. 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. These `<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.

2.2 An Abstract Model for RAAC

To provide a basis for our XACML-based mechanism for RAAC, we summarise an abstract model for RAAC based on earlier work by Chen et al. [3]. 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, each representing an active entity in a system that may request access to resources. In the context of RAAC, access control decisions are made on the basis of a risk analysis. In general, the risk of granting a permission to a subject can be interpreted as the likelihood of the permission being misused 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, and the context (e.g. device, location or current time) from which the subject is requesting. We model an access request as a tuple $\langle s, p \rangle$, where $s \in S$ and $p \in P$. Let Σ denote a set of system states, and let $\mathcal{K} = \{k \in \mathbb{R} : 0 \leqslant k \leqslant 1\}$ denote a risk domain. We define a risk function $\mathsf{Risk} : Q \times \Sigma \to \mathcal{K}$ that takes as input an access request $q = \langle s, p \rangle \in Q$ 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-dependant, and thus outside the scope of this paper.

From the system's perspective, we need to determine risk thresholds 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

[1] This target may be omitted, in which case it is assumed to be the target of the policy to which the rule belongs.

and risk mitigation on a per-permission basis. We write $[k, k')$ to denote the *risk interval* $\{x \in \mathcal{K} : k \leqslant x < k'\}$. Let $\mathcal{O}$ denote a set of *obligations*, where $o \in \mathcal{O}$ is some action that must be taken by the system when enforcing an access control decision (as in XACML [7]). Then we define a *risk mitigation strategy* to be a list $[\langle k_0, O_0\rangle, \langle k_1, O_1\rangle, \ldots, \langle k_{n-1}, O_{n-1}\rangle, \langle k_n, O_n\rangle]$, where $0 = k_0 < k_1 < \cdots < k_n \leqslant 1$ and $O_i \subseteq \mathcal{O}$. Let M denote a set of risk mitigation strategies. We define a function $\mu : P \to M$, 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; the approach advocated in Cheng et al.'s work [4].

Formally, given a request $q = \langle s, p\rangle$ and a system state σ, we define an authorisation function Auth as,

$$\mathsf{Auth}(q, \sigma) = \begin{cases} \langle \mathsf{allow}, O_i\rangle & \text{if } \mathsf{Risk}(q, \sigma) \in [k_i, k_{i+1}),\ 1 \leqslant i < n, \\ \langle \mathsf{deny}, O_n\rangle & \text{otherwise.} \end{cases}$$

In other words, the request $\langle s, p\rangle$ is permitted but the system must enforce obligations O_i if the risk of allowing $\langle s, p\rangle$ belongs to $[k_i, k_{i+1}]$, and the request $\langle s, p\rangle$ is denied but the system must perform O_n 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 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 [10], 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 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 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 the `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.

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 [9].

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. 2.2, we can simply define a Role `<PolicySet>` for `EmergencyDoctor` which references a Permission `<PolicySet>`. The Permission `<PolicySet>` specifies the permission for reading patients' SCRs, 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" [9]. 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 02-06) and a `<PolicyIdRef>` element (line 07). The `<Target>` specifies that the `<PolicySet>` is only applicable to subjects who have 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` role by assessing the risk of their requests.

```
00 <!-- Role Assignment <PolicySet> -->
01 <PolicySet PolicySetId="emergencydoctor:role:requirements"...>
02   <Target>
03     <AnyOf><AllOf><Match>has email address *@nhs.com</Match></AllOf></AnyOf>
04     <AnyOf><AllOf><Match>EmergencyDoctor</Match></AllOf></AnyOf>
05     <AnyOf><AllOf><Match>EnableRole</Match></AllOf></AnyOf>
06   </Target>
07   <PolicyIdRef>rm:audit</PolicyIdRef>
08 </PolicySet>
```

We define a risk mitigation strategy in a Risk Mitigation <Policy> that is treated as a first-class entity. In other words, 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 02-04) is used to define a risk threshold for the mitigation 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 <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 <Rule> (lines 05-28) has Permit as its effect when the condition is satisfied (lines 06-19); that is, the risk value for the access request lies in the interval $[0, 0.7)$. Note that the <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 access-risk category and being issued by a trusted authority (line 09). We describe how this mechanism works in the next section. Similarly, the second <Rule> (lines 29-36) has Deny as its effect if the risk value lies in the interval $[0.7, 1]$. Additionally, both rules contain <ObligationExpression> elements (lines 21, 22 and 35), 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 <system:alert> id (lines 22-26) defines an email attribute which indicates that the system is obliged to send an email to a privacy officer. When evaluating this <ObligationExpression>, the PDP determines the value for <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. To this end, we propose a concrete form of the XACML obligation service for interpreting and enforcing different types of obligations. We provide some more detail of our implementation of this obligations service in Sect. 4.

```
00 <!-- Risk Mitigation <Policy> -->
01 <Policy PolicyId="rm:audit" RuleCombiningAlgId="first-applicable">
02   <VariableDefinition VariableId="risk-threshold-1">
03     <AttributeValue>0.7</AttributeValue>
04   </VariableDefinition>
05   <Rule RuleId="first-risk-interval" Effect="Permit">
06     <Condition><Apply FunctionId="function:and">
07       <Apply FunctionId="double-greater-than-or-equal">
08         <Apply FunctionId="function:double-one-and-only">
09           <AttributeDesignator Category="access-risk" AttributeId="risk" Issuer="TA"/>
```

```
10          </Apply>
11          <AttibuteValue>0</AttributeValue>
12        </Apply>
13        <Apply FunctionId="function:double-less-than">
14          <Apply FunctionId="function:double-one-and-only">
15            <AttributeDesignator Category="access-risk" AttributeId="risk" Issuer="TA"/>
16          </Apply>
17          <VariableReference VariableId="risk-threshold-1"/></VariableReference>
18        </Apply>
19      </Apply></Condition>
20      <ObligationExpressions>
21        <ObligationExpression ObligationId="system:log">...</ObligationExpression>
22        <ObligationExpression ObligationId="system:alert">
23          <AttributeAssignmentExpression AttributeId="emailId">
24            <AttributeDesignator AttributeId="officer-email" Category="access-subject"/>
25          </AttributeAssignmentExpression>
26        </ObligationExpression>
27      </ObligationExpressions>
28    </Rule>
29    <Rule RuleId="second-risk-interval" Effect="Deny">
30      <Condition><Apply FunctionId="double-greater-than-or-equal">
31        <Apply FunctionId="function:double-one-and-only">
32          <AttributeDesignator Category="access-risk" AttributeId="risk" Issuer="TA"/>
33        </Apply>
34        <VariableReference VariableId="risk-threshold-1"></Apply></Condition>
35      <ObligationExpression ObligationId="system:log">...</ObligationExpression>
36    </Rule>
37  </Policy>
```


It can be seen that we can define two or more `<Rule>`s in the Risk Mitigation `<Policy>`, each of which corresponds to checking a risk interval. For the sake of readability, we arrange these rules in order of the risk intervals from low to high ($[0, 0.7)$ to $[0.7, 1]$, for example). This naturally leads us to use the `first-applicable` algorithm [7, Appendix C] for combining the results of rules in the Risk Mitigation `<Policy>`. This algorithm forces the evaluation of the rules in the order listed in the policy, and ensures that for a particular rule, if its target and condition are evaluated to `True`, then the result for the policy is the effect of the rule (`Permit` or `Deny`). For example, if the `<Rule>` in lines 05-28 evaluates to `Permit`, then the second `<Rule>` is not evaluated, and a value of `Permit` is returned for the `<Policy>` (lines 01-37).

3.2 Risk Assessment

Recall that, given an access request, a Role Assignment `<PolicySet>` is evaluated to `Permit` for the request only if the conditions defined in its target are met by the request and its associated Risk Mitigation `<Policy>` evaluates to `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 appearing 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 [4]. 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 `risk`, under the `access-risk` category and require that the values for this attribute are issued by a special trusted authority. When evaluating the Risk Mitigation `<Policy>`, the PDP is instructed to request values for this risk attribute in the request context from the context hander. 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 complies with the requirement of RAAC regarding dynamic risk analysis.

We explored this approach by implementing our medical emergency example based on Balana[2]. The Balana implementation provides interfaces that allow us to extend the PIP to perform risk retrieval and risk calculation in a modular way as shown in Fig. 2. 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 (an anomaly detection system, for example) (step w) or a risk assessment module built inside the PIP (step 1). In the later case, a generic `RiskAssessment` module is used to connect the `RiskAttributeFinder` module with the other three modules. Specifically, the `RiskAttributeFinder` calls the `RiskAssessment` (step 1), supplying attributes obtained from the request context (typically, the subject-id and the resource-id). On the basis of this information, the `RiskAssessment` 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 `RiskAssessment` module with a method that accumulates risk factors (competence and environmental threat) into a single value.

4 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 [4, 6] and role-based access control [1, 2]. Very little of that research is concerned with the design of authorisation architectures that accommodate the awareness of risk, with the exception of Chen et al. [8]. This work extends the XACML standard with new XML schema for policies and additional components to support risk-adaptive access control. In contract,

[2] Balana is an open source Java implementation of XACML 3.0, extending the Sun XACML 2.0 implementation: http://xacmlinfo.com/category/balana/.

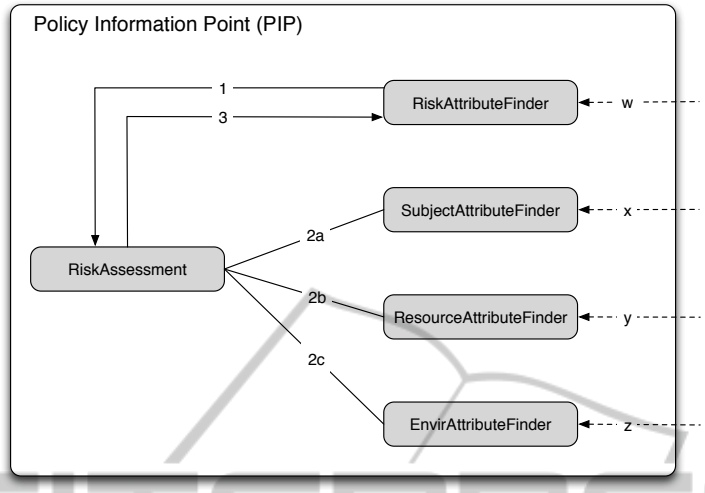

Fig. 2. The extended PIP architecture for supporting risk assessment.

our approach to implementing RAAC is fully compliant with the XACML standard without introducing extra elements.

The XACML standard treats an obligation as an attribute assignment, and leaves the interpretation of these obligations to the PEP. 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 preliminary and is not reflected in the current XACML standard. Instead, we implement the XACML obligation service as a comprehensive *obligation handler* that supports obligation monitoring and enforcement. In particular, our proposed architecture of the obligation service supports handling of *user obligations* which are used as one type of risk mitigation methods in RAAC [3]. Space does not permit a detailed presentation and evaluation of our approach, however, this will be the subject of future work.

Building upon our implementation of the proposed risk-aware XACML architecture developed upon Balana, another avenue for future work is to apply this approach in a real-world system. Within the context of the TRUMP project[3], we are applying RAAC models as part of a trusted infrastructure for mobile healthcare application for chronic illness including diabetes.

5 Conclusions

In this paper we have proposed an approach that uses standard XACML features to implement RAAC. We provided a simple and flexible way to encode risk mitigation and risk-aware authorisation by risk mitigation `<Policy>`s, and illustrated how these policies can be referenced in other policies making them risk-aware. Although we illustrate our approach in the RB-XACML setting, our approach is self-contained and can be employed on any existing XACML applications. We also discussed our approach to

[3] http://www.trump-india-uk.org.

utilizing the PIP for risk attribute retrieval and risk calculation. 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.

References

1. Bijon, K. Z., Krishnan, R., Sandhu, R. S.: Risk-aware RBAC sessions. In: Proceedings of the 8th International Conference on Information Systems Security. (2012) 59–74
2. Chen, L., Crampton, J.: Risk-aware role-based access control. In: Proceedings of the 7th International Workshop on Security and Trust Management. (2011) 140–156
3. Chen, L., Crampton, J., Kollingbaum, M. J., Norman, T. J.: Obligations in risk-aware access control. In: Proceedings of the Tenth Annual Conference on Privacy, Security and Trust. (2012) 145–152
4. Cheng, P. C., Rohatgi, P., Keser, C., Karger, P. A., Wagner, G. M., Reninger, A. S.: Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy. (2007) 222–230
5. Kandala, S., Sandhu, R. S., Bhamidipati, V.: An attribute based framework for risk-adaptive access control models. In: Proceedings of the Sixth International Conference on Availability, Reliability and Security. (2011) 236–241
6. Ni, Q., Bertino, E., Lobo, J.: Risk-based access control systems built on fuzzy inferences. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. (2010) 250–260
7. OASIS: eXtensible Access Control Markup Language (XACML) Version 3.0. (2013) OASIS Standard (E. Rissanen, editor).
8. Chen, C., Han, W., Yong, J.: Specify and enforce the policies of quantified risk adaptive access control. In: Proceedings of the 14th International Conference on Computer Supported Cooperative Work in Design. (2010) 110–115
9. OASIS: XACML v3.0 Core and hierarchical Role Based Access Control (RBAC) profile Version 1.0. (2010) Committee Specification (E. Rissanen, editor).
10. American National Standards Institute: American National Standard for Information Technology – Role Based Access Control. (2004) ANSI INCITS 359-2004.
11. OASIS: XACML v3.0 Obligation Families Version 1.0. (2007) OASIS Working Draft (E. Rissanen, editor).