Access control for on-demand provisioned cloud infrastructure services
Ngo, C.T.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Logical Model and Mechanisms for XACML

This chapter is based on the following publications:


4.1 Introduction

XACML is an authorization policy language in XML format based on the ABAC model. It composes policies from set of attribute criteria joined by logical operators to decide if authorization requests are granted. XACML is scalable in arranging policies in the hierarchical order in the repository. The policy language also supports delegations, obligations and advices, that makes it applicable in many areas such as networking, grids, clouds, enterprise organization and management. However, expansions of policies to address system scales will increase the complexity of the repository, which drops the policies evaluation performance.

XACML policies has complex structures containing a sophisticated logical model as follows:

- Policies are organized hierarchically in a policy-tree with rules, policies and policy-sets elements. The tree contains internal nodes and external nodes. An internal node can either be a policyset or a policy. Children of a policyset node can be other policysets or policies. Children of a policy are rules, which are external nodes. Because children can produce conflicting decisions, parent nodes can resolve them by predefined combining algorithms.
• Policy decisions are not only *permit* and *deny*, but also other intermediate values to handle error and un-matched situations such as *not-applicable*, *indeterminate* decisions (see section 4.3). It means that operations on combining policies’ decisions cannot be derived from binary logical operators. They should be defined in multi-valued logical domains.

• Not all attributes are processed equally, some of them are marked as critical (with the flag ‘*MustBePresent=true*’): during the evaluation, the missing of these attributes should yield *indeterminate* values rather than the *not-applicable*.

• Because policies have their own predicates to match with requests, attribute comparisons are scattered in the policy-tree. Thus, typical implementations discussed in [96] often have redundancies in evaluations: an attribute may be compared multiple times in different policy nodes.

With these characteristics, there are challenges to propose high performance policy evaluation solutions or resolve policy analysis and management problems. We need practical mechanisms that not only can gather predicates and efficiently reduce them in aware of combining algorithms, but also guarantee multi-valued logical semantics of the XACML.

To facilitate the high performance policy evaluation mechanism in the access control systems for clouds using XACML [48, 49, 97], from recent policy evaluation approaches [98, 99], and state-of-the-art of XACML engines performance [96], in this chapter we analyze the logic behind XACML standard and propose a practical decision diagram mechanism. It includes interval partition processing, MIDDs and their combination algorithms, which are then applied to design a high performance policy evaluation engine in Chapter 5. Our contributions in this chapter are as follows:

• Analyze the logic of XACML components evaluation, which essentially is a many-valued logic system with equivalent operators on different domains.

• Define the MIDD and X-MIDD data structures definitions representing logical expressions in XACML. Along with them, we define interval partition processing and related operators, which are then used to process XACML elements.

• Compared to related work, our approach covers most of XACML features in XACML 3.0 [36], including continuous data-types, complex comparisons, correctness of combining algorithm semantics, error handling and critical attribute setting.

The proposed mechanisms can also be applied to solve XACML policy management problems such as policy comparison, policy redundancy detection, policy testings or authorization reverse queries.

The rest of the chapter is organized as follows. Section 4.2 reviews the related work on policy analysis, management, integration and high performance evaluation.
Section 4.3 analyzes XACML logic that provides the basis for the proposed solution. Section 4.4 formulates the approach to evaluate the complete logical expressions using interval decision diagrams. Section 4.5 defines fundamental operations to process intervals, partitions and decision diagrams. These materials and mechanisms in this chapter can be applied to solve different policy management problems, which are pointed out in Section 4.6. Finally, Section 4.7 concludes the chapter.

4.2 Related Work

There are numerous prior works on access control policies that mainly focus on policy verification, analysis and testing to detect and remove redundancy [100–103]. Authors in [100] used propositional logic in XACML to identify properties of given policies and analyze the change-impact of two policies to summarize their differences. The proposal was implemented in the Margrave project using Multi-Terminal Binary Decision Diagram (MTBDD) [104] as the underlying mechanism. Because of using one binary variable for each attribute-value pair, their approach is only applicable for policies containing all predefined attribute values. In other aspect, by modeling decisions as binary values, it omitted error use-cases handling in all XACML evaluation semantics. Li & Tripunitara [101] and Hu & Ahn [103] proposed methodologies to verify and correct policies under the RBAC model. Kolovski et al. [102] used description logic to represent XACML policies and use DL reasoners for analysis tasks such as policy comparisons, verification and querying. However, because description logic could only covers a subset of XACML, this approach did not handle complex comparisons, indeterminate decisions handling as well as left out one-applicable combining algorithm. Masi et al. [105] formalized the XACML 2.0 semantics and proposed an alternative syntax supporting policy composition. They implemented a tool to compile policies into Java classes following the proposed semantic rules, where these classed are executed to compute policy decisions.

Policy integration and composition was introduced firstly by Bonatti et al. [106]. They defined an algebra with constraints to compose and translate policies into logic programs. However, the algebra did not bind with any practical policy language. Mazzoleni et al. [107] proposed the policy integration preferences, which is an XACML extension that specified how to integrate policies from different parties. In spite of that, they did not show any applicable mechanisms for such integration. Bruns et al. [108] attempted to use Belnap logic to formalize XACML 2.0, in which they map four logic values to XACML policy decisions. Even that, the logic of XACML is different from Belnap logic because the indeterminate values cannot map to any Belnap logical value. Subsequently, [109] used $D$-algebra to formulate combining algorithms in XACML 2.0. But the $D$-algebra did not correctly represent indeterminate decisions: e.g., with permit-override algorithm for \textit{indeterminate-p} ($\{p, n\}$) and \textit{deny} ($\{d\}$), the combination should be \textit{indeterminate-dp} ($\{p, d, n\}$) rather than the $\{p, d\}$. Rao et al. [110] defined a 3-value Fine-grained Integration Algebra (FIA) which attempted to formulate operations of XACML elements via FIA operators. They used MTBDD to represent their approach. However, the FIA could not represent all \textit{indeterminate} values and did not distinguish differences between
LOGICAL MODEL AND MECHANISMS FOR XACML

To solve the problem of policy evaluation performance, in the preliminary version [111] and later [98], Liu et al. attempted to transform policies into decision diagrams. Their approach started with the numericalization by mapping all attribute values into integer numbers, which was only applicable for equally comparisons policies with predefined attribute values. Although boosting the evaluation performance, this proposal only covered a subset of XACML policies when did not support the complex attribute comparisons, correct indeterminate decisions handling, critical attribute evaluation as well as obligations. Marouf et al. [112] reordered most frequent applicable policies using clustering techniques based on statistics from past requests, which could increase the chance in evaluating only a subset rather than the whole policies. This technique can only improve performance when incoming requests repeated with high probability, rather than the uniform random requests. Moreover, reordering policies would not support obligations handling.

The approach of Pina Ros et al.[99] extended [111] with interval techniques. They kept original data types of attributes with more supported comparison operators. The approach used two trees: the Matching Tree (MT) is a decision diagram built up from extracted predicates in target expressions, and the Combining Tree (CT) is at each MT’s leaf nodes. A CT contains a subset of the policy tree with only applicable rules or policies without Target elements. The CT evaluation followed defined combining algorithms in the subtree. This approach is different from [111] when it stores the subset of policy tree rather than the flat of applicable rules. However, this proposal has following flaws: first, it ignored critical attribute evaluation handling: e.g., if an attribute is missed from the request, the evaluation will yield a indeterminate with a critical attribute predicate rather than not-applicable. This leads to different decisions when combining with other rules, such as with deny-override algorithm for indeterminate and deny, the outcome will be indeterminate\textsubscript{DP}, rather than combining not-applicable and deny to deny. Second, because the CT only contains applicable rules equivalent to the matching path from the root of the MT, this approach could not handle error well if an attribute in the path is missed from the request. The evaluation would be blocked without giving any decision, while in practice, it always has a consistent answer for a given request.

Ramli et al. [113] presented the most recent work analyzing the logic behind XACML, in which evaluation semantics relied on operators over domains $V_3$ and $V_6$. Comparing to previous work, it covered most aspects in analyzing XACML logic, however, this approach still omitted to handle critical attribute setting, obligations as well as did not have any applicable implementation.

4.3 Semantics of XACML Policy Components

4.3.1 Abstraction

XACML elements [36] are organized in a hierarchical order, which contains policysets, policies and rules. Each of them has a Target expression as the criteria
for incoming requests. The returned decision is either defined in the rule's "effect" property, or combined decisions of children rules, policies or policysets.

Attributes in XACML have different data types. Without loss of generality, we assume a XACML attribute domain $D_i$ can be normalized to either $\mathbb{R}$ or a sub-domain of $\mathbb{R}$. So we can say that $D_i$ is the totally ordered domain representing a continuous data type.

A XACML request $X = \{x_1, x_2, \ldots, x_n\}$ is the set of attribute values, each item $x_i \in D_i$ with $D_i$ is an XACML attribute domain.

XACML specification [36] describes elements in XML using XML Schema Definition (XSD). For short representation purpose, we abstract main XACML elements that our logical analysis focuses in Table 4.1 using Backus-Naur Form notation.

The $<object>$ in obligation and advice represents a general object. The $<attr-id>$ identifies an attribute $a_i \in D_i$, the $<attr-value>$ specifies a constant value $v_i \in D_i$.

The combining algorithms are in the Table. 4.2. They define how to combine children’s decisions to the policy or policyset result.

Evaluation values of Match, AllOf, AnyOf, Target and Condition elements are summarized in Table 4.3, while the decision values of Rule, Policy and Policyset elements are in Table 4.4. XACML extends decision values with "not-applicable" and "indeterminate", compared to previous policy languages with only "permit" an "deny" decisions. The "not-applicable" means that the request does not match with the rules or policies, while the "indeterminate" values indicate that some errors may occur during evaluation (e.g., requests miss the critical attribute, errors in parsing policies). Because of this feature, XACML essentially is as a many-valued logic policy language, while prior work did not analyze and solve following this direction [98–100, 111].

A sample XACML policy is shown in the Listing 4.1. Originally it is the XML documents following XACML schema standard [36]. However in this example, we illustrate policies and rules as JSON objects for short representation. Target elements in the example are expressed as logical expressions.

In the sample policy, the ‘vol’ is the volume attribute, ‘p’ is the price attribute and ‘t’ is the time attribute. In the rule $R_0$, ‘vol’ attribute is marked as critical by the underline, otherwise it is optional.

### 4.3.2 Predicate Elements

#### 4.3.2.1 Match element

The XACML Match element is composed from a tuple of $(match-id, v, x)$ where the $match-id$ is a two-operand predicate Boolean function, $v$ is the attribute value as the first operand and $x$ is the attribute-id as the second one.

The match evaluation returns one of values in Table 4.3 as follows:

- If the comparison returns $true$, the result is "Matched".
- If there’s either no attribute $x$ found in the request, or the comparison is $false$, the result is "No-matched".
Table 4.1: XACML abstract syntax

\[
\begin{align*}
\langle Policyset \rangle & := (\langle target \rangle, \langle Policy-list \rangle, \langle combine-algo \rangle) \\
\langle Policy-list \rangle & := \langle Policy-item \rangle | \langle Policy-item \rangle \langle Policy-list \rangle \\
\langle Policy-item \rangle & := \langle Policy-set \rangle | \langle Policy \rangle \\
\langle Policy \rangle & := (\langle target \rangle, \langle Rule-list \rangle, \langle combine-algo \rangle) \\
\langle Rule-list \rangle & := \langle Rule \rangle | \langle Rule \rangle \langle Rule-list \rangle \\
\langle Rule \rangle & := (\langle target \rangle, \langle condition \rangle, \langle effect \rangle, \langle obligation-exprs \rangle, \langle advice-exprs \rangle) \\
\langle target \rangle & := \langle anyof-list \rangle \\
\langle anyof-list \rangle & := | \langle anyof \rangle \langle anyof-list \rangle \\
\langle anyof \rangle & := \langle allof-list \rangle \\
\langle allof-list \rangle & := \langle allof \rangle | \langle allof \rangle \langle allof-list \rangle \\
\langle allof \rangle & := \langle match-list \rangle \\
\langle match-list \rangle & := \langle match \rangle | \langle match \rangle \langle match-list \rangle \\
\langle match \rangle & := (\langle match-id \rangle, \langle attr-value \rangle, \langle attr-id \rangle) \\
\langle obligation-exprs \rangle & := | \langle obligation-expr \rangle \langle obligation-exprs \rangle \\
\langle obligation-expr \rangle & := (\langle object \rangle, \langle fulfill-on \rangle) \\
\langle fulfill-on \rangle & := \langle effect \rangle \\
\langle advice-exprs \rangle & := | \langle advice-expr \rangle \langle advice-exprs \rangle \\
\langle advice-expr \rangle & := (\langle object \rangle, \langle applies-to \rangle) \\
\langle applies-to \rangle & := \langle effect \rangle \\
\langle match-id \rangle & := eq | ne | gt | lt | ge | le \\
\langle combine-algo \rangle & := po | do | fa | ooa | pud | dup \\
\langle effect \rangle & := "permit" | "deny" \\
\langle request \rangle & := \langle attribute-list \rangle \\
\langle attribute-list \rangle & := \langle attribute \rangle | \langle attribute \rangle \langle attribute-list \rangle \\
\langle attribute \rangle & := (\langle attr-id \rangle, \langle attr-value \rangle)
\end{align*}
\]
Table 4.2: XACML combining algorithms

<table>
<thead>
<tr>
<th>Combining algorithms</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit-override</td>
<td>po</td>
</tr>
<tr>
<td>Deny-override</td>
<td>do</td>
</tr>
<tr>
<td>First-applicable</td>
<td>fa</td>
</tr>
<tr>
<td>Only-one-applicable</td>
<td>ooa</td>
</tr>
<tr>
<td>Permit-unless-deny</td>
<td>pud</td>
</tr>
<tr>
<td>Deny-unless-permit</td>
<td>dup</td>
</tr>
</tbody>
</table>

Table 4.3: XACML evaluation values for elements: Match, AllOf, AnyOf, Target and Condition

<table>
<thead>
<tr>
<th>Evaluation values</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matched</td>
<td>T</td>
</tr>
<tr>
<td>No-matched</td>
<td>F</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>IN</td>
</tr>
</tbody>
</table>

Table 4.4: XACML decision values for Rule, Policy and Policyset elements

<table>
<thead>
<tr>
<th>Decision values</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit</td>
<td>P</td>
</tr>
<tr>
<td>Deny</td>
<td>D</td>
</tr>
<tr>
<td>NotApplicable</td>
<td>N</td>
</tr>
<tr>
<td>Indeterminate{P}</td>
<td>IN_P</td>
</tr>
<tr>
<td>Indeterminate{D}</td>
<td>IN_D</td>
</tr>
<tr>
<td>Indeterminate{PD}</td>
<td>IN_PD</td>
</tr>
</tbody>
</table>

- If there’s no attribute $x$ found in the request, and this attribute is marked as critical with the flag $\text{MustBePresent}=\text{true}$, the result is “indeterminate”, meaning that an error occurs.

Denoting the set $V_M := \{T, F, IN\}$, the match evaluation can be represented as the function mapping from an attribute domain $D_i$ to match values $V_M$:

$$\mu(x_i) : D_i \rightarrow V_M$$

(4.1)

4.3.2.2 AllOf, AnyOf and Target elements

The AllOf element is a list of Match items joined by the $\land$ operator. Because the Match items return values in $V_M$ domain, the $\land$ operator is extended from the regular “AND” boolean operator:

$$\prod_{i=1}^{k} m_i := m_1 \land m_2 \cdots \land m_k = \begin{cases} 
T & \text{if } \forall i \in [1, k], m_i = T \\
F & \text{if } \exists i \in [1, k], m_i = F \\
IN & \text{if } \forall i \in [1, k], m_i \neq F; \exists j \in [1, k], m_j = IN
\end{cases}$$

(4.2)

The AnyOf element is a list of AllOf items joined by the $\lor$ operator, which is
Listing 4.1: Sample XACML policies

Policy $P_0$: \{ combine-algo: po, 
  target: \((vol \geq 100) \land (vol \leq 500)\), 
  rule-list: \([R_1, R_2]\), 
\}

Rule $R_1$: \{ effect: permit, 
  target: \[[100 \leq vol \leq 150] \land (12 \leq t \leq 17) \land (3 \leq p \leq 4)] \lor \[(300 \leq vol \leq 500) \land (1 \leq p \leq 2)] \lor \[(100 \leq vol \leq 150) \land (6 \leq t \leq 9) \land (1 \leq p \leq 2)], 
  obligations: \([{\{O_1, \text{permit}\}}]\) 
\}

Rule $R_2$: \{ effect: deny, 
  target: \[(vol = 100) \land (t = 17) \lor [(100 \leq vol \leq 300) \land (t = 9)] \lor [(vol = 500) \land (t \geq 12)], 
  obligations: \([{\{O_2, \text{deny}\}}]\) 
\}

defined as:

$$
\prod_{i=1}^{k} a_i := \text{a}_1 \lor \text{a}_2 \ldots \lor \text{a}_k = \begin{cases} 
T & \text{if } \exists i \in [1, k], a_i = T \\
F & \text{if } \forall i \in [1, k], a_i = F \\
IN & \forall i \in [1, k], a_i \neq T, \exists j \in [1, k], a_j = IN 
\end{cases}
$$

(4.3)

The Target element joins AnyOf elements by the $\land$ operator like in Eq. (4.2). In other aspect, the Target evaluation over incoming request $X \in D_1 \times D_2 \times \ldots D_n$ is also defined as the function $\tau$:

$$
\tau(X) : D_1 \times D_2 \times \ldots D_n \to V_M
$$

(4.4)

The XACML defines that an empty Target element returns the $T$ value.

From Eq. (4.2) and Eq. (4.3), we can see that these operators along with the set $V_M$ form a lattice $(V_M, \leq)$ with the order $F \leq IN \leq T$; $\land$ is the meet operator; $\lor$ is the join operator.

4.3.2.3 Condition element

The Condition element represents a complex logical expression evaluated by the set of attributes $X$ in the request to return a value in the $V_M$ domain. Without loss of generality, we can denote the Condition element as the function $\kappa$:

$$
\kappa(X) : D_1 \times D_2 \times \ldots D_n \to V_M
$$

(4.5)

4.3.3 Rules and Policies

4.3.3.1 Rule Evaluation

The rule $R = \{t, c, e\}$ in which $t, c, e$ are Target, Condition and Effect elements, respectively, is evaluated against a request $X$. The decision is based on the com-
bination of Target and Condition results, along with the effect value as in Table 4.5.

Table 4.5: XACML rule evaluation specification

<table>
<thead>
<tr>
<th>Target</th>
<th>Condition</th>
<th>Rule Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>Effect e</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>N</td>
</tr>
<tr>
<td>T</td>
<td>IN</td>
<td>INP if e = P, IND if e = D</td>
</tr>
<tr>
<td>F</td>
<td>any value</td>
<td>N</td>
</tr>
<tr>
<td>IN</td>
<td>any value</td>
<td>INP if e = P, IND if e = D</td>
</tr>
</tbody>
</table>

Denoting the set $E := \{P, D\}$ containing effect values and the set $V_R := \{P, D, N, INP, IND, INDP\}$ having decision values from Table 4.4, the rule evaluation can be represented as follows:

$$R(t, c, e) : V_M \times V_M \times E \rightarrow V_R$$

in which $t, c \in V_M$ are the results of the Target and the Condition evaluations, respectively; $e \in E$ is the Effect value. According to Table 4.5, the function $R(t, c, e)$ is evaluated as:

$$R(t, c, e) = \begin{cases} 
P & \text{if } t \land c = T \text{ and } e = P \\
D & \text{if } t \land c = T \text{ and } e = D \\
N & \text{if } t \land c = F \\
IN_e & \text{otherwise}
\end{cases}$$ (4.7)

The denotation $IN_e$ means the value $IN_P$ if $e = P$ and $IN_D$ if $e = D$.

4.3.3.2 Policy and Policyset Evaluations

Policy and Policyset evaluations are similar. Their decisions are relied on the Target element and the combined decision of their children using a combining algorithm. According to the XACML 3.0 standard, their evaluations are summarized in Table 4.6.

Table 4.6: XACML Policy/Policyset evaluation specification

<table>
<thead>
<tr>
<th>Target</th>
<th>Combining-algo decisions</th>
<th>Policy/Policyset decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>any value</td>
<td>N</td>
</tr>
<tr>
<td>F</td>
<td>any value</td>
<td>N</td>
</tr>
<tr>
<td>IN</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td></td>
<td>$P$</td>
<td>$INP$</td>
</tr>
<tr>
<td></td>
<td>$D$</td>
<td>$IND$</td>
</tr>
<tr>
<td></td>
<td>$INDP$</td>
<td>$INDP$</td>
</tr>
<tr>
<td></td>
<td>$INP$</td>
<td>$INP$</td>
</tr>
<tr>
<td></td>
<td>$IND$</td>
<td>$IND$</td>
</tr>
</tbody>
</table>

Denoting a policy $P = \{t, ca, \{R_i\}_{i=1}^k\}$ in which $t$ is the evaluation result of the policy's target: $t = \tau(X) \in V_M$; $ca$ is a combining algorithm in Table 4.2 and $R_i$ is a child rule of the policy. The policy evaluation $P$ is represented as the function:
which $\omega_{ca}(R_i)_{i=1}^k \in V_R$ is the combining function of children decisions in Section 4.3.4.

According to Table 4.6, denoting $\psi = \omega_{ca}(R_i)_{i=1}^k$, the function in Eq. (4.8) is evaluated as:

$$P(t, \psi) := t \triangledown \psi = \begin{cases} 
\psi & \text{if } t = T \\
N & \text{if } t = F \text{ or } t = IN, \psi = N \\
IN_P & \text{if } t = IN \text{ and } \psi \in \{P, IN_P\} \\
IN_D & \text{if } t = IN \text{ and } \psi \in \{D, IN_D\} \\
IN_{DP} & \text{if } t = IN \text{ and } \psi = IN_{DP} 
\end{cases} \quad (4.9)$$

Given a policy set $PS = \{t, ca, \{P_i\}_{i=1}^k\}$, the evaluation also relies on Eq. (4.9) to combine its policies’ decisions.

### 4.3.4 Combining Algorithms

XACML 3.0 combining algorithms operate on the $V_R$ domain, which are used to form the ancestor’s decision according to Eq. (4.9). Denoting a combining operator as the $\omega_{ca}$, in which $ca$ is the identifier of an algorithm in Table 4.2:

$$w_{ca}(v_1, v_2, \ldots v_k) : V_R^k \rightarrow V_R \quad (4.10)$$

with $v_i \in V_R$ is the child decision.

Combining functions in Table 4.2 are defined as follows:

$$\omega_{pa}(v_1, v_2, \ldots v_k) = \begin{cases} 
P & \text{if } \exists v_i = P \\
D & \text{if } \forall i \in [1, k], v_i \notin \{IN_P, IN_{DP}\} \text{ and } \exists v_j = D \\
N & \text{if } \forall i \in [1, k], v_i = N \\
IN_P & \text{if } \forall i \in [1, k], v_i \in \{IN_P, N\}, \exists v_j = IN_P \\
IN_D & \text{if } \forall i \in [1, k], v_i \in \{IN_D, N\}, \exists v_j = IN_D \\
IN_{DP} & \text{otherwise} 
\end{cases} \quad (4.11)$$

$$\omega_{do}(v_1, v_2, \ldots v_k) = \begin{cases} 
P & \text{if } \forall i \in [1, k], v_i \notin \{IN_D, IN_{DP}\} \text{ and } \exists v_j = P \\
D & \text{if } \exists v_i = D \\
N & \text{if } \forall i \in [1, k], v_i = N \\
IN_P & \text{if } \forall i \in [1, k], v_i \in \{IN_P, N\}, \exists v_j = IN_P \\
IN_D & \text{if } \forall i \in [1, k], v_i \in \{IN_D, N\}, \exists v_j = IN_D \\
IN_{DP} & \text{otherwise} 
\end{cases} \quad (4.12)$$
\[
\omega_{fa}(v_1, v_2, \ldots v_k) = \begin{cases} 
P & \text{if } \exists v_i = P \text{ and } \forall j \in [1, i), v_j \notin \{P, D\} \\
D & \text{if } \exists v_i = D \text{ and } \forall j \in [1, i), v_j \notin \{P, D\} \\
N & \text{if } \forall i \in [1, k], v_i = N \\
IN & \text{if } \forall i, v_i \in \{N, IN\} \text{ and } \exists v_j = IN
\end{cases} \quad (4.13)
\]

\[
\omega_{ooa}(v_1, v_2, \ldots v_k) = \begin{cases} 
P & \text{if } \exists ! v_i = P \text{ and } \forall j \neq i, v_j = N \\
D & \text{if } \exists ! v_i = D \text{ and } \forall j \neq i, v_j = N \\
N & \text{if } \forall i, v_i = N \\
IN & \text{otherwise}
\end{cases} \quad (4.14)
\]

\[
\omega_{pud}(v_1, v_2, \ldots v_k) = \begin{cases} 
D & \text{if } \exists i \in [1, k], v_i = D \\
P & \text{otherwise}
\end{cases} \quad (4.15)
\]

\[
\omega_{dup}(v_1, v_2, \ldots v_k) = \begin{cases} 
P & \text{if } \exists i \in [1, k], v_i = P \\
D & \text{otherwise}
\end{cases} \quad (4.16)
\]

We can see that for XACML elements, the Match, AllOf, AnyOf and Target elements operate over \(V_M\) domain, while the rule, policy and policyset evaluations use operators in \(V_R\) domains. In the next section, we define data structures and algorithms representing such elements and related operators, which facilitate the XACML evaluation implementation.

### 4.4 Multi-data-types Interval Decision Diagrams

#### 4.4.1 Introduction

Binary Decision Diagram (BDD) [114] and its extensions (e.g., Interval Decision Diagram (IDD) [115], Multi-Terminal Interval Decision Diagram (MTIDD) [116]) are popular in model checking, verification, firewall and policy analysis. However, they are not fully suitable in XACML processing. In their approaches, attribute data-types limit in discrete domains [115] or only using equality comparisons [98]. In general, actual attributes use continuous data-types with different comparable operators.

It is also possible to apply BDD techniques by using a variable for each attribute-value pair in the proposal of [100]. However, the depth of decision diagrams will be the product of the number of unique values and the number of attributes, therefore time and space complexities are much higher than our approach.

In this section, we construct a decision diagram based approach to implement operators described in the previous section efficiently. We define MIDD data structures with equivalent operators to meet such requirements. First, the section will revisit the basic logical function decomposition for continuous variables, which is represented by the decision diagram \(G(V, E)\). However, to match it with our functions over \(V_M\) and \(V_R\) domains in section 4.3, we need to extend the basic decision diagram into equivalent MIDD and X-MIDD diagrams. Then, we transform operators in section 4.3 to algorithms over MIDD and X-MIDD as in the next section.
4.4.2 Logical Function Decomposition

Denoting a multi-variable logical function with following signature:

\[
f : D_1 \times D_2 \ldots \times D_n \to \{true, false\}
\]  

(4.17)

where \(D_i\) is a totally ordered domain representing a continuous data-type in XACML. Denoting vector \(X = (x_i | i = 1..n, x_i \in D_i)\), Eq. (4.17) is also seen as: \(f(X) \to \{true, false\}\)

**Definition 4.1** (Data interval). A data interval \(I \subseteq D_i\) is a range of values in the domain \(D_i\) which is formed by two endpoints. It can either be an open, closed or half-closed interval, depending on whether endpoints are included in the interval.

**Example**: The sample policy (Listing 4.1) has different intervals for the ‘vol’ variable: in the rule \(R_1\), \(vol \in [100, 150]\) or \(vol \in [300, 500]\); or in the policy \(P_0\), \(vol \in [100, 500]\).

**Definition 4.2** (Interval partition). An interval partition \(P\) is a set of disjoint intervals in the domain \(D_i\):

\[
P = \{I | I \subseteq D_i : \forall I_i, I_j \in P, i \neq j, I_i \cap I_j = \emptyset\}
\]

**Example**: If we want to represent the ‘vol’ variable having values in the range \([100, 150]\) or \([300, 500]\), we define \(vol \in \{[100, 150], [300, 500]\}\). It is the interval partition containing disjoint intervals.

Given an interval partition \(P\), the denotation \(x_i \in P\) means that \(\exists I \in P, \text{s.t} \ x_i \in I\).

We define a boolean function \(h_{x_i}(P)\) as:

\[
h_{x_i}(P) = \begin{cases} 
0 & \text{if } x_i \notin P \\
1 & \text{if } x_i \in P 
\end{cases}
\]

(4.18)

Function in Eq. (4.17) is called independent with \(x_i \in X\) in the interval partition \(P\) when:

\[
\forall a, b \in P, f(X | x_i := a) = f(X | x_i := b)
\]

(4.19)

In this case, we denote \(f_{x_i}^P\) as the partial function:

\[
f_{x_i}^P := f(x_1,..,x_{i-1}, b, x_{i+1},..,x_n) | \forall b \in P
\]

(4.20)

**Example**: The function \(f\) below is said to be independent with \(vol\) in the partition \(P = \{[100, 150]\}\):

\[
f(vol, t, p) = (100 \leq vol \leq 150) \land (12 \leq t \leq 17) \land (3 \leq p \leq 4)
\]

So the partial function \(f_{vol}^{[100,150]} = (12 \leq t \leq 17) \land (3 \leq p \leq 4)\).

Given a domain \(D_i\), the set of partitions \(\mathcal{P}(D_i) = \{P_1, P_2,..,P_{d_i}\}\) is called to cover the domain \(D_i\) when

\[
D_i = \bigcup_{P \in \mathcal{P}(D_i)} \left( \bigcup_{I \in P} I \right)
\]

(4.21)
The cover $\mathcal{P}(D_i)$ is disjoint if there’s no common interval between them:

$$\forall i, j \in [1, d_i], i \neq j : P_i \cap P_j = \emptyset$$  \hspace{1cm} (4.22)

According to Boole-Shannon expansion, the function $f$ can be decomposed to set of partial functions in respect of variable $x_i$ against a disjoint, covered partition $\mathcal{P}(D_i)$

$$f(X) = \bigvee_{P \in \mathcal{P}(D_i)} h_{x_i}(P) \land f_{x_i}^P$$  \hspace{1cm} (4.23)

**Example:** With the example function $f$ above, we define the Boolean function $h_{[100,150]}^{vol} = 1$ if $vol \in [100, 150]$, otherwise $h_{[100,150]}^{vol} = 0$. So the function $f$ can be represented by the decomposition:

$$f(vol, p, t) = h_{[100,150]}^{vol} \land f_{[100,150]}^{vol}$$

We can represent the Boolean function $h(vol)$ by a simple decision diagram with one node having variable ‘vol’, an out-going edge with the predicate $[100,150]$ as illustrated in the Figure 4.1.

![Figure 4.1: An example of the function decomposition](image)

Each partial function $f_{x_i}^P$ can also be decomposed in respect to other variables, until it is independent from $\forall x_i \in X$. We can symbolize $f$ as a decision diagram $G(V, E)$ with following properties:

- $G$ is a rooted, directed acyclic graph (DAG) with the node set $V$ having two types of nodes: internal nodes containing variables and leaf nodes containing boolean values.

- The internal node $v_{x_i} \in V$ has a variable $x_i \in D_i$ of the function $f$. Each out-going edge $e_{x_i} \in E$ represents the function in Eq. (4.18) over a partition $P_{x_i} \in \mathcal{P}(D_i)$. It states the clause: $x_i$ has the value in the range of the partition $P_{x_i}$.

- Each sub-graph of the node $v_{x_i}$ is a partial function $f_{x_i}^P$ in Eq. (4.23).

The Figure 4.2 illustrates an example of $G(V, E)$. 
4.4.3 Multi-data-type Interval Decision Diagrams

The DAG $G(V, E)$ can only represent a boolean function in Eq. (4.17). In Section 4.3, Match, AllOf, AnyOf and Target elements have the signature in Eq. (4.24), while Rule, Policy and Policy set elements have the signature in Eq. (4.25).

$$f : D_1 \times D_2 \ldots \times D_n \rightarrow V_M$$

(4.24)

$$f : D_1 \times D_2 \ldots \times D_n \rightarrow V_R$$

(4.25)

We extend $G(V, E)$ to MIDD and X-MIDD representing Eq. (4.24) and Eq. (4.25) respectively.

**Definition 4.3** (MIDD). MIDD is the $G(V, E)$ representing a function having signature (4.24) over the $V_M$ domain:

- Each internal node $m$ in the MIDD is the tuple of $(x, s, C)$ in which $x$ is the node variable, $s$ is the state value: $s \in \{F, IN\}$. If $x$ is marked as critical, $s = IN$, otherwise $s = F$.
- The $C$ is the set of tuples $(p, c) \in C$, each represents an out-going edge containing a reduced interval partition $p$ connecting $m$ to a descendant node $c$.
- The descendant node $c$ could either be another internal node or the external node containing $T$ value. It is called the $T$-leaf-node.
- The evaluation of a request $X$ against a MIDD is the traversal from the root node: at an internal node $(x_i, s, C)$, an out-going edge $(p, c) \in C$ is selected if the value $x_i$ of the request $X$ belongs to the interval partition $p$: $x_i \in p$. 

![Figure 4.2: A decision diagram sample for the function decomposition](image-url)
• In the evaluation process, if $\exists x_i \in X$, the returned value is the state of the current internal node, which is either $F$ or $IN$. The evaluation returns $T$ if it reaches the $T$-leaf-node.

Examples of MIDDs can be found in the Figure 4.3.

![MIDD of the R1's Target](image1)

![MIDD of the R2's Target](image2)

(a) MIDD of the $R_1$’s Target  
(b) MIDD of the $R_2$’s Target

**Figure 4.3: Sample MIDDs of the Target elements**

**Definition 4.4 (X-MIDD).** X-MIDD is the $G(V, E)$ representing a function having signature (4.25) over the $V_R$ domain:

- An internal node $m$ is the tuple of $(x, s, O, C)$: the state $s \in V_R$; $O$ contains list of obligations and advices matching with $s$ if $s \in \{P, D\}$, otherwise it is empty.

- An external node contains a policy evaluation result, which can be represented as a tuple of $(s, O)$ with $s$ and $O$ are similar to the internal node.

- The evaluation of the X-MIDD can be defined recursively in the Algorithm 4.1.

Examples of X-MIDDs can be found in the Figure 5.1.

We see that functions over $V_R$ and $V_M$ domains in section 4.3 have their set of operators. To facilitate the representation of such functions with MIDD and X-MIDD, we build equivalent algorithms in the next section.

### 4.5 Interval Processing and Decision Diagram Operations

This section defines interval processing and MIDD composition operations, which are used to create MIDDs from XACML elements.
Input: Request $X$ and the X-MIDD with the root $m$
Output: Evaluation decision

1 begin
2 if $(m.x \notin X)$ then
3 return $(m.s, m.O)$;
4 else
5 foreach $((p, c) \in m.C)$ do
6 if $(X[m.x] \in p)$ then
7 return $x_{\text{eval}}(X, c)$;
8 end
9 end
10 return $(m.s, m.O)$;
11 end
12 end

Algorithm 4.1: X-MIDD evaluation: $x_{\text{eval}}$ function

4.5.1 Interval Partition Operations

**Definition 4.5** (Reduced interval partition). A reduced interval partition has least number of intervals compared to others having the same data ranges.

The reduction process of an interval partition is as follows: we find and combine intervals having adjacent ranges repeatedly until no adjacent-range interval is found. The result is the reduced interval partition.

While approach in [115] can only be used for integer data-type, our following definitions are more general and can support continuous data-types. Given two reduced interval partitions $P_1$ and $P_2$, we define operations as follows:

**Definition 4.6** (Union). $P = P_1 \lor P_2$ has below properties:
- $P$ is a reduced interval partition.
- All values belong to either partitions $P_1$ or $P_2$ also belong to $P$: $\forall v \in P_1 \cup P_2, v \in P$

**Definition 4.7** (Intersect). $P = P_1 \land P_2$ has the following properties:
- $P$ is a reduced interval partition.
- $P$ is composed from all common values of $P_1$ and $P_2$: $\forall v \in P_1 \cap P_2, v \in P$

**Definition 4.8** (Complement). $P = P_1 \ominus P_2$ is an interval partition that:
- $P$ is a reduced interval partition.
- It contains values of $P_1$ but not $P_2$: $\forall v \in P_1 \setminus P_2, v \in P$

For example, with $P_1 = \{[-3, 4.5], [6.3, 8]\}$, $P_2 = \{(2, 5.1], (7.5, 9]\}$, we have:
- $P_1 \lor P_2 = \{[-3, 5.1], [6.3, 9]\}$
- $P_1 \land P_2 = \{(2, 4.5], (7.5, 8]\}$
- $P_1 \ominus P_2 = \{[-3, 2], [6.3, 7.5]\}$
4.5.2 MIDD Operations

Given two functions $f_1$ and $f_2$ following the signature (4.24), we define conjunctive and disjunctive join algorithms representing operators in (4.2) and (4.3), respectively.

Variable ordering can affect the complexity of the MIDD, that we leave it for future work. Currently we choose an order in which variables appear in the policies. In the MIDD, the variable orders are lowest at the root and higher at deeper levels.

Let’s call $m_1$ and $m_2$ are MIDDs for functions $f_1$ and $f_2$, the combining operators are shown in Algorithm 4.2 and 4.3, respectively.

We denote $m.P$ as the union of all partitions of node $m$’s out-going edges:

\[ m.P := \{ \forall p, (p, c) \in m.C \} \]

4.5.2.1 MIDD Conjunctive Join

The algorithm representing conjunctive join operation in Algo. 4.2 is as follows:

- If either $m_1$ or $m_2$ is a T-leaf-node, the result is the other.
- Otherwise, if roots of $m_1$ and $m_2$ have the same variable, the root $m$ of the result MIDD contains the same variable. The node state of $m$ is the result of joining $m_1$ and $m_2$ states by the $\land$ operator. Child nodes of $m$ are created by conjunctively joining children of $m_1$ and $m_2$ with equivalent intervals.
- If two inputs do not have the same variable, the children of the lower order (say $l$) are conjunctively joined with the higher order (say $h$) to create the descendants of the result MIDD.

4.5.2.2 MIDD Disjunctive Join

The Algorithm 4.3 for disjunctive join operator is quite similar. We add a new edge as the complement of all union-ed $l$’s partitions to connect to $h$, meaning that if the value of $l.x$ in the request does not satisfy with $l.P$ predicate, then the decision is $h$. A special value $\perp$ is defined to handle the situation if the variable $l.x$ does not exist in the request $X$, the evaluation can traverse via this edge (lines 21-22).

In the Figure 4.4, we have three MIDDs constructed from parts of the $R_1$’s target expression in Listing 4.1. Applying the Algorithm 4.3, we have the combined MIDD representing the target expression of the rule $R_1$ in Figure 4.3a. We note that the critical attribute setting of the variable ‘vol’ in the first MIDD is transformed into the ‘IN’ state. However, in the disjunctive combination the result has the ‘$F’ state due to the operation in the line 5 following Eq. (4.3): $F \lor IN \rightarrow F$

4.5.3 MIDD to X-MIDD Transformation

With conjunctive and disjunctive join algorithms, we can compose MIDDs representing logical expressions in Target and Condition elements. In the rule evaluation, we need to transform a MIDD over $V_M$ domain into a X-MIDD over $V_R$ based on (4.7) as follows:
**Algorithm 4.2:** The MIDD conjunctive join algorithm

- Join Target and Condition MIDDs using $\land$ operator.
- Replace the T-leaf-node by the decision-leaf-node containing $(e, O)$: $e$ is the rule's effect, $O$ contains applicable obligations and advices.
- For each internal node $m := (x, s, C)$, we map the state $m.s$ from $\{F, IN\}$ to $V_R$ as follows:
  - If $m.s = F$, the new state is $N$.
  - If $m.s = IN$, the new state is $IN_e$ with $e$ is the rule's effect.

In [47], the critical attribute handling was implemented by the similar transforming function from MIDD to X-MIDD. However it required that critical settings of the same attribute in match expressions must be identical, which are either critical or non-critical. In this paper, we improve by the definition 4.3 in which the state value $s$ can store critical settings of match expressions. This state value is then handled by all MIDD algorithms in section 4.5.2 and the transformation process from MIDD to X-MIDD.

### 4.5.4 X-MIDD Operations

Policy evaluation logic and XACML combining algorithms in sections 4.3.3.2 and 4.3.4 require operations over X-MIDDs, which are represented in following algorithms:
Input: Two MIDDs : $m_1, m_2$
Output: Disjunctive join $m = m_1 \lor m_2$

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{if} (any $m_i$ is a T-leaf-node) \textbf{then}
\State \hspace{1em} \textbf{return} T-leaf-node;
\State \textbf{else if} ($m_1.x \equiv m_2.x$) \textbf{then}
\State \hspace{1em} $s \leftarrow m_1.s \lor m_2.s$;
\State \hspace{1em} $m \leftarrow (m_1.x, s, C := \emptyset)$;
\State \hspace{1em} $P \leftarrow m_1.P \bowtie m_2.P$;
\State \hspace{2em} \textbf{foreach} (interval $I \in P$) \textbf{do}
\State \hspace{3em} $c \leftarrow m_1.C[I] \lor m_2.C[I]$;
\State \hspace{3em} $p \leftarrow \{I\}$;
\State \hspace{3em} $m.C \leftarrow m.C \cup \{(p, c)\}$;
\State \hspace{1em} \textbf{end}
\State \textbf{else}
\State \hspace{1em} $l \leftarrow m_i$ that has lower variable order;
\State \hspace{1em} $h \leftarrow m_j$ that has higher variable order;
\State \hspace{1em} $m \leftarrow (l.x, l.s, C := \emptyset)$;
\State \hspace{2em} \textbf{foreach} (($p, c) \in l.C$) \textbf{do}
\State \hspace{3em} $c' \leftarrow c \lor h$;
\State \hspace{3em} $m.C \leftarrow m.C \cup \{(p, c')\}$;
\State \hspace{2em} \textbf{end}
\State \hspace{1em} $p' \leftarrow (((-\infty, +\infty)) \bowtie l.P) \setminus \{\bot\}$;
\State \hspace{1em} $m.C \leftarrow m.C \cup \{(p', h)\}$;
\State \hspace{1em} \textbf{end}
\State \hspace{1em} \textbf{return} $m$;
\State \textbf{end}
\end{algorithmic}
\end{algorithm}

Algorithm 4.3: The MIDD disjunctive join algorithm

4.5.4.1 Join Target with a Combined Children X-MIDD

The Algorithm 4.4 for policy evaluation Eq. (4.9) is almost similar to the MIDD conjunctive operator (Algorithm 4.2), with the differences that the join $\land$ is replaced by the $\bowtie$ operator.

4.5.4.2 Join X-MIDDs using XACML Combining Operators

Combining operators in Section 4.3.4 are used to join X-MIDDs, which are implemented in the Algorithm 4.5.

In the algorithm, if both $m_1$ and $m_2$ are decision-leaf-nodes, their decisions are combined using $\omega_{ca}$ operator with matching obligations and advices.

Otherwise, if roots of $m_1$ and $m_2$ have the same variable, the root of new X-MIDD also contains this variable. The default returned decisions from $m_1$ and $m_2$ are combined using $\omega_{ca}$. Its children is the combination of each of $m_1$ and $m_2$ children, respectively, aligned with each interval in the union interval partition $P = m_1.P \bowtie m_2.P$.

If a $m_i$ (say $l$) has lower variable order than the other (say $h$), we combine $h$ with each child of $l$ and add the output as the descendant of result MIDD $m$. We also add a new edge as the complement of all union-ed $l$’s partitions to connect to $h$, meaning that if the value of $l.x$ in the request does not satisfy with $l.P$ predicate,
then the decision is \( h \). A special value \( \bot \) is defined to handle the situation if the variable \( l.x \) does not exist in the request \( X \), the evaluation can traverse via this edge.
Input: X-MIDDs $m_1, m_2$; combining algorithm $ca$
Output: Combined X-MIDD: $m = \omega_{ca}(m_1, m_2)$

begin
if (all $m_i$ are decision-leaf nodes) then
  $s \leftarrow \omega_{ca}(m_1.s, m_2.s);$
  $O \leftarrow m_1.O(s) \cup m_2.O(s);$
  return $(s, O);$ 
end
if ($m_1.x \equiv m_2.x$) then
  $s \leftarrow \omega_{ca}(m_1.s, m_2.s);$
  $O \leftarrow m_1.O(s) \cup m_2.O(s);$
  $m \leftarrow (m_1.x, s, O, C := \emptyset);$
  $P \leftarrow m_1.P \cup m_2.P;$
  foreach (interval $I \in P$) do
    $c \leftarrow \omega_{ca}(m_1.C[I], m_2.C[I]);$
    $p \leftarrow \{I\};$
    $m.C \leftarrow m.C \cup \{(p, c)\};$
  end
else
  $l \leftarrow m_i$ that has lower variable order;
  $h \leftarrow m_j$ that has higher variable order;
  $m \leftarrow (l.x, l.s, l.O, C := \emptyset);$
  foreach $((p, c) \in l.C)$ do
    $c' \leftarrow \omega_{ca}(c, h);$
    $m.C \leftarrow m.C \cup \{(p, c')\};$
  end
  $P' \leftarrow \{(-\infty, +\infty) \cup l.P\} \cup \{\bot\};$
  $m.C \leftarrow m.C \cup \{p', h\};$
end
return $m;$
end

Algorithm 4.5: The algorithm for combining operators

4.6 Applications

Our main objective is to propose a high performance policy evaluation mechanism that can be applied in our access control approach for cloud. We illustrate that it can be solved by applying MIDD techniques in the next chapter. Beside that, we also point out that our mechanism can be reused in other policy management problems. At first, we define the following concept:

Given a policy $P$, let’s define $|P|_e$ is the set of requests that policy evaluation is $e$ with $e \in V_R$: $\forall X \in |P|_e, P(X) = e$.

**Definition 4.9** (Policy subset). Given two policy-trees $P_1$ and $P_2$ using the same attribute profile $\{a_1, \ldots, a_n\}$, $P_1$ is called the subset of $P_2$, denoting as $P_1 \subset P_2$ when $\forall e \in V_R \setminus N, |P_1|_e = |P_2|_e$.

We can see that $|P|_e$ essentially is the set of traversed paths from the root of the equivalent X-MIDD to the decision having value $e$.

Using the definition 4.9, we can point out following applications of the MIDD mechanisms:
• Policy testing: An important task in authorization policy testing is to enumerate all permit/deny or any decision \( e \in V_R \) requests that a given policy can yield. It can be done by transforming this policy in to X-MIDD, then enumerating all traverse paths from the X-MIDD’s root to a node containing decision value.

• Policy comparison: We can compare if the policy \( P_1 \subseteq P_2 \) by transforming them into X-MIDDs \( m_1 \) and \( m_2 \), enumerating all possible paths from \( m_1 \)’s root to decisions then make sure these paths also exists in \( P_2 \). The complexity of this problem is the complexities of transforming two policies (which are analyzed later) and the tree traversal problem.

• Reverse queries: the X-MIDD allows us to answer authorization queries in reverse orders: given a partial request, return the missing attributes and possible values that the policy can yield permit or deny decisions. A familiar sample reverse query could be “which resources can the subject Alice access during 9am-6pm?”. Using X-MIDD the problem becomes enumerating all possible paths reaching the permit decision for the partial request \{‘Alice’, ‘read’, time \( \in [9, 18] \}\}.

4.7 Conclusions

In this chapter, we analyze the logic behind XACML components and their evaluations. While XACML language is convenient to compose and manage authorization policies, there’s no efficient implementation mechanism used for policy evaluation and analysis. For such purpose, we define a new data structure based on the decision diagram concept, known as the MIDD, along with operations on interval processing and combining MIDD algorithms. These mechanisms can be applied to substantially improve evaluation performance of the PDP engine, which will be shown in the next chapter.