A Lightweight JSON-based Access Control Policy Evaluation Framework

Hao Jiang, Ahmed Bouabdallah
IMT Atlantique
France
jiang.hao@outlook.com
ahmed.bouabdallah@telecom-bretagne.eu

ABSTRACT: *Along with the rapid development of ICT technologies, new areas like Industry 4.0, IoT and 5G have emerged and brought out the need for protecting shared resources and services under time-critical and energy-constrained scenarios with real-time policy-based access control. The process of policy evaluation under these circumstances must be executed within an unobservable delay and strictly comply with security objectives. To achieve this, the policy language needs to be very expressive but lightweight and efficient. Many existing implementations are using XML (Extensible Markup Language) to encode policies, which is verbose, inefficient to parse, and not readable by humans. On the contrary, JSON (JavaScript Object Notation) is a lightweight, text-based and language-independent data interchange format that is simple for humans to read and write and easy for machines to parse and generate. Several attempts have emerged to convert existing XML policies and requests into JSON, however, there are very few policy specification proposals that are based on JSON with well-defined syntax and semantics. This paper investigates these challenges, and identifies a set of key requirements for a policy language to optimize the policy evaluation performance. According to these performance requirements, we introduce JACPoL, a descriptive, scalable and expressive policy language in JSON. JACPoL by design provides a flexible and fine-grained ABAC (Attribute-based Access Control), and meanwhile it can be easily tailored to express a broad range of other access control models. This paper systematically illustrates the design and implementation of JACPoL and evaluates it in comparison with other existing policy languages. The result shows that JACPoL can be as expressive as existing ones but more simple, scalable and efficient.*

Keywords: Lightweight Policy Language, JSON, Real-time Access Control, Policy Language Assessment, Policy Evaluation Framework

Received: 2 August 2017, Revised 11 September 2017, Accepted 18 September 2017

1. Introduction

Access control is an important security mechanism involving user specified policies to determine the actions that a principal can perform on resources. Typically, the access requests are intercepted and analyzed by a PEP (Policy Enforcement Point), which then transfers the request details to a PDP (Policy Decision Point) for evaluation and authorization decision [2]. In most implementations, the stateless nature of PEP enables its ease of scale. However, the PDP has to consult the right policy set and apply the rules therein to reach a decision for each request and thus is often the performance bottleneck of policy-based access control systems. Therefore, a policy language determining how policies are expressed and evaluated is important and has a direct influence on the performance of the PDP.

Especially, in nowadays, protecting private resources in real-time has evolved into a rigid demand in domains such as home automation, smart cities, health care services and intelligent transportation systems, etc., where the environments are characterized by heterogeneous, distributed computing systems exchanging enormous volumes of time-critical data with varying levels of access control in a dynamic network. An access control policy language for these environments needs to be very well-structured, expressive but lightweight and easily extensible [3].

In this paper, we investigate the relationship between the performance of the PDP, the language that is used to encode the policies and the access requests that it decides upon, and identify a set of key requirements for a policy language to guarantee the performance of the PDP. We argue that JSON would be more efficient and suitable than other alternatives (XML, etc.) as a policy data format in critical environments. According to these observations, we proposed a simple but expressive access control policy language (JACPoL) in JSON. A PoC (Proof of Concept) has been conducted through the implementation of JACPoL in a policy engine operated in the testbed of an European project [1]. At last we carefully positioned JACPoL in comparison with other existing policy languages.

The main contribution of this work is therefore the definition of JACPoL, which utilizes JSON to encode a novel access control policy specification language with well-defined syntax and semantics. We identify key requirements and technical trends for future policy languages. We incidentally propose the new notion of Implicit Logic Operators (ILO), which can greatly reduce the size and complexity of a policy set while providing fine-grained access control. We also elaborate on the applicability of JACPoL on ABAC model, RBAC model and their combinations or their by-products. Last but not least, our implementation leads to a novel and performant policy engine adopting the PDP/PEP architecture [2] and JACPoL policy language based on Node.js and Redis.

The remainder of this paper is structured as follows. In Section 2, we refine our problematic by delimiting precisely its perimeter. In Section 3 we illustrate in depth with representative policy examples the design of our policy language in terms of the constructs, semantics and other important features like Implicit Logic Operators, combining algorithms and implementation. Section 4 further evaluates JACPoL and compares it with other existing access control policy specification languages. The ABAC nature of JACPoL is detailed in Section 5 with a comprehensive discussion on other possible application of JACPoL to ARBAC (Attribute-centric RBAC) and RABAC (Role-centric ABAC) security models. Section 6 summarizes our work and discuss future research directions.

2. Problematic

In the past decades, a lot of policy languages have been proposed for the specification of access control policies using XML, such as EPAL [6], X-GTRBAC [7] and the standardized XACML [8]. Nevertheless, it is generally acknowledged that XACML suffers from providing poorly defined and counterintuitive semantics [9], which makes it not good in simplicity and flexibility. On the other hand, XML performs well in expressiveness and adaptability but sacrifices its efficiency and scalability, compared to which JSON is considered to be more well-proportioned with respect to these requirements, and even simpler, easier, more efficient and thus favored by more and more nowadays' policy designers [10][11][12][13].

To address the aforementioned inefficiency issues of the XML format, the XACML Technical Committee recently designed the JSON profile [18] to be used in the exchange of XACML request and response messages between the PEP and PDP. However, the profile does not define the specification of XACML policies, which means, after the PDP parses the JSON-formatted XACML requests, it still needs to evaluate the parsed attributes with respect to the policies expressed in XML. Leigh Griffin and his colleagues [12] have proposed JSONPL, a policy vocabulary encoded in JSON that semantically was identical to the original XML policy but stripped away the redundant meta data and cleaned up the array translation process. Their performance experiments showed that JSON could provide very similar expressiveness as XML but with much less verbosity. On the other hand, as much as we understand, JSONPL is merely aimed at implementing XACML policies in JSON and thus lacks its own formal schema and full specification as a policy specification language [19].

Major service providers such as Amazon Web Services (AWS) [20] have a tendency to implement their own security languages in JSON, but such kind of approaches are normally for proprietary usage thus provide only self-sufficient features and support limited use cases, which are not suitable to be a common policy language. To the best of our knowledge, there are very few proposals that combine a rich set of language features with well-defined syntax and semantics, and such kind of access control policy language based on JSON has not even been attempted before and as such JACPoL can be considered to be an original and innovative contribution.

3. JACPOL Detailed Design

This section presents JACPoL in depth. We first recall the foundations of JACPoL, and then introduce its structures with an overview of how an access request is evaluated with respect to JACPoL policies. After that, we describe the syntax and semantics in detail along with policy examples.

3.1 Fundamental Design Choices

The goal is to design a simple but expressive access control policy language. To achieve this, we beforehand introduce the important design decisions for JACPoL as below.

First, JACPoL is JSON-formatted [21].

Second, JACPoL is attribute-based by design but meanwhile supports RBAC [14]. When integrating RBAC, user roles are considered as an attribute (ARBAC) [23], or attributes are used to constrain user permissions (RABAC) [24], which obtains the advantages of RBAC while maintaining ABAC's flexibility and expressiveness.

Third, JACPoL adopts hierarchically nested structures similar to XACML. The layered architecture as shown in Figure 1 not only enables scalable and fine-grained access control, but also eases the work of policy definition and management for policy designers.

Forth, JACPoL supports Implicit Logic Operators which make use of JSON built-in data structures (Object and Array) to implicitly denote logic operations. This allows a policy designer to express complex operations without explicitly using logical operators, and makes JACPoL policies greatly reduced in size and easier to read and write by humans.

Fifth, JACPoL supports Obligations to offer a rich set of security and network management features.

3.2 Policy Structure

JACPoL uses hierarchical structures very similar to the XACML standard [22]. As shown in Figure 1, JACPoL policies are structured as Policy Sets that consist of one or more child policy sets or policies, and a Policy is composed of a set of Rules.

Because not all Rules, Policies, or Policy Sets are relevant to a given request, JACPoL includes the notion of a Target. A Target determines whether a Rule/Policy/Policy Set is applicable to a request by setting constraints on attributes using simple Boolean expressions. A Policy Set is said to be Applicable if the access request satisfies the Target, and if so, then its child Policies are evaluated and the results returned by those child policies are combined using the policycombining algorithm; otherwise, the Policy Set is skipped without further examining its child policies and returns a Not Applicable decision. Likewise, the Target of a Policy or a Rule has similar semantics.

The Rule is the fundamental unit that is evaluated eventually and can generate a conclusive decision (Permit or Deny specified in its Effect field). The Condition field in a rule is a simple or complex Boolean expression that refines the applicability of the rule beyond the predicates specified by its target, and is optional. If a request satisfies both the Target and Condition of a rule, then the rule is applicable to the request and its Effect is returned as its decision; otherwise, Not Applicable is returned.

For each Rule, Policy, or Policy Set, an id field is used to be uniquely identified, and an Obligation field is used to specify the operations which should be performed (typically by a PEP) before or after granting or denying an access request, while a Priority is specified for conflict resolution between different Rules, Policies, or Policy Sets.

3.3 Syntax and Conventions

JACPoL uses JSON syntax to construct and validate its policies. A policy must follow correct JSON syntax to take effect. In this subsection, we do not provide a complete description of what constitutes valid JSON. However, below is a list of fundamental characteristics of JSON:

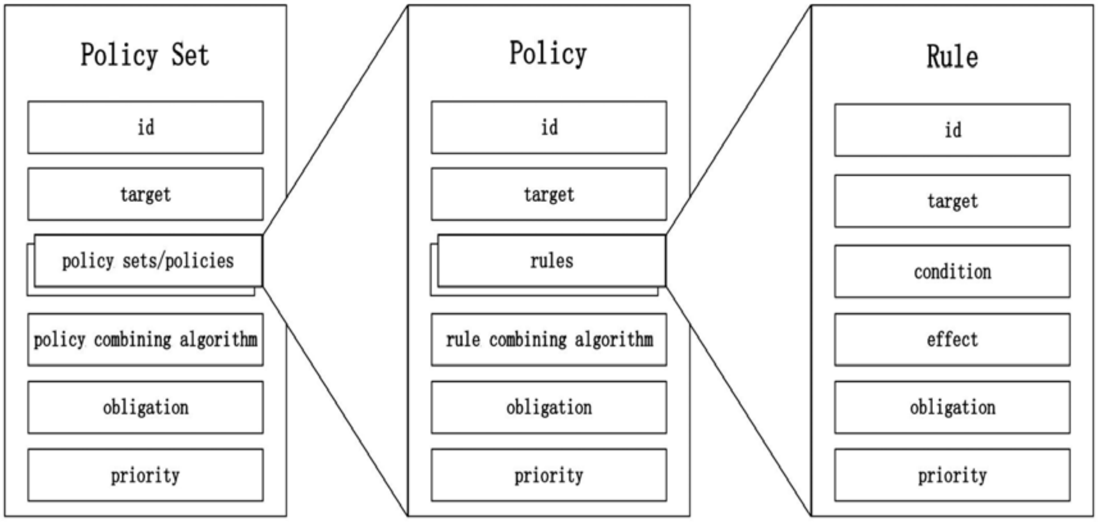


Figure 1. JACPoL's hierarchical nested structure

- JSON is built on two universal data structures: object and array.
- An object is denoted by braces ({}) that can hold multiple name-value pairs. For each name-value pair, a colon (:) is used to separate the name and the value, whilst multiple name-value pairs are separated by comma (,) as in the following example: {"id": 1, "effect": "permit"}.
- An array is denoted by brackets ([ ]) that can hold multiple values separated by commas (,) as in the following example: ["Monday", "Friday", "Sunday"].
- A value can be a string in double quotes, or a number, or a Boolean value (true or false), or null, or an object or an array.
- Whitespace can be inserted between any pair of JSON tokens ({ } [ ] " , :).

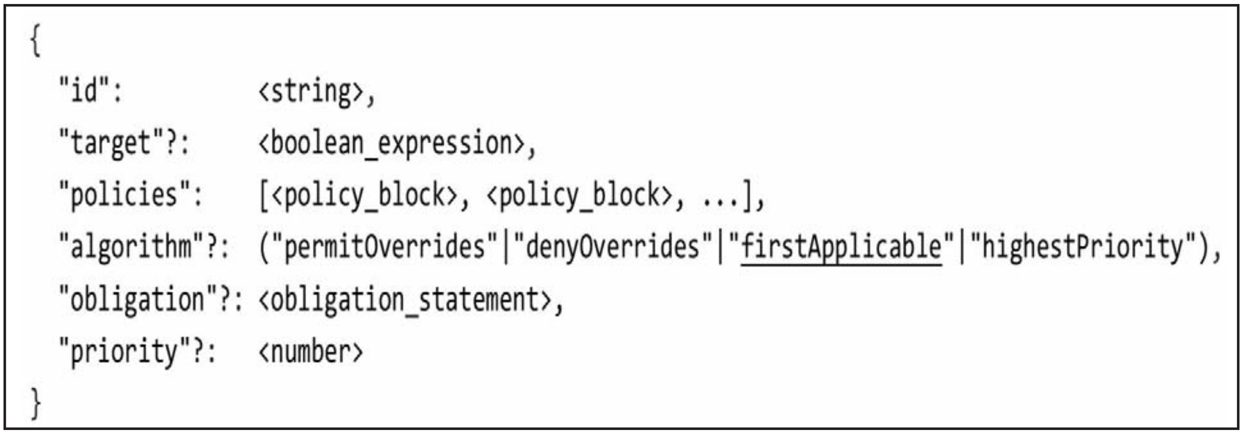
In the subsequent subsections, we will elaborate on the syntax and semantics for each policy element. To illustrate better, the following conventions are used:

- The following characters are special characters used in the description of the grammar and are not included in the policy syntax: < > ... ( ) |.
- If an element allows multiple values, it is indicated using the repeated values, commas, and an ellipsis (...). Example: [<rule_block>, <rule_block>, ...].
- A question mark (?) following an element indicates that element is optional. Example: {"condition"?: <boolean_expression>}.
- A vertical line (|) between elements indicates alternatives. Parentheses define the scope of the alternatives. The default value is underlined if the field is optional, Example: {"algorithm"?: ("permitOverrides" | "firstApplicable")}.
- Elements that must be literal strings are enclosed in double quotation marks.

3.4 Policy Sets, Policies and Rules

This subsection describes the grammar of the Policy Set, the Policy and the Rule. In JACPoL, a policy set, a policy, or a rule always starts and ends with a brace, which denotes a policy set block, a policy block, or a rule block.

Policy Set Block. Figure 2 describes the grammar of the policy set block, which is composed of six name-value pairs that exactly correspond to the six elements of a policy set. As shown in the figure, the "id" field is a string which can be either numeric or descriptive to uniquely identify a policy set. The "target" specifies a Boolean expression indicating the resources, subjects, actions or the environment attributes to which the policy set is applied. The "policies" stores a list of policy blocks with each one corresponding to a policy. The “algorithm” field specifies the name of a decision-combining algorithm to compute the final decision according to the results returned by its child policies. The “obligation” specifies actions to take in case a particular conclusive decision (Permit or Deny) is reached. The “priority” provides a numeric value indicating the weight of the policy set when its decision conflicts with other policy sets under the highest Priority algorithm.

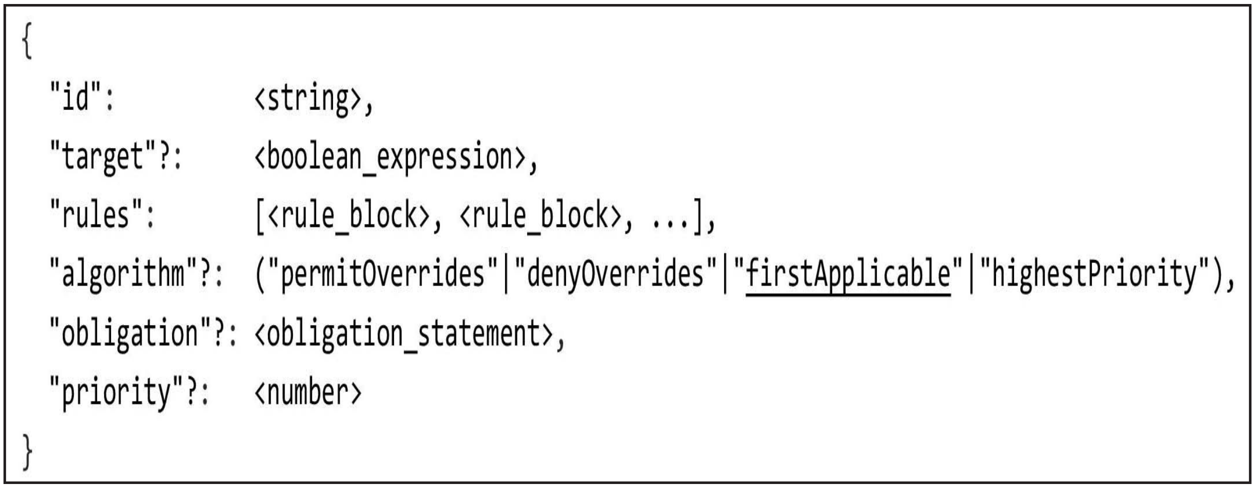


```
{
  "id":          <string>,
  "target"?:     <boolean_expression>,
  "policies":    [<policy_block>, <policy_block>, ...],
  "algorithm"?:  ("permitOverrides"|"denyOverrides"|"firstApplicable"|"highestPriority"),
  "obligation"?: <obligation_statement>,
  "priority"?:   <number>
}
```

Figure 2. Grammar of the policy set block

Note that elements like target, algorithm, obligation and priority are optional and, if omitted, the predefined default values would be taken (e.g., target: true, algorithm: first Applicable, obligation: null, priority: 0.5).

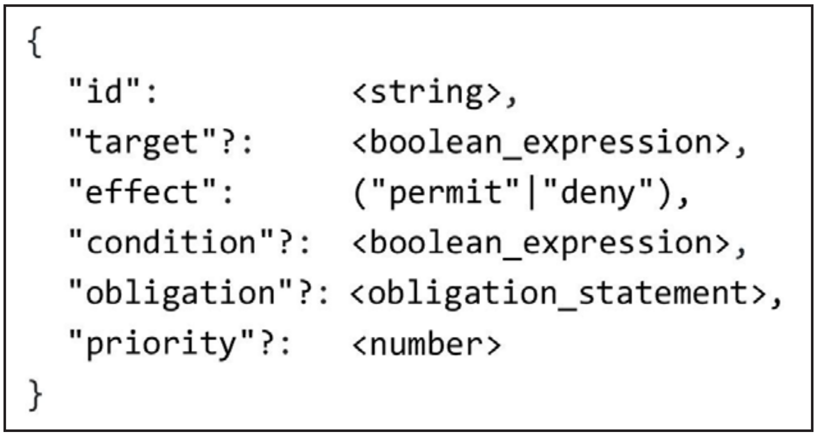
Policy Block. As shown in Figure 3, a policy block contains an id, a target, an algorithm, an obligation and a priority similar to a policy set. The difference is, it has a “rules” list that holds one or more rule blocks instead of policy blocks.



```
{
  "id":          <string>,
  "target"?:     <boolean_expression>,
  "rules":       [<rule_block>, <rule_block>, ...],
  "algorithm"?:  ("permitOverrides"|"denyOverrides"|"firstApplicable"|"highestPriority"),
  "obligation"?: <obligation_statement>,
  "priority"?:   <number>
}
```

Figure 3. Grammar of the policy block

Rule Block. Figure 4 describes the grammar of the rule block. Unlike a policy set block or a policy block, a rule block does not contain any leaf nodes like child policies or child rules and thus a decision-combining algorithm field is not needed either. Instead, it possesses a “condition” element that specifies the condition for applying the rule, and an “effect” element that, if the rule is applied, would be the returned decision of the rule as either Permit or Deny. In comparison to a target, a condition is typically more complex and often includes functions (e.g., “greater-than”) for the comparison of attribute values, and logic operations (e.g., “and”, “or”) for the combination of multiple conditions. If either the target or the condition is not satisfied, a Not Applicable would be taken as the result instead of the specified effect. Note that the Condition is by default true if omitted.



```
{
  "id":          <string>,
  "target"?:     <boolean_expression>,
  "effect":      ("permit"|"deny"),
  "condition"?:  <boolean_expression>,
  "obligation"?: <obligation_statement>,
  "priority"?:   <number>
}
```

Figure 4. Grammar of the rule block

3.5 Targets and Conditions

As aforementioned, a Target or a *Condition* is a Boolean expression specifying constraints on attributes such as the subject, the resource, the environment, and the action of requests. The Boolean expression of a *Target* is often simple and very likely to be just a test of string equality, but that of a condition can be sometimes complex with constraints on multiple attributes (*attribute conditions*).

Attribute Condition is a simple Boolean expression that consists of a key-value pair as shown below:

```
{“<attribute_expression>”: <condition_expression>}
```

The key is an *attribute expression* in string format that specifies an attribute or a particular computation between a set of attributes; the value is a *condition expression*, which is a JSON block composed of one or more operator-parameter pairs specifying specifically the requirements that the attribute expression needs to meet. The simplest format of an attribute condition is to verify the equality /inequality between the attribute (e.g., time) and the parameter (e.g., 10:00:00) using comparative operators (e.g., greater-than, less-than, equal-to, etc.):

```
{“<attribute>”: {“<comparative_operator>”: <parameter>}}
```

However, there are also cases where we have multiple constraints (operator-parameter pairs) on the same attribute, connected by logical relations like *AND, OR, NOT,* which are respectively denoted by the keywords *allOf* , *anyOf and not*.

Logical Operators. JACPoL uses logical operators in a form of constructing key-value pairs. The logical operator is the key and, depending on the number of arguments, *allOf* and *anyOf* operators are to be followed by an *array* ([ ]) of multiple constraints, while the not operator is to be followed by an *object* ({ }). An *allOf* operation would be evaluated to true only if all subsequently included constraints are evaluated to true, but an *anyOf* operation would be true as long as there is at least one of the constraints which is true. An *not* operation would be true if the followed constraint is evaluated to false. Logical operators can be nested to construct logical relations such as n*ot any of, not all of*. For example, an *attribute* condition containing multiple constraints with nested logical operators as below:

```
{“sumOf x y”: {“not”: {“anyOf”: [
                 {“between”: “j k”},
                 {“equals”: “z”}]}}}
```

In which the string “sumOf x y” is an attribute expression. The keyword *sumOf* defines a function to compute the sum of attributes *x* and *y*, which is to be evaluated by the subsequent condition expression. Please note that a parameter like *j*, *k* or *z* can also possibly be another attribute instead of an explicit value.

In addition, logical operators can also be used to combine multiple attribute conditions in order to express complex constraints on more than one attributes easily and flexibly. As an example, the condition below expresses constraints on two attributes and would be evaluated to true only when both (*allOf* the two) constraints are met:

```
{“allOf”: [<attribute_condition>, <attribute_condition>]}
```

Less is More: Implicit Logical Operators. A complex condition might contain many logical operators which make the policy wordy and hard to read. To overcome this, we make use of JSON's built-in data structures, *object* and *array*, and define following implicit logical operators as alternatives to *allOf* and *anyOf* :

• An *object* is implicitly an *allOf* operator which would be evaluated to true only if all the included key-value pairs are evaluated to true.

• An array is implicitly an *anyOf* operator which would be evaluated to true as long as at least one of its elements is evaluated to true.

For example, below is a condition statement using implicit logical operators to verify if it is working hour.

```
{
“time”:{“between”: [“09:00 12:00”, “14:00 18:00"]},
“weekday”: {“not”: {“equals”: [“saturday”, “sunday“]}}
}
```

As a comparison, below is for the same verification with explicit logical operators.

```
{
“allOf”: [{
      “time”: {“anyOf”: [
            {“between”: “09:00 12:00”},
            {“between”: “13:00 18:00”}]}
   },{
      “weekday”: {“not”: {“anyOf”: [
            {“equals”: “saturday”},
            {“equals”: “sunday”}]}}}]
}
```

Apparently, implicit operators save a lot size and make policies more readable, which has later turned out to be very useful in our policy engine implementation.

3.6 Combining Algorithms

In JACPoL, policies or rules may conflict and produce totally different decisions for the same request. JACPoL resolves this by adopting four kinds of decision-combining algorithms: *Permit-Overrides*, *Deny-Overrides*, *First-Applicable*, and *Highest-Priority*. Each algorithm represents a different way for combining multiple local decisions into a single global decision:

• *Permit-Overrides* returns *Permit* if any decision evaluates to *Permit*; and returns Deny if all decisions evaluate to *Deny*.

• *Deny-Overrides* returns *Deny* if any decision evaluates to *Deny*; returns *Permit* if all decisions evaluate to *Permit*.

• *First-Applicable* returns the first decision that evaluates to either of *Permit* or *Deny*. This is very useful to shortcut policy evaluation.

• *Highest-Priority* returns the highest priority decision that evaluates to either of *Permit* or *Deny*. If there are multiple equally highest priority decisions that conflict, then *deny-overrides* algorithm would be applied among those highest priority decisions.

Please note that for all of these combining algorithms, Not Applicable is returned if not any of the child rules (or policies) is applicable. Hence, the set of possible decisions is 3-valued.

3.7 Obligations

JACPoL includes the notion of obligation. An *Obligation* optionally specified in a Rule, a Policy or a PolicySet is an operation that should be performed by the PEP in conjunction with the enforcement of an authorization decision. It can be triggered on either Permit or Deny. We employ the format as below to express obligations in JACPoL:

```
{“<decision>”: {“<operation>”: [<parameter>, <parameter>, ...]}}
```

For example, the obligation below is for the access control of a document.

```
{
 “permit”: {“watermark”: [“DRAFT”]},
 “deny”: {
      “feedback”: [“ACCESS DENIED”],
      “notify”: [“admin@gmail.com”, “hr@gmail.com”]
}
}
```

It specifies that if an access request is denied, the user would be informed with an access denied message and the administrator and HR would also be notified; if an access request is approved, watermark the document “DRAFT” before delivery. It worth’s to be mentioned that the referred obligations have an eminently locale nature in the sense that their execution is the exclusive prerogative of the PEP which can possibly rely on the information available in the PIP [2]. More general and distributed obligations are currently under investigation.

3.8 Implementation

We implemented JACPoL in a Javascript/Node.js/Redis based policy engine [1] which is available on Github. As shown in Figure 5, the policy engine employs the classical PDP/PEP architecture [2]. The PDP retrieves policies from the PRP (Policy Retrieval Point), and evaluates authorization requests from the PEP by examining the finite relevant attributes against the policies. If more attributes are required to reach a decision, the PDP will request the PIP(Policy Information Point) as an external information source. The latter may also be requested in the case of obligations.

The non-blocking nature of Node.js allows the system to provide an efficient and scalable access control, and a Redis server

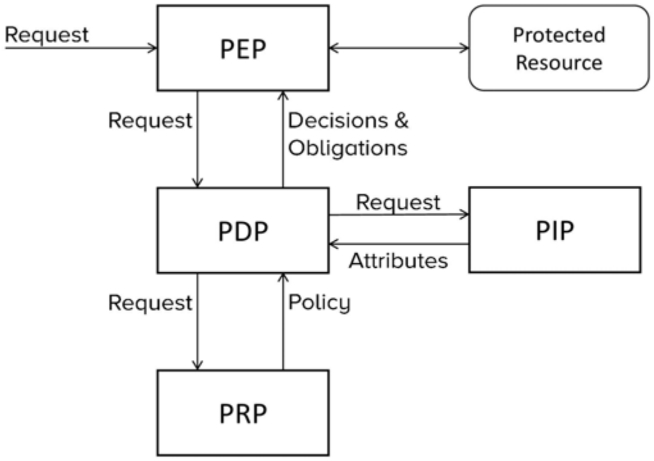
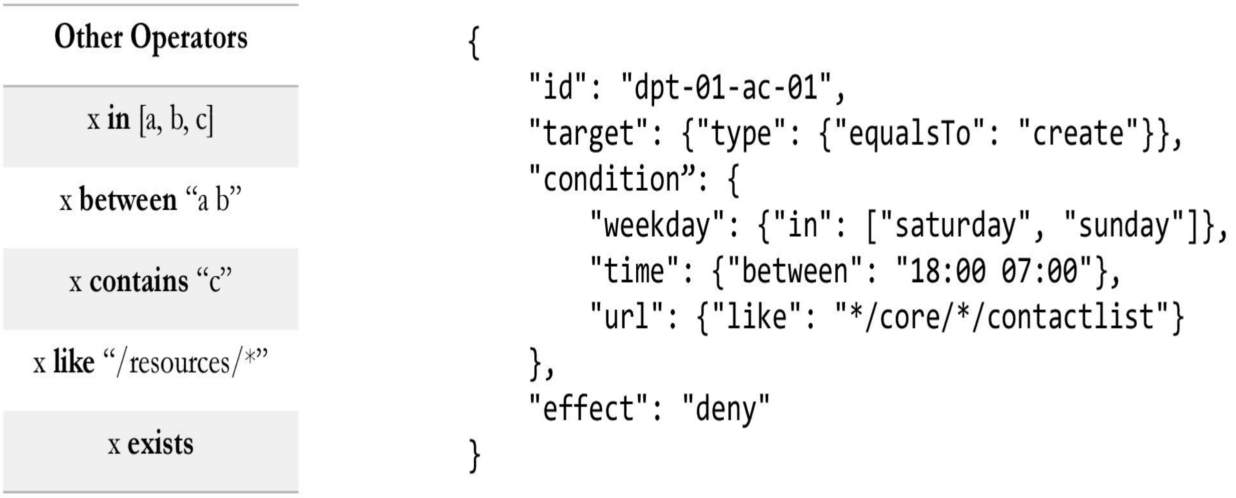


Figure 5. Policy engine architecture

was employed to enable flexible and high-performance data persistence and caching. In order to validate its functionality, we have deployed this policy engine on a messaging node of an European project [1]. This policy engine adopted the ABAC model and customized the vocabulary of JACPoL for the requirements of the European project framework. With JACPoL based lightweight policies, Node.js based nonblocking I/O, and Redis based fast caching, it provided a highly performant access control according to various tailored attributes in an expressive and flexible way. In addition to comparative operators like greaterThan, lessThan, equalsTo, JACPoL is extended to support more operators as listed in the left of Figure 6 with a rule example in the right:



Other Operators
x **in** [a, b, c]
x **between** “a b”
x **contains** “c”
x **like** “/resources/*”
x **exists**

```
{
    "id": "dpt-01-ac-01",
    "target": {"type": {"equalsTo": "create"}},
    "condition”: {
        "weekday": {"in": ["saturday", "sunday"]},
        "time": {"between": "18:00 07:00"},
        "url": {"like": "*/core/*/contactlist"}
    },
    "effect": "deny"
}
```

Figure 6. Other supported operators (left) and a rule example (right)

4. Evaluation

This section evaluates JACPoL with a comprehensive comparison to other pre- and post-XACML policy languages, which respectively are JSONPL [19], AWS IAM [20], XACML [8], Ponder [5], Rei [26], XACL [27], KAoS [28], EPAL [6], and ASL [29]. To begine with, we identify following requirements for an access control policy language to meet the increasing needs of security management for today’s ICT systems:

• *Expressiveness* to support wide range of policy needs and be able to specify various complex, advanced policies that a policy maker intend to express [4].

• *Extensibility* to cater for new features or concepts of policy in the future [5].

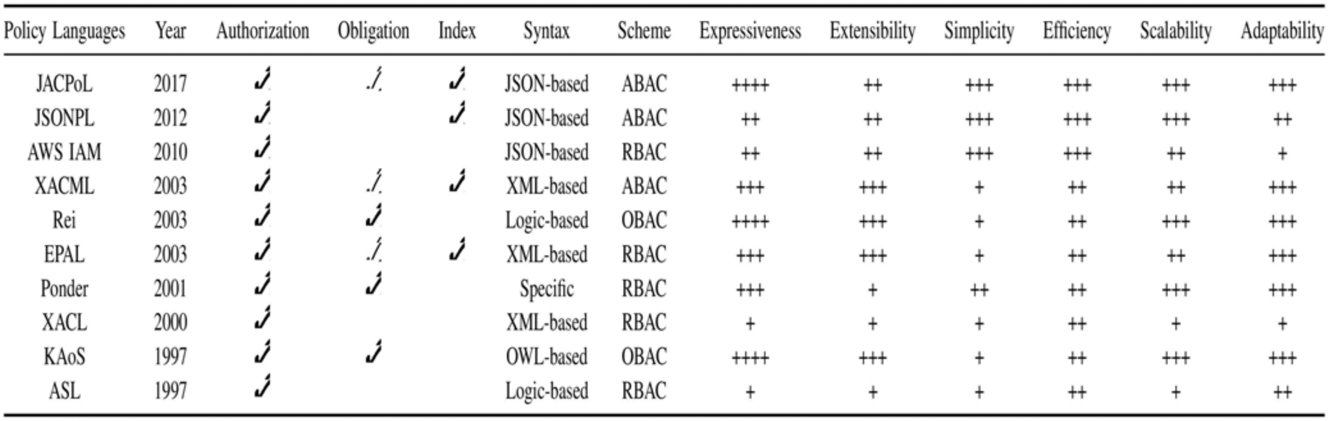
• *Simplicity* to ease the policy definition and management tasks for the policy makers with different levels of expertise. This includes both conciseness and readability to avoid long learning curve and complex training.

• *Efficiency* to ensure the speed for machines to parse the policies defined by humans. This can be affected by policy structure, syntax, and data format.

• *Scalability* to ensure the performance as the network grows in size and complexity. This is important especially in large-scale or multi-domain networks.

• *Adaptability* to be compatible with any access control tasks derived from an ICT system. Any user could directly tailor the enforcement code and related tool set provided by the policy language to their authorization systems.

Table 1 shows the complete evaluation of these policy languages. In the table, ‘✓’ and ‘✓’ respectively indicate ‘support’ and ‘strongly support’, while ‘+’, ‘++’, ‘+++’ and ‘++++’ mean ‘poor’, ‘good’, ‘very good’ and ‘excellent’. The comparison mainly focuses on their design and implementation choices regarding *authorization, obligation, index, syntax and scheme,* and their performance with respect to the six previously defined criteria. Among these features, index refers to whether there exists a special item for policy engine to retrieve the required policies more efficiently.



Policy Languages	Year	Authorization	Obligation	Index	Syntax	Scheme	Expressiveness	Extensibility	Simplicity	Efficiency	Scalability	Adaptability
JACPoL	2017	✓	./.	✓	JSON-based	ABAC	++++	++	+++	+++	+++	+++
JSONPL	2012	✓		✓	JSON-based	ABAC	++	++	+++	+++	+++	++
AWS IAM	2010	✓			JSON-based	RBAC	++	++	+++	+++	++	+
XACML	2003	✓	./.	✓	XML-based	ABAC	+++	+++	+	++	++	+++
Rei	2003	✓	✓		Logic-based	OBAC	++++	+++	+	++	+++	+++
EPAL	2003	✓	./.	✓	XML-based	RBAC	+++	+++	+	++	++	+++
Ponder	2001	✓	✓		Specific	RBAC	+++	+	++	++	+++	+++
XACL	2000	✓			XML-based	RBAC	+	+	+	++	+	+
KAoS	1997	✓	✓		OWL-based	OBAC	++++	+++	+	++	+++	+++
ASL	1997	✓			Logic-based	RBAC	+	+	+	++	+	++

Table 1. Evaluation and comparison between JACPoL and other policy languages

Like many other languages, JACPoL provides support for authorization and obligation capabilities as previously introduced. In addition, it includes a concept of Target within each Policy Set, Policy and Rule to allow efficient policy index. In terms of expressiveness, JACPoL, Rei and KAoS extensively support the specification of constraints, which can be set on numerous attributes in a flexible expression [30].

On the other hand, compared to XML-based languages, JSON-based JACPoL is simpler and more efficient, but meanwhile, we have to admit that JSON is less sophisticated than XML, which accordingly may make JACPoL less extensible. JACPoL is scalable and the reasons are twofold: first, its efficient performance in policy index and evaluation allows it to deal with complex policies under a large-scale network environment; second, its concise semantics and lightweight data representation make it easily replicable and transferable for distributed systems. As for adaptability, compared to other languages, the application specific nature of AWS IAM makes it relatively harder to be adapted to other systems.

Figure 7 and Figure 8 below show the results of policy evaluation performance tests on JACPoL and XACML. We evaluated respectively how both languages' policy processing time increases with the growth of nesting layers (policy depth) and with the growth of sibling rules (policy scale). First we used the two languages to express the same policy set with 400 recursively

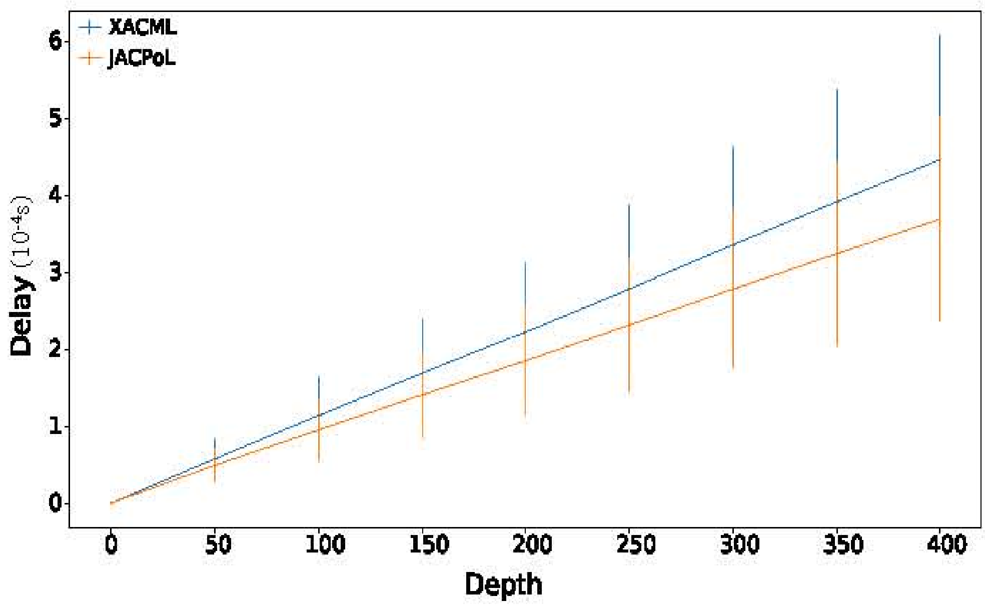


Figure 7. Effect of policy depth

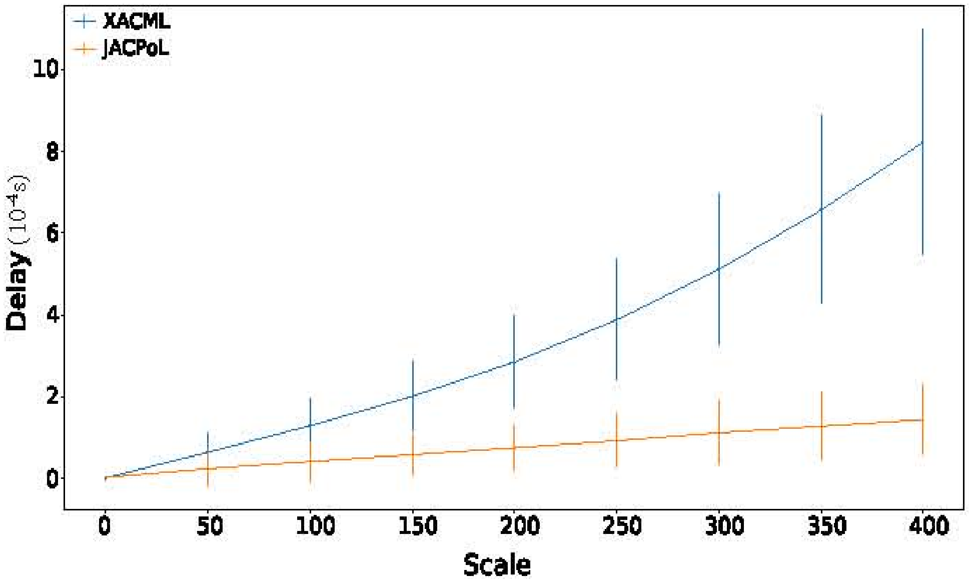


Figure 8. Effect of policy scale

nesting child policies, and recorded the delay when each child policy was evaluated until the deepest one was reached. The result is shown in Figure 7. Then we did the same with a policy set containing 400 sibling child policies and got the result in Figure 8. Each test was repeated 1000 times conducted using Python on a Windows 10 PC with 16G memory and a 2.6GHz Intel core i7-6700HQ processor. From the results we can see that JACPoL is more efficient and scalable and evaluated with less latency than XACML thanks to its JSON syntax and well-defined semantics.

5. Application of JACPOL to Security Models

5.1 RBAC vs ABAC

In policy-based access control systems, a request for access to protected resources is evaluated with respect to a policy that defines which requests are authorized. The policy itself conforms to a security model upstream chosen by the system security administrator because it elegantly copes with the constraints associated with the targeted information system. In the RBAC model, roles are pre-defined and permission sets for resources are pre-assigned to roles. Users are then granted one or more roles in order to receive access to resources [16]. ABAC, on the other hand, relies on user attributes for access decisions. These include: subject attributes, which are attributes concerning the actor being evaluated; object attributes, which are attributes of the resource being affected; action attributes, which are attributes about the operation being executed; and environment attributes, which provides other contextual information such as time of the day, etc. [15]. Generally speaking, RBAC is simple, static and auditable, but is not expressive nor context-aware, while ABAC, by contrast, provides fine-grained, flexible and dynamic access control in real time but is complex and unauditable. Combing these two models judiciously to integrate their advantages thus becomes an essential work in recent research [16] [17] [24] [25].

5.2 Attribute-centric RBAC Application

JACPoL can be implemented to express permission specification policies (PSP) in an attribute-centric RBAC model. For example, Fig. 9 defines a policy set with each policy specifying permissions that are associated to the targeted role. When evaluating a request, the PDP first retrieves all the roles (e.g., from the PIP) that are pre-assigned to the requester, and then examines the permission policies that are associated to these roles to reach a decision. Unlike other traditional statically defined RBAC permissions, JACPoL allows its permissions to be expressed in a quite dynamic and flexible way similar to ABAC. The role attribute is placed as the target for the topmost level of policies to allow easy view of user permissions.

5.3 Role-centric ABAC Application

JACPoL can also be used to implement a language for permission filtering policies (PFP) in a role-centric ABAC model [24]. Similar to the ABAC model in Fig. 5, but in addition to external attributes, the PDP also relies on the PIP to get role permission

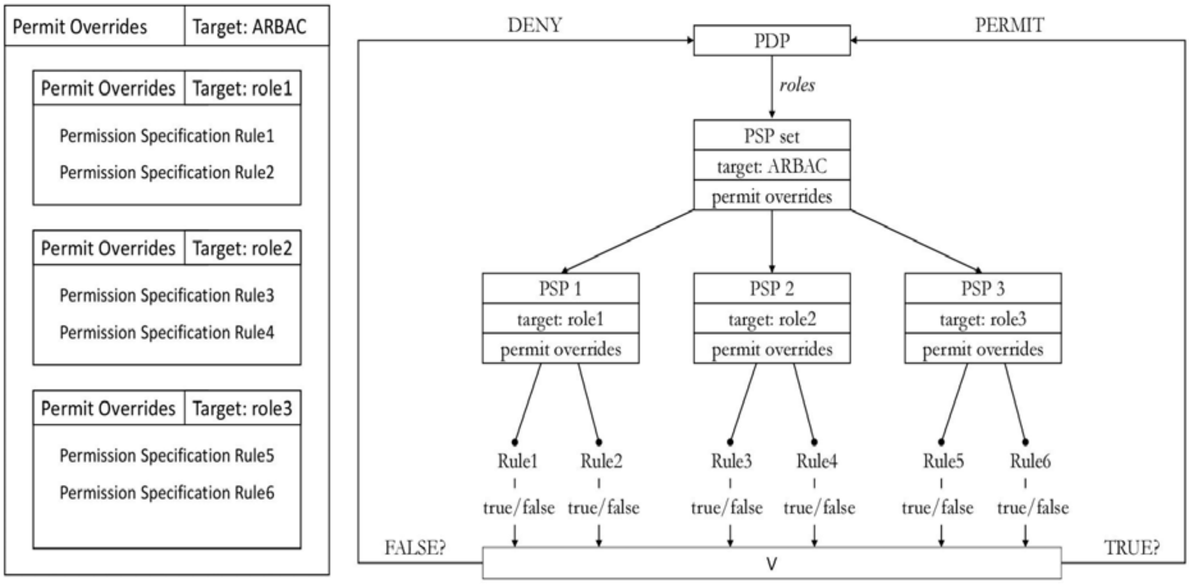


Figure 9. An example ARBAC permission specification policy and its structure tree

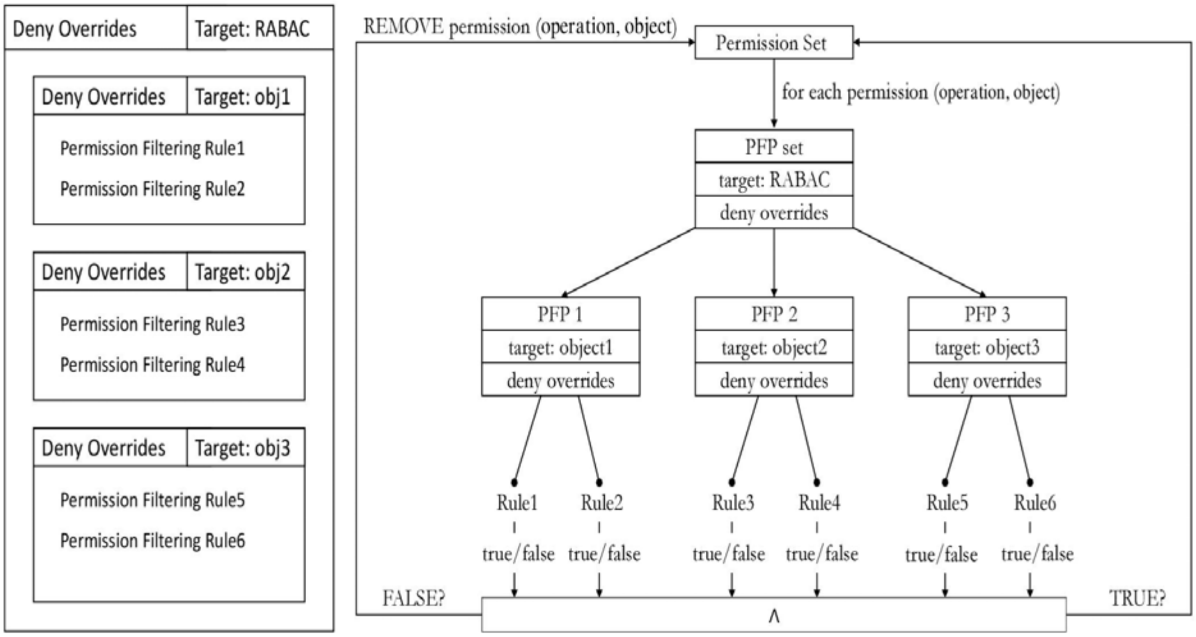


Figure 10. An example RABAC permission specification policy and its structure tree

sets which, as defined by NIST RBAC model, specify the maximum set of available permissions that users can have. These permission sets can be further constrained by the filtering policies based on JACPoL, as shown in Figure 10. Note that this time the target of each PFP maps each object to a subset of the filtering rules. At the same time, the target and condition of each rule determine whether or not the rule is applicable. The applicable filtering rules are invoked one by one against each of the permissions in the permission set. If any of the rules return FALSE, the permission is then blocked and removed from the available permission set for the current session. At the end of this process, the final available permission set available to users therefore will be the intersection of P and R, where P is the set of permissions assigned to the subject’s active roles and R is the set of permissions specified by the applicable JACPoL rules [25].

6. Conclusion

Traditionally, performance has not ever been a major focus in the design of access control systems. Applications are emerging, however, that require policies to be evaluated with a very low latency and high throughput. Under this background, we designed and implemented JACPoL, a fast JSON-based, attribute-centric and light-weight access control policy language. JACPoL provides a good solution for policy specification and evaluation in such applications with low processing delay. We evaluated our policy language with respect to a set of representative criteria in comparison with other existing policy languages. The evaluation showed that JACPoL can be as expressive as XACML but more simple, scalable and efficient. On the other hand, JACPoL leaves room for future improvements in many areas. Obligation capabilities can be further enhanced and delegation support can be formally introduced with well-defined semantics. After that, a more comprehensive experimental performance evaluation would need to be conducted with extensive policy sets for various use cases. Our future work would also include the development of a JACPoL based natural language policy specification and semantic validation toolkit.

Acknowledgment

This work has received funding from the European Unions Horizon 2020 research and innovation program. We gratefully acknowledge support from our colleagues, who made this work possible with their invaluable suggestions.

References

[2] Yavatkar, R., Pendarakis, D., Guerin, R. (2000). A Framework for Policy-based Admission Control. IETF, RFC 2753. (January).

[3] Borders, K., Zhao, X., Prakash, A. (2005). CPOL: High-performance policy evaluation. *In*: The 12th ACM conference on Computer and Communications Security. ACM.

[4] He, L., Qiu, X., Wang, Y., Gao, T. (2013). Design of policy language expression in SIoT, *In:* Wireless and Optical Communication Conference, 321-326. IEEE.

[5] Damianou, N., Dulay, N., Lupu, E., Sloman, M. (2001). The ponder policy specification language. *In*: Policies for Distributed Systems and Networks, *p*. 18-38. Springer Berlin Heidelberg.

[6] Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M. (2003). Enterprise privacy authorization language (EPAL). IBM Research. (March)

[7] Bhatti, R., Ghafoor, A., Bertino, E., Joshi, J, B. (2005). X-GTRBAC: an XMLbased policy specification framework and architecture for enterprise-wide access control. *ACM Transactions on Information and System Security* (TISSEC) (8.2) 187-227.

[8] OASIS XACML Technical Committee: "eXtensible access control markup language (XACML) Version 3.0. Oasis Standard, OASIS (2013). URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-specos- en.html, last accessed 2017/05/17.

[9] Crampton, J., Morisset, C. (2012). PTaCL: A language for attribute-based access control in open systems. *In:* International Conference on Principles of Security and Trust, p. 390-409. Springer Berlin Heidelberg.

[10] Crockford, D. (2017). JSON — The fat-free alternative to XML (Vol. 2006). URL: http://www.json.org/fatfree.html. last accessed 2017/05/17.

[11] El-Aziz, A. A., Kannan, A. (2014). JSON encryption. *Computer Communication and Informatics (ICCCI)*, 2014 International Conference on. IEEE.

[12] Griffin, L., Butler, B., Leastar, de., E., Jennings, B., Botvich, D. (2012). On the performance of access control policy evaluation. *In:* Policies for Distributed Systems and Networks (POLICY), 2012 IEEE International Symposium on, p. 25-32. IEEE .

[13] W3schools: JSON vs XML. URL: www.w3schools.com/js/js_json_xml.asp, last accessed 2017/05/24.

[14] Ferraiolo, D. F., Kuhn, D. R. (2009). Role-based Access Controls." arXiv preprint arXiv: 0903.2171. (March).

[15] Hu, V, C., Ferraiolo, D., Kuhn, R. (2013). Guide to attribute based access control (ABAC) definition and considerations. NIST special publication 800.162.

[16] Empower ID (2013). Best practices in enterprise authorization: The RBAC/ABAC hybrid approach. Empower ID, White paper.

[17] Coyne, E., Weil, T. R. (2013). ABAC and RBAC: scalable, flexible, and auditable access management. *IT Professional* 15 (3) 0014-16.

[18] Brossard, David. (2014). JSON Profile of XACML 3.0 Version 1.0. XACML Committee Specification 01, (December). URL: http://docs.oasis-open.org/xacml/xacml-json-http/v1.0/cs01/xacml-jsonhttp- v1.0-cs01.pdf, last accessed 2017-05-26.

[19] Steven, D., Bernard, B., Leigh, G. (2013). JSON-encoded ABAC (XACML) policies. FAME project of Waterford Institute of Technology. Presentation to OASIS XACML TC concerning JSON-encoded XACML policies, 2013-05-30.

[20] Amazon Web Services: AWS Identity and Access Management(IAM) User Guide. URL: http://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html, last accessed 2017-05-27.

[21] ECMA International: ECMA-404 The JSON Data Interchange Standard. URL: http://www.json.org/, last accessed 2017-05-27.

[22] Ferraiolo, David. (2016). Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC) *In*: Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control. ACM.

[23] Obrsta, L., McCandlessb, D., Ferrella, D. (2012). Fast semantic Attribute-Role- Based Access Control (ARBAC) in a collaborative environment, *In*: 2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), Pittsburgh, PA, USA, 14-17 (October).

[24] Jin, X., Sandhu, R., Krishnan, R. (2012). RABAC: role-centric attribute-based access control, *In*: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. Springer Berlin Heidelberg.

[25] Kuhn, D. R., Coyne, E J., Weil, T. R. (2010). Adding attributes to role-based access control. *Computer* 43 (6) 79-81.

[26] Kagal, L., Finin, T., Joshi, A. (2003). A policy language for a pervasive computing environment. *Policies for Distributed Systems and Networks*, 2003. *In*: Proceedings. POLICY 2003. IEEE 4th International Workshop on. IEEE.

[27] Hada, S., Kudo, M. (2000). XML Access Control Language: provisional authorization for XML documents.

[28] Uszok, A., Bradshaw, J. M., Jeffers, R. (2004). Kaos: A policy and domain services framework for grid computing and semantic web services, *In*: International Conference on Trust Management. Springer Berlin Heidelberg.

[29] Jajodia, S., Samarati, P., Subrahmanian, V. S. (1997). A logical language for expressing authorizations, *In*: Proceedings of IEEE Symposium on Security and Privacy, IEEE.

[30] Neuhaus, C., Polze, A., Chowdhuryy, M. M. (2011). Survey on healthcare IT systems: standards, regulations and security. No. 45. Universitätsverlag Potsdam.