Chapter 4
A JSON-Based Fast and Expressive Access Control Policy Framework

Hao Jiang
New H3C Technologies Co. Ltd., China

Ahmed Bouabdallah
IMT Atlantique, France

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. To achieve this, the policy language needs to be very expressive but lightweight and efficient. These challenges are investigated and a set of key requirements for such a policy language is identified. JACPoL is accordingly introduced as a descriptive, scalable, and expressive policy language in JSON. JACPoL by design provides a flexible and fine-grained ABAC style (attribute-based access control) while it can be easily tailored to express other access control models. The design and implementation of JACPoL are illustrated together with its evaluation 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. The performance evaluation shows that JACPoL requires much less processing time and memory space than XACML.

INTRODUCTION

Policies represent sets of properties of information processing systems (Clarkson&Schneider, 2010). Their implementation mainly rests on the IETF architecture (Yavatkar et al., 2000) initially introduced to manage QoS policies in networks. It consists in two main entities namely the PDP (Policy Decision Point) and the PEP (Policy Enforcement Point). The first one which is the smart part of the architecture acts as a controller the goal of which consists in handling and interpreting policy events, and deciding in

DOI: 10.4018/978-1-5225-8446-9.ch004

accordance with the policy currently applicable, what action should be taken. The decision is transmitted to the PEP which has to concretely carry it out.

Access control policies are a specific kind of security policies aiming to control the actions that principals can perform on resources by permitting their access only to the authorized ones. Typically, the access requests are intercepted and analyzed by the PEP, which then transfers the request details to the PDP for evaluation and authorization decision (Yavatkar et al., 2000). 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 (Borders et al., 2005).

In this work, the authors 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. The authors 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, the authors propose a simple but expressive access control policy language (JACPoL) based on JSON. A PoC (Proof of Concept) has been conducted through the implementation of JACPoL in a policy engine operated in reTHINK testbed (reTHINK Project Testbed, 2016)). At last JACPoL is carefully positioned in comparison with 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. The authors identify key requirements and technical trends for future policy languages. They 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. Quantitative evaluations of JACPoL by comparison to XACML show that JACPoL systematically requires much less time and memory space than XACML.The authors also elaborate on the applicability of JACPoL on ABAC model, RBAC model and their combinations or their by-products. Last but not least, their implementation leads to a novel and performant policy engine adopting the PDP/PEP architecture (Yavatkar et al., 2000) and JACPoL policy language based on Node.js[1] and Redis[2].

The remainder of this chapter is structured as follows. In Section 2, the problematic is refined by delimiting precisely its perimeter. Section 3 provides an illustration in depth with representative policy examples of the design of the policy language in terms of the constructs, semantics and other important features like Implicit Logic Operators, combining algorithms and implementation. Section 4 further qualitatively evaluates JACPoL and compares it with other existing access control policy specification languages. In section 5 a detailed performance evaluation is given. The ABAC-native nature of JACPoL is detailed in section 6 along with a comprehensive discussion on other possible application of JACPoL to ARBAC (Attribute-centric RBAC) and RABAC (Role-centric ABAC) security models. In Section 7 the authors summarize their work and discuss future research directions.

PROBLEM STATEMENT

In the past decades, a lot of policy languages have been proposed for the specification of access control policies using XML, such as EPAL (Ashley et al., 2003), X-GTRBAC (Bhatti et al., 2005) and the standardized XACML (OASIS, 2013). Nevertheless, it is generally acknowledged that XACML suffers from providing poorly defined and counterintuitive semantics (Crampton & Morisset, 2012), 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 favoured by more and more nowadays' policy designers (Crockford, 2006; El-Aziz & Kannan, 2014; Griffin, 2012; W3schools).

To address the aforementioned inefficiency issues of the XML format, the XACML Technical Committee recently designed the JSON profile (Brossard, 2014) 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 (Griffin, 2012) 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 it can be understood, 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 (Steven, 2013).

Major service providers such as Amazon Web Services (AWS) (Amazon Web Services) 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 the authors' 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.

JACPoL DETAILED DESIGN

This section presents JACPoL in depth. The foundations of JACPoL are first recalled, and then its structures is introduced with an overview of how an access request is evaluated with respect to JACPoL policies. After that, the syntax and semantics is described in detail along with policy examples.

Fundamental Design Choices

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

First, JACPoL is JSON-formatted (ECMA).

Second, JACPoL is attribute-based by design but meanwhile supports RBAC (Ferraiolo & Kuhn, 2009). When integrating RBAC, user roles are considered as attributes (ARBAC) (Obrsta, 2012), or attributes are used to constrain user permissions (RABAC) (Jin et al., 2012), 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.

Policy Structure

JACPoL uses hierarchical structures very similar to the XACML standard (Ferraiolo et al., 2016). 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 policy-combining 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.

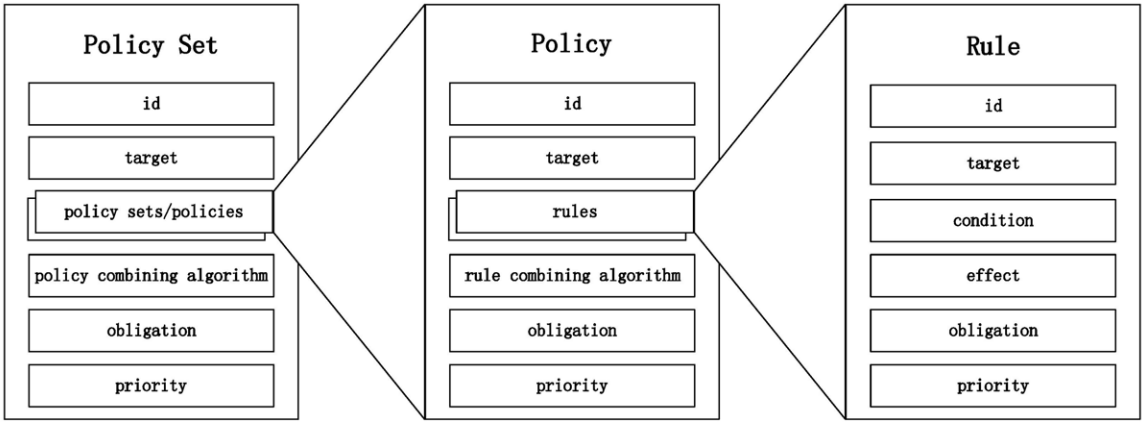
For each Rule, Policy, or Policy Set, an id field is used to be uniquely identified, and an Obligation field is used to specify the operations which should be performed (typically by a PEP) before or after granting or denying an access request, while a Priority is specified for conflict resolution between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

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 a list of fundamental characteristics of JSON is recalled; a more detailled treatment of JSON can be found in (EMA):

- JSON is built on two universal data structures: object and array.

Figure 1. JACPoL's hierarchical nested structure



- 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, the syntax and semantics for each policy element are elaborated. 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 this 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.

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 highestPriority algorithm.

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: *firstApplicable,* 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 concerns the fact that it has a "rules" list that holds one or more rule blocks instead of policy blocks.
- **Rule Block**: Figure n4 describes the grammar of the rule block. Unlike a policy set block or a policy block, a rule block does not contain 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 con-dition 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.

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

Figure 2. Grammar of the policy set block

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

Figure 3. Grammar of the policy block

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

Figure 4. Grammar of the rule block

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

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 occur 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 Operator:** 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 not 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 addition, logical operators can also be used to combine multiple attribute conditions in order to express complex constraints on more than one attribute, 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, the JSON's built-in data structures, *object* and *array*, allow to 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 op-
erators.
{
        "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 of space and make policies more readable, which has later turned out to be very useful in the policy engine implementation.

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.

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. The format below is introduced 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"]
        }
}
```

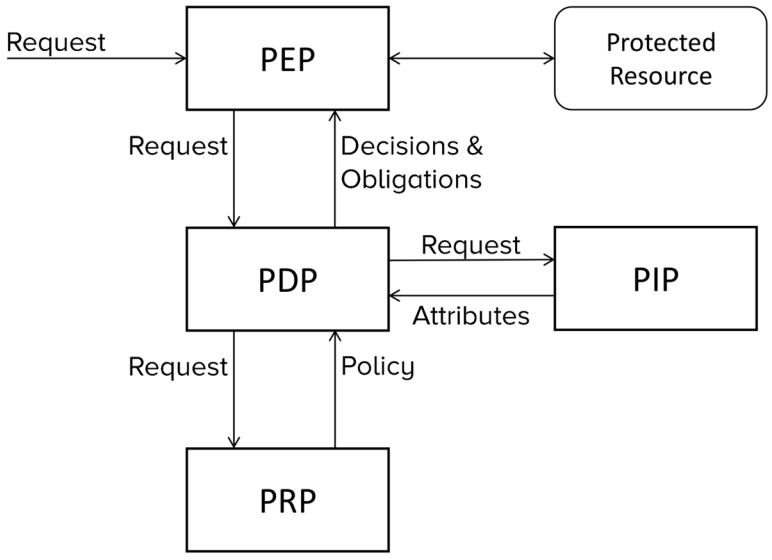
It specifies that if an access request is denied, the user would be informed with an access denied message and the administrator and HR would also be notified; if an access request is approved, watermark the document "DRAFT" before delivery. It is worth mentioning that the referred obligations have an 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 (Yavatkar et al., 2000). More general and distributed obligations deserve a dedicated investigation.

Implementation

An implementation of JACPoL has been done in a Javascript/Node.js/Redis based policy engine (reTHINK CSP Policy Engine, 2016) which is available on Github. As shown in Figure 5, the policy engine employs the classical PDP/PEP architecture (Yavatkar et al., 2000). 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 was employed to enable flexible and high-performance data persistence and caching. In order to validate its functionality, this policy engine has been deployed on a messaging node

Figure 5. Policy engine architecture



of reTHINK project (reTHINK Project, 2016). This reTHINK policy engine (reTHINK CSP Policy Engine, 2016) adopted the ABAC model and customized the vocabulary of JACPoL for the requirements of reTHINK framework. With JACPoL based lightweight policies, Node.js based non-blocking 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 (reTHINK Deliverable 6.4, 2016). In reTHINK, in addition to comparative operators greaterThan, lessThan, equalsTo, JACPoL is extended to support more operators as listed in Figure 6 with a rule example:

COMPARATIVE ANALYSIS

This section evaluates JACPoL with a comprehensive comparison to other pre- and post-XACML policy languages, which respectively are JSONPL (Steven, 2013), AWS IAM (Amazon Web Services), XACML (OASIS, 2013), Ponder (Damianou et al., 2001), Rei (Kagal et al., 2003), XACL (Hada & Kudo, 2000), KAoS (Uszok et al., 2004), EPAL (Ashley et al., 2003), and ASL (Jajodia et al., 1997), followed by a simple quantitative comparison with XACML in terms of processing delay. To begin with, the following requirements are identified 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 (He et al., 2013).
- *Extensibility* to cater for new features or concepts of policy in the future (Damianou et al., 2001).
- *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.

Figure 6. Other supported operators (left) and a rule example (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"
}
```

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.

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 (Neuhaus et al., 2011).

On the other hand, compared to XML-based languages, JSON-based JACPoL is simpler and more efficient, but meanwhile, there is a clear agreement 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.

PRELIMINARY PERFORMANCE EVALUATION

This section presents the results of preliminary performance tests on JACPoL and XACML. The assessment is given of respectively how both languages' evaluation time and memory usage are affected with regards to the increase of nesting policies (policy depth) and the increase of sibling ones (policy scale). The values are ranged from 0 to 10000 with a step of 1000, as shown in Figure 7.

The two policy languages are used to express the same policy task and to compare their performances with different depth (number of nesting policies) and scale (number of sibling policies). Figure 8 provides XACML and JACPoL policy examples used for assessment. The two ellipses in each example indicate nesting (the upper and inner ellipsis) and sibling (the lower and outer ellipsis) child policies respectively. When processing a given request, the Boolean expression in the Target field of each policy is evaluated to verify the applicability of its child policies until the last child policy is evaluated. Note that the JSON profile of XACML defines XACML requests and responses but not policies, therefore it is not possible to compare JACPoL with JSON-formatted XACML.

As shown in Figure 9, the policy evaluation performance of the PDP is very related to the writing, loading and processing performance of a policy language. It is clear that the schemes and technologies used in requesting/responding operations would also influence the performance, which is however beyond the scope of current JACPoL's specification.

Therefore, four sets of tests are conducted in order to evaluate the writing, loading and processing speed and the memory consumption of the two policy languages. Respectively, the writing test measures the average time (in seconds) to write policies from the memory into a single file; the loading test measures the average time (in seconds) to load policies from a single file into the memory; the processing test measures the average time (in seconds) to process a request against policies that are already

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

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	+	+	+	++	+	++

Figure 7. Examples of nesting policies (a) and sibling policies (b)

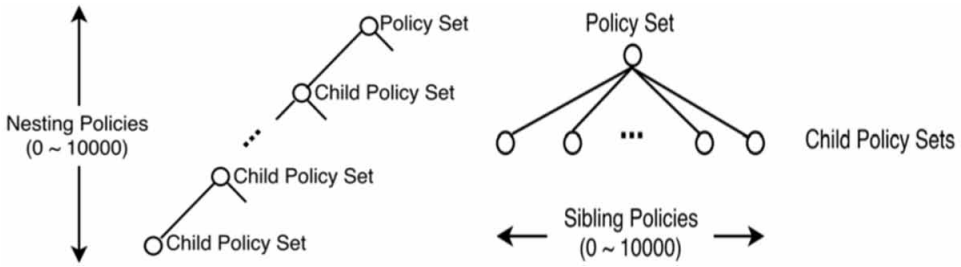


Figure 8a. Examples of XACML policies

```
<PolicySet Id="0" PolicyCombiningAlgorithm="firstApplicable" Update="2017-03-14 17:18:31" Version="1">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="…#string-equal">
          <AttributeValue DataType="…#string">Sam</AttributeValue>
          <SubjectAttributeDesignator AttributeId="…:subject:subject-id" DataType="…#string" />
        </SubjectMatch>
      </Subject>
    </Subjects>
  </Target>
  <PolicySet Id="1" PolicyCombiningAlgorithm="firstApplicable" Update="2017-03-14 17:18:31" Version="1">
    …
  </PolicySet>
  …
</PolicySet>
```

loaded in the memory; and the last test measures the space consumption (in Megabytes) of policies in the memory. Each test evaluates the two languages with different policy nesting depth and sibling scale, and was repeated 10000 times conducted using Python on a Windows 10 ASUS N552VW laptop with 16G memory and a 2.6GHz Intel core i7-6700HQ processor.

Figure 10 shows the writing time versus the number of policies. It can be observed that there are near linear correlations between the average writing time and number policies (both in depth and in scale) for both languages. Compared to XACML, JACPoL consumes only approximately half of the time taken by the former.

Figure 11 shows the average loading time versus the number of policies in depth and in scale. Similar to Figure 10, there is an almost linear correlation between the loading time and the number of policies. It can be seen that the loading time of JACPoL becomes much less than XACML as the number of policies increases.

Figure 12 shows the processing time versus the number of policies. Similar as the writing time and loading time, there is also an almost linear correlation between the processing time and the number of policies. On the other hand, unlike Figure 10 and Figure 11, it can be seen that the policy structure

Figure 8b. JACPoL policies


```
{
  "Target": {"Subject": {"equals": "Sam"}},
  "Update": "2017-03-14 17:18:31",
  "PolicyCombiningAlgorithm": "firstApplicable",
  "Version": 1,
  "Id": 0,
  "Policies": [
    {
      "Target": {"Subject": {"equals": "Sam"}},
      "Update": "2017-03-14 17:18:31",
      "PolicyCombiningAlgorithm": "firstApplicable",
      "Version": 1,
      "Id": 1,
      "Policies": […]
    },
    …
  ]
}
```


Figure 9. Policy evaluation procedure

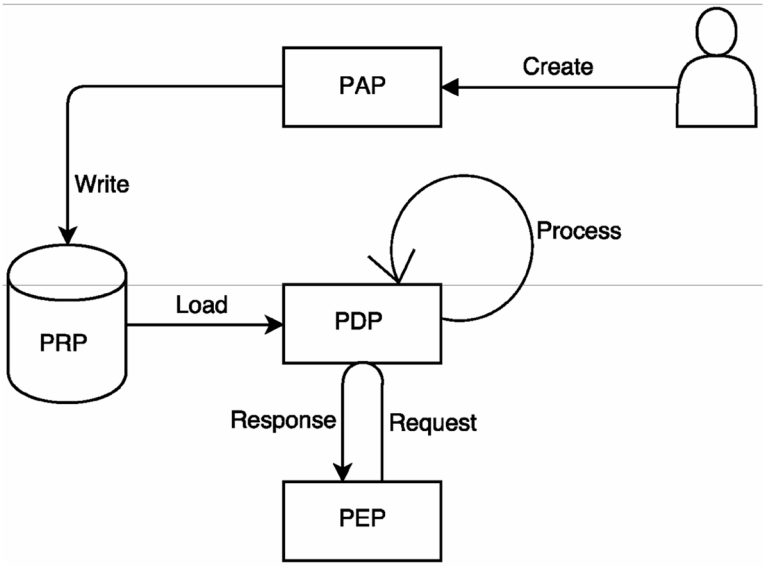


Figure 10. Comparing writing time consumption between JACPoL and XACML

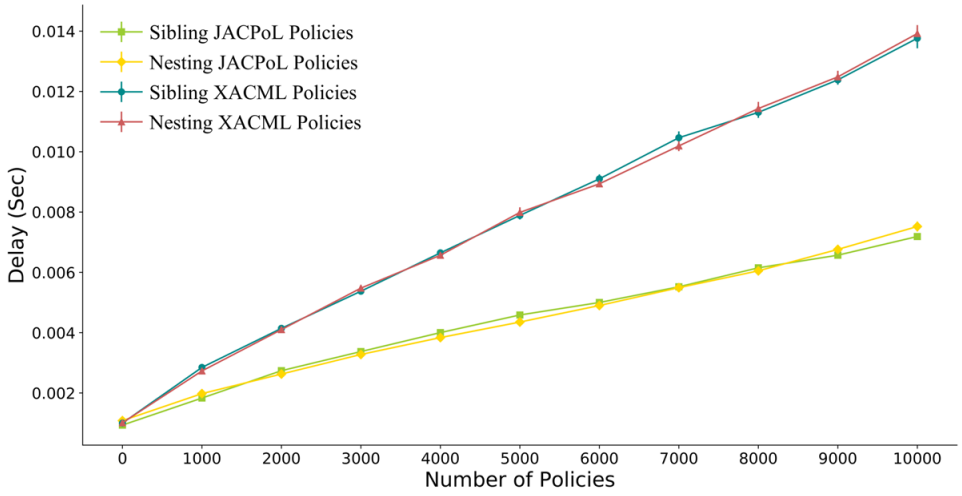
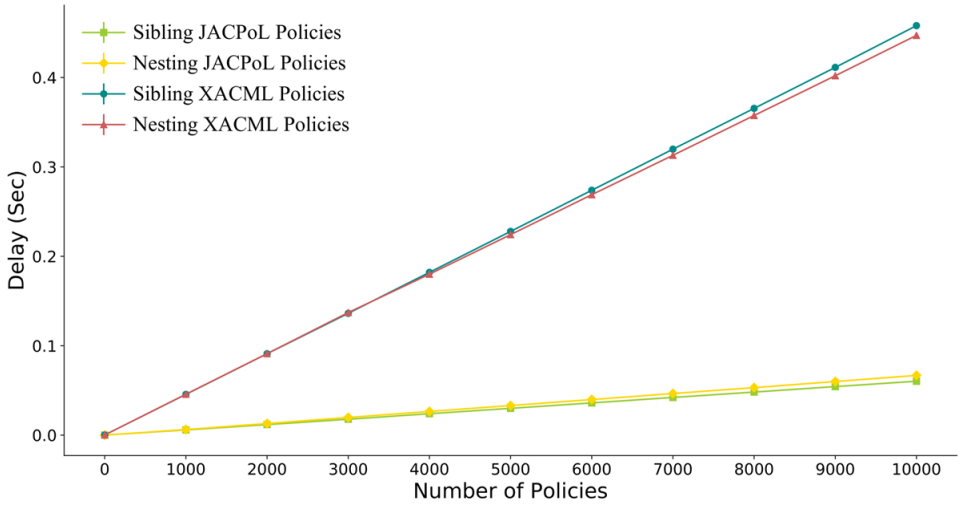


Figure 11. Comparing loading time consumption between JACPoL and XACML



would influence the processing time given the same policy size. It can also be observed that JACPoL is processed faster than XACML policies.

Figure 13 shows the memory consumption versus the number of policies. Similar as all figures above, the memory consumption increases also almost linearly with the growth of the number of policies. Given the same policy size, the memory space used by JACPoL is nearly half of that used by XACML.

These figures demonstrate that JACPoL is highly scalable and efficient in comparison with XACML, with much faster writing, loading, processing speed and lower memory consumption. The involved evaluation is however preliminary in the sense that we used several assumptions which limit its scope.

Figure 12. Comparing processing time consumption between JACPoL and XACML

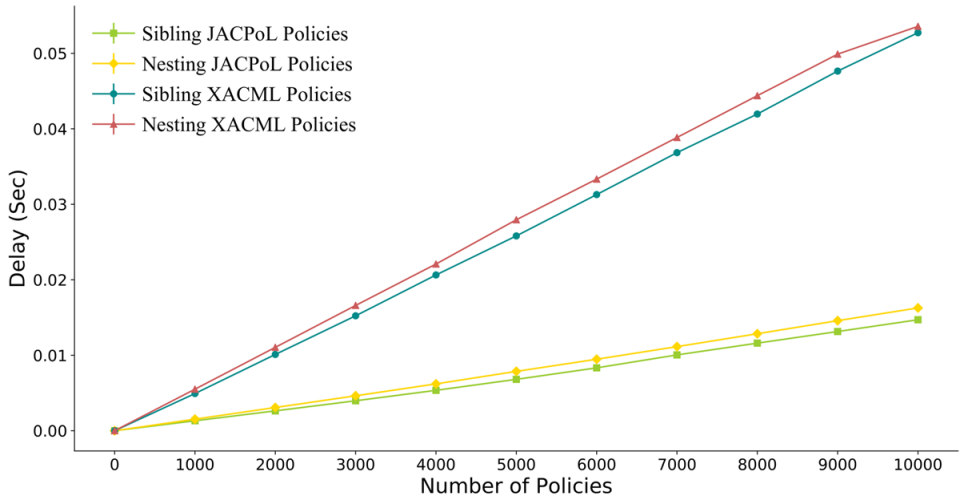
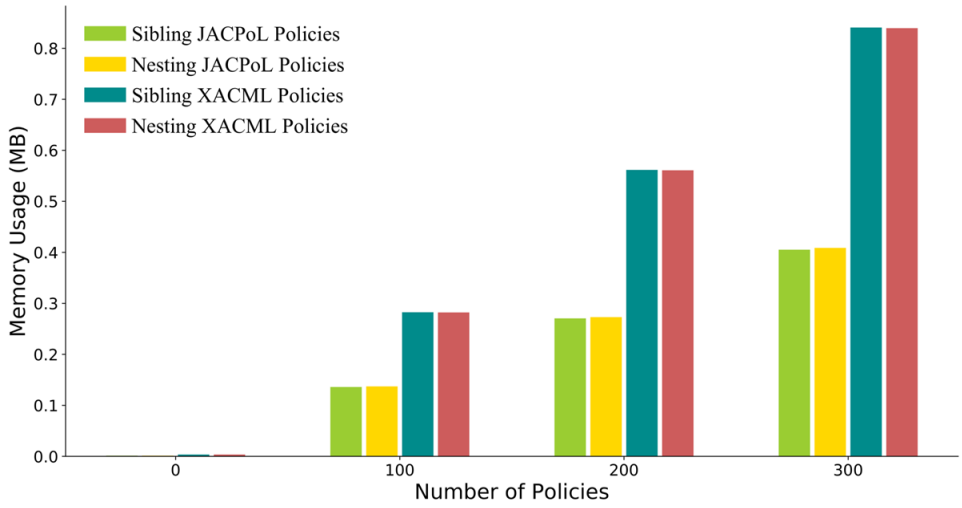


Figure 13. Comparing memory consumption between the JACPoL and XACML



We indeed compared two generic and structurally identical families of policies without taking into account their corresponding application domain neither the algorithmic used to implement the different evaluted processes, neither its eventual optimisation which could be derived from the underlying language XML or JSON. If this evaluation demonstrates that JACPoL offers interesting and promising prospects in terms of results, it will however have to be validated by an extensive and in-depth analysis that is out the scope of this work.

APPLICATION OF JACPoL TO SECURITY MODELS

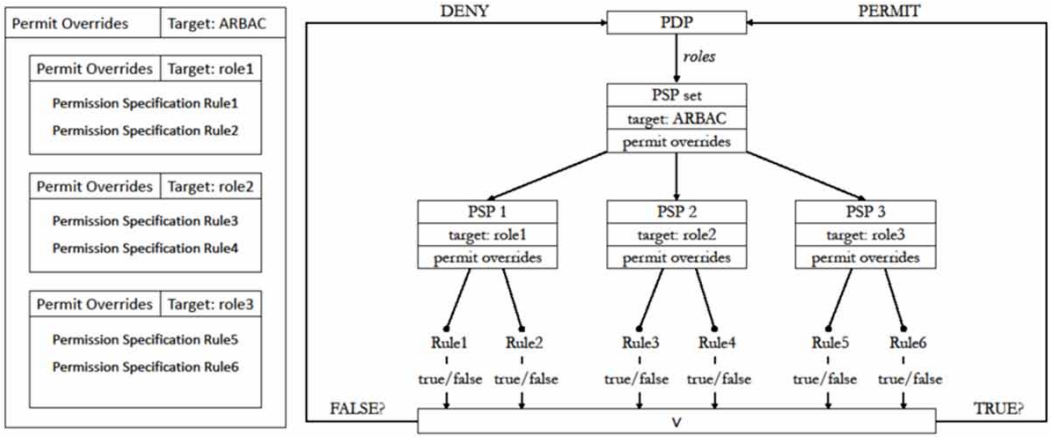
RBAC vs. ABAC

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 (Empower ID, 2013). 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. (Hu et al., 2013). 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 realtime but is complex and unauditable. Combining these two models judiciously to integrate their advantages thus becomes an essential work in recent research (Empower ID, 2013; Coyne & Weil, 2014; Jin et al., 2012; Kuhn et al., 2010).

Attribute-Centric RBAC Application

JACPoL can be implemented to express permission specification policies (PSP) in an attribute-centric RBAC model. For example, Figure 14 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. Please note that the role attribute is suggested to be placed as the target for the topmost level of policies in order to allow an easier view of user permissions as shown in Figure 14.

Figure 14. An example ARBAC permission specification policy and its structure tree



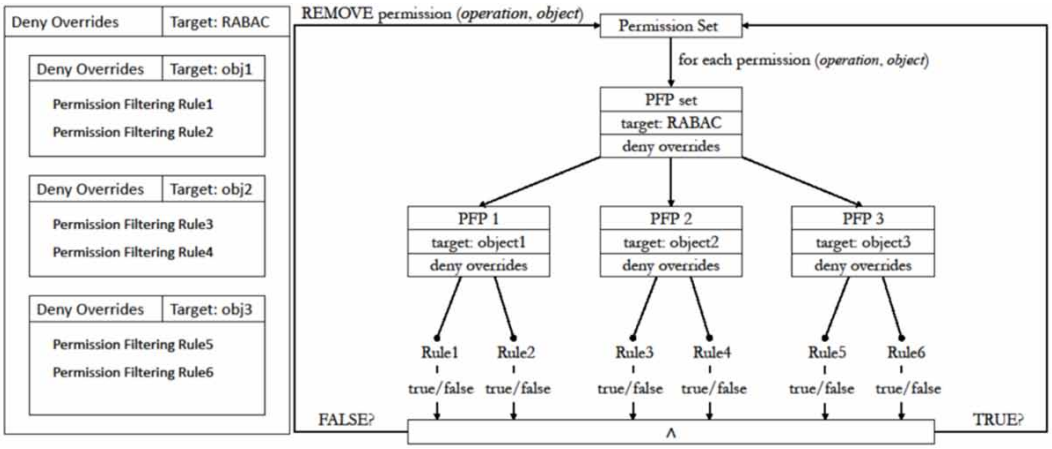
Role-Centric ABAC Application

JACPoL can also be used to implement a language for permission filtering policies (PFP) in a role-centric ABAC model (Jin et al., 2012). Similar to the ABAC model in Figure 5, but in addition to external attributes, the PDP also relies on the PIP to get role permission 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 15. 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 (Kuhn et al., 2010).

CONCLUSION

Traditionally, performance has not 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, JACPoL, a fast JSON-based, attribute-centric and light-weight access control policy language has been designed and implemented. JACPoL provides a good solution for policy specification and evaluation in such applications with low processing delay. The qualitative evaluation of this policy language is given 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.

Figure 15. An example RABAC permission filtering policy and its structure tree



A preliminary quantitative evaluation is introduced focusing on the PDP performances by assessing the speed of writing, loading and processing generic families of policies expressed in JACPoL and XACML together with their relative memory consumption. The comparisons show that JACPoL systematically requires much less time and memory space and demonstrate that it offers interesting and promising prospects that should be validated by an extensive and in-depth analysis.

On the other hand, JACPoL leaves room for future improvements in many areas. For example, obligation capabilities can be further enhanced and delegation support can be formally introduced. It could also be considered to combine JACPoL and XACML to benefit, for example, from the good performance of the former and the high extensibility of the latter, by integrating them into a common framework. Actually JACPoL and XACML follows the same PEP/PDP model and Rule/Policy/PolicySet architecture, the integration would be supported (or almost supported with minor development required) by design, one possible solution could be for example to support inserting an XACML policy in a JACPoL policySet, or an XACML rule in a JACPoL policy.

REFERENCES

W3schools. (n.d.). *JSON vs XML*. Retrieved from www.w3schools.com/js/js_json_xml.asp

Amazon Web Services. (n.d.). *AWS Identity and Access Management(IAM) User Guide*. Retrieved from http://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

Ashley, P., Hada, S., Karjoth, G., Powers, C., & Schunter, M. (2003). *Enterprise privacy authorization language (EPAL)*. IBM Research.

Bhatti, R., Ghafoor, A., Bertino, E., & Joshi, J. B. (2005). X-GTRBAC: An XML-based policy specification framework and architecture for enterprise-wide access control. *ACM Transactions on Information and System Security*, *8*(2), 187–227.

Borders, K., Zhao, X., & Prakash, A. (2005). CPOL: High-performance policy evaluation. *The 12th ACM conference on Computer and communications security*.

Brossard, D. (2014). *JSON Profile of XACML 3.0 Version 1.0*. XACML Committee Specification 01. Retrieved from http://docs.oasis-open.org/xacml/xacml-json-http/v1.0/cs01/xacml-json-http-v1.0-cs01.pdf

Clarkson, M. R., & Schneider, F. B. (2010). Hyperproperties. *Journal of Computer Security*, *18*(6), 1157–1210. doi:10.3233/JCS-2009-0393

Coyne, E., & Weil, T. R. (2014). ABAC and RBAC: Scalable, flexible, and auditable access management. *IT Professional*, *15*(3), 14–16. doi:10.1109/MITP.2013.37

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* (pp. 390-409). Springer. 10.1007/978-3-642-28641-4_21

Crockford, D. (2006). *JSON — The fat-free alternative to XML*. Retrieved from http://www.json.org/fatfree.html

Damianou, N., Dulay, N., Lupu, E., & Sloman, M. (2001). *The ponder policy specification language*. doi:10.1109/WOCC.2013.6676386

ECMA International. (n.d.). *ECMA-404 The JSON Data Interchange Standard*. Retrieved from http://www.json.org/

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

Empower, I. D. (2013). *Best practices in enterprise authorization: The RBAC/ABAC hybrid approach*. Empower ID, White paper.

Ferraiolo, D. (2016). Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC). *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*. 10.1145/2875491.2875496

Ferraiolo, D. F., & Kuhn, D. R. (2009). *Role-based Access Controls*. arXiv preprint arXiv: 0903.2171

Griffin, L., Butler, B., de Leastar, 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* (pp. 25-32). IEEE. 10.1109/POLICY.2012.15

Hada, S., & Kudo, M. (2000). *XML Access Control Language: provisional authorization for XML documents*. Academic Press.

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

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

Jajodia, S., Samarati, P., & Subrahmanian, V. S. (1997). A logical language for expressing autho- rizations. In *Proceedings of IEEE Symposium on Security and Privacy*. IEEE.

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.

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

Kuhn, D. R., Coyne, E. J., & Weil, T. R. (2010). Adding attributes to role-based access control. *Computer*, *43*(6), 79–81. doi:10.1109/MC.2010.155

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

OASIS XACML Technical Committee. (2013). *eXtensible access control markup langage (XACML) Version 3.0. Oasis Standard, OASIS*. Retrieved from http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

Obrsta, L., McCandlessb, D., & Ferrella, D. (2012). Fast semantic Attribute-Role-Based Access Control (ARBAC) in a collaborative environment. *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing*.

reTHINK CSP Policy Engine. (2016). Retrieved from github.com/reTHINK-project/dev-msg-node-nodejs/tree/master/src/main/components/policyEngine

reTHINK Deliverable 6.4. (2016). *Assessment Report*. reTHINK H2020 Project.

reTHINK Project. (2016). Retrieved from github.com/reTHINK-project/

reTHINK Project Testbed. (2016). *Deliverable D6.1: Testbed Specification*. Retrieved from https://bscw.rethink-project.eu/pub/bscw.cgi/d35657/D6.1%20Testbed%20specification.pdf

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.

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. 10.1007/978-3-540-24747-0_2

Yavatkar, R., Pendarakis, D., & Guerin, R. (2000). *A Framework for Policy-based Admission Control*. IETF, RFC 2753.

ENDNOTES

1 nodejs.org
2 Redis.io