Chapter 5

High Performance XACML Policy Evaluation

This chapter is based on the following publications:


5.1 Introduction

Access control systems for clouds should require high performance request throughput. Motivated from requirements of our proposals using XACML [48, 49, 97] and state-of-the-art of policy XACML engines and approaches [96, 98, 99], in this chapter we design and implement a novel high performance XACML engine. Compare to prior work, it is the most complete work on policy evaluation with important XACML 3.0 features that are event absent from others before.

Formulated from the XACML logical analysis in Chapter 4, our engine distinguishes from prior work by the following contributions:

- Support complex comparison functions for continuous data-types: the MIDD mechanism allows us to transform policies with inequality comparisons as in Listing 4.1.

- Handle all logical expression forms in the policies.

- Preserve original evaluation semantics in XACML elements and combining algorithms, not only in no-error scenarios but also in cases with "not-applicable"
and "indeterminate" values in XACML 3.0, which is also compatible for version 2.0.

- Support Condition elements processing.
- Allow to define critical and non-critical attributes by the "MustBePresent" flag which is compliant with the XACML standard.
- Handle obligation and advice expressions in XACML elements.

Based on the refined analysis in section 4.3, our new algorithms are designed to follow evaluation standard: given a policy to be evaluated, first we create a MIDD for the target element; X-MIDDs of children rules are combined using the Algorithm 4.5; after that we join the target's MIDD with the combined X-MIDD using the Algorithm 4.4.

At first, we present the transformation process from regular XACML policies into decision diagram X-MIDDs in Section 5.2, and after that is the evaluation process in Section 5.3. We analyze the complexities of our approach in Section 5.4, then an implementation and experiments in Section 5.5. Finally, we conclude the chapter in section 5.6.

5.2 XACML Policy Transformations

In this section, by utilizing above defined operations, we solve the XACML evaluation problem by parsing and transforming XACML policies or policysets into X-MIDDs having equivalent evaluation semantics.

Given a XACML policy tree, our approach to construct an equivalent X-MIDD is as follows:

- Extract intervals and build MIDDs representing Target and Condition elements.
- Create the X-MIDD for each rule from its MIDDs following the mechanism in section 4.5.3.
- For a policy or policyset, join X-MIDDs of its children by equivalent combining operators to construct the final X-MIDD instance representing the root policy.

The final X-MIDD is then used for policy evaluation against incoming requests, which can be illustrated in Figure 5.2.

5.2.1 Creating MIDD from the Target Element

As described in Section 4.3, a match expression $m$ is a tuple of $(f, v, x)$ with the variable $x$ containing a state $s \in F, IN$ representing the "MustBePresent" setting. It can be represented by a MIDD with two nodes: the internal node $(x, s, C := \{I, c\})$, in which the interval $I = \{i \mid f(v, i) \rightarrow true\}$; $c$ is the external Tleaf-node.

In the algorithm 5.1, the MIDD of an AllOf element having a list of Match can be composed quickly by aggregating intervals of Match MIDDs having the same
variable $x_i$ using function $I' = \text{restrict}(I, f, v) := \{i | i \in I, f(v, i) \rightarrow \text{true}\}$. For example, if $I = (-5, 8)$, then $\text{restrict}(I, \geq 3) \rightarrow [3, 8)$.

From list of intervals $il$ having exactly an interval per variable, we build a MIDD that has only a path from the root to a T-leaf-node. Each node $n$ in the path has a variable $x_i$ joined state $sl[x]$ and an out-going edge having an interval $il[x]$.

```
begin
  il ← ∅; sl ← ∅;
  foreach ($m_i \in m$) do
    $I ← (il[m_i.x] = \text{null}) ? (-\infty, +\infty) : il[m_i.x]$;
    $s ← (sl[m_i.x] = \text{null}) ? F : sl[m_i.x]$;
    $il[m_i.x] ← \text{restrict}(I, m_i.f, m_i.v)$;
    $sl[m_i.x] ← s \land m_i.x.s$;
  end
  root ← null; tail ← null;
  foreach (attribute $x \in \text{sort}(il.keys))$ do
    $n ← (x, sl[x], C := (il[x], \text{null}))$;
    if (root = null) then
      root ← n; tail ← root;
    else
      tail.addChild(n); tail ← n;
    end
  tail.addChild(T-leaf-node);
  return root;
end
```

Algorithm 5.1: The parseAllOf function

The algorithm 5.2 creates the MIDD for a Target element using conjunctive and disjunctive join MIDD operators.

```
begin
  $t ← \text{T-leaf-node}$;
  foreach (AnyOf element $a \in A$) do
    $d ← \text{T-leaf-node}$;
    foreach (AllOf element $b \in a$) do
      $d' ← \text{parseAllOf}(b)$;
      $d ← d \lor d'$;
    end
    $t ← t \land d$;
  end
  return $t$;
end
```

Algorithm 5.2: The parseTarget function

Using this algorithm, the MIDDs of Target elements in rules $R_1$ and $R_2$ in the Listing 4.1 are shown in Figure 4.3.
5.2.2 Creating MIDD from a Condition Element

Given a request \( X \) and a \( \text{Condition} \) element \( \kappa(X) \) in Eq. (4.5), we create a temporary variable \( c := \kappa(X) \). In this case, the \( \text{Condition} \) element is similar to a \textit{Match} element with the signature \((=, T, c)\) and can be represented by the MIDD having an internal node \((c, s, \mathcal{C} := \{([T], e)\})\) connecting to the \textit{T-leaf-node} \( e \). The state \( s \) receives \textit{IN} value whenever there’s a critical attribute in the expression, otherwise it is \textit{F} value. We denote \( \text{parseCondition}(\text{cond}) \) as the function to create the MIDD from a \textit{Condition} element. It will be used in the process creating X-MIDD representing the rule evaluation.

5.2.3 Creating X-MIDD for a XACML Policy Tree

XACML policies are organized as a tree in which a node can be a rule, policy or policyset. Given a XACML policy node, the algorithm 5.3 is used to create its equivalent X-MIDD as follows:

- If this is a rule, we create MIDDs for \textit{Target} and \textit{Condition}, then transform the result into the rule’s X-MIDD following the Section 4.5.3 using \textit{transform} function.

- If the node is either a policy or a policyset, based on the policy evaluation in Section 4.3.3.2, we parse its children recursively to obtain X-MIDDs and combine them using \textit{\( \omega_{\text{ca}} \)} in algorithm 4.5. The combination is then joined to the node’s \textit{Target} MIDD using algorithm 4.4. The result is the X-MIDD representing the policy node.

```
Input: A policy node in the policy tree: \( n \)
Output: X-MIDD representing the subtree: \( m \)

begin
  \( d_t \leftarrow \text{parseTarget}(n.t) \);
  if (\( n \) is a rule) then
    \( d_c \leftarrow \text{parseCondition}(n.C) \);
    \( d \leftarrow d_t \land d_c \);
    \( m \leftarrow \text{transform}(d, n.e, n.O) \);
  else
    \( \psi \leftarrow \text{null} \);
    foreach (\( n_i \in n.\text{children} \)) do
      \( d \leftarrow \text{parsePolicyNode}(n_i) \);
      \( \psi \leftarrow \omega_{\text{ca}}(\psi, d, n.O) \);
    end
    \( m \leftarrow m \land \psi \);
  end
return \( m \);
end
```

\textbf{Algorithm 5.3:} The \( \text{parsePolicyNode} \) function

From MIDDs in Figure 4.3, using \textit{transform} function we have equivalent X-MIDDs in Figure 5.1.
Using the algorithm 5.3 starting from the root, we can create a X-MIDD representing the policy tree. For the sample in Listing 4.1, we have the X-MIDD in Figure 5.2.
5.3 XACML Policy Evaluations

5.3.1 Single-valued Request Evaluation

According to the XACML 3.0 standard [36], the evaluation process at the PDP is to find and combine decisions of applicable rules and policies in a policy tree for a given authorization request. Our transformation process in Section 5.3 allows to construct an X-MIDD structure having equivalent evaluation semantics with the policy tree. The evaluation process of a single-valued request \( X = \{x_1, x_2, ..., x_n\} \) against the X-MIDD \( m \) is as follows:

- The traverse is from the root of \( m \) to a leaf-node, or when no attribute value is found.
- At an internal node \( m_i \), if the value of the attribute \( m_i.x \) exists in the \( X \), find a matched out-going edge \((p, c) \in m_i.C\) from \( m_i \) and continue the traverse from the child node \( c \).
- If either no matched out-going edge found, or \( X \) does not contain value of \( m_i.x \), the evaluation return the node's state \( m_i.s \).
- If the current traverse node \( c \) is a leaf-node, returns its decision \((s, O)\).

A single-valued request \( X \) is defined as a list of attribute values, exactly one value per attribute, \( X = \{x_1, x_2, ..., x_n\} \). Evaluating \( X \) against policies is to find the matching path in the equivalent X-MIDD. At a node, finding the matching edge can be done either using sequential or binary searches, since out-going edges of an internal nodes having ordered interval partitions. We currently support single-valued requests.

If the matching path found, the engine reaches the leaf node. Here, if the condition \( C \) is "true", the result is the decision value and equivalent objects (i.e., obligations, advices). Otherwise, it returns \( N \). If the matching path is not found due to missing attributes or no applicable edge at the node \( n \), the evaluation return \( n.s \in V_R \), the default returned-decision.

5.3.2 Multi-valued Request Evaluation

XACML allows multi-valued attribute requests, where an attribute can store a list of values (e.g., a person can have several roles: employees, managers, etc). However, current XACML behavior on multi-valued requests processing has some concerns. With policy in Listing 4.1, given a request \( X = \{150, \{10, 19\}, 4\} \) with the \( \text{time} \) attribute can either 10am (10) or 7pm (19), the evaluation following XACML standards [36, 38] such as SunXACML [82] claims that \( R_1 \) is the applicable rule. It has such result because each value in the bag of \( \text{time} \) attribute \( \{10, 19\} \) is checked with each match expressions: the value 10 passes the condition \( \text{time} < 17 \), and value 19 passed the condition \( \text{time} > 12 \). While in practical, we expect \( R_1 \) is not applicable when it requires a \( \text{time} \) value passes both conditions, not two separate values.
[98] and [99] decomposed multi-valued requests in set of single-valued requests to evaluate and combine all of applicable rules. As illustrated in the above example, this processing differs from original standards.

For multi-valued policies supports in approaches of [98, 99], we argue that they only used in XACML condition expressions, not in target expressions due to the value \( v \) in \( m_k := (x, f, v) \) can only receive a literal value [36, 38]. Because they only deal with target expressions, their supports are redundant. Our solution replaces condition expressions as variables, so it is possible to handle multi-valued attributes in policies.

5.4 Analysis

5.4.1 Features Comparison

Based on logical analysis in Section 4.3, our proposed mechanism covers most of missing XACML features from prior works (e.g. in [98] and [99]):

- We have succeeded to fully support XACML logical expressions analyzed in Section 4.3 with multiple data-types and comparison operators.
- Preserving original combining algorithms semantic in handling indeterminate and not-applicable states: prior work could handle simple Permit or Deny decisions, but incorrectly for others.
- Critical attribute setting: to the best of our knowledge, we are the first work to support this feature with high performance evaluation.

5.4.2 Complexities

5.4.2.1 Space complexity

We suppose that a policy-tree uses \( n \) attributes \( \{a_1, a_2, ..., a_n\} \), \( a_i \in P_i \). Assuming that the domain \( P_i \) has \( k_i \) different values appearing in the policies, so \( P_i \) can be separated into at most \( 2k_i + 1 \) intervals or partitions, including open and degenerate intervals. The X-MIDD representing the policy tree has at most \( 2k_i + 1 \) out-going edges from any node at level \( l_i \).

With \( n + 1 \) levels, the largest number of nodes at level \( l_i \) of the X-MIDD is \( \prod_{j=1}^{i} (2k_j + 1) \). Therefore, the worst-case space complexity is \( O(\sum_{i=1}^{n} \prod_{j=1}^{i} (2k_j + 1)) \).

It shows that the space complexity in the worst case does not depend on neither the policies size, height of policy tree, nor the complexity of logical formulas in their target expressions. It only depends on the number of attributes and number of distinct attribute values in the policy tree. Similar to the BDD approach, the size of the MIDD is affected heavily by the attribute ordering [114]. The implementation shows that our algorithms have efficient performance with reasonable graph size.
5.4.2.2 Evaluation time complexity

The policy evaluation process in our X-MIDD is the traversal from the root to a leaf node or an internal node where it cannot find any applicable out-going edges. At the level $l_i$ of the X-MIDD, it has to find an applicable item among at most $2k_i + 1$ out-going edges. Using the sequential search, the evaluation time complexity is $O\left(\sum_{i=1}^{n} (2k_i + 1)\right)$. For binary search it is $O\left(\sum_{i=1}^{n} \lceil \log_2(2k_i + 1) + 1 \rceil \right)$.

Similar to the space complexity, it is shown that the evaluation time complexity only depends on number of attributes $n$ and number of appearing attribute values $k_i | i = 1..n$. It is the advantage of the proposed approach to evaluate a large number of policies containing complex logical expressions. However, the drawback is the memory cost with policies having a large number of $n$ and $k_i$ values.

Our approach has the time complexity does not depend on number of policies, but may not handle well on systems with high number of attributes. Actually, numbers of attributes in authorization systems are often quite limited, could be up to about 10 attributes. However, the number of policies usually expands in proportion to the organization and system scale. Therefore, the proposed approach is still useful in practical.

5.5 Implementation and Evaluation

In our experiments, we compare our implementation with the standard SunXACML engine [82].

We do not make direct experiments to compare with prior work [98, 99]. The work in [98] has better performance in datasets using only equality operators due to numerical comparisons are faster. In other cases, it does not support datasets with inequality operators. The approach in [99] does not publish neither its implementation nor datasets. However, we see that our X-MIDD structure has similar time complexity to their matching-tree (MT) in the worst case. But after MT evaluation, they need to evaluate the combining-tree (CT), that the time complexity relies on height of policy tree. So ours has somewhat better evaluation performance.

5.5.1 Environment and Datasets

We implement our SNE-XACML engine [85] in JRE 1.7. Experiments are carried out on a Linux x64 system with Intel i5 core 2.67 GHz and 4GB RAM. The datasets are XACML 3.0 policies.

Due to lacking of XACML 3.0 implementations, we can only compare our work with the popular XACML 2.0 engine, SunXACML. It means that sophisticate indeterminate decisions in v3.0 in our work cannot compare to decisions from SunXACML. This evaluation needs to be improved by conformance tests of XACML 3.0 in the future. Currently, such tests do not present in the XACML community.

In our experiments, we convert policies 3.0 back to version 2.0 in order to be compatible with the referenced SunXACML. All indeterminate values in version 3.0...
are mapped to an indeterminate value in version 2.0.

We use three datasets in Table 5.1. The first one is a real-life policy taken from GEYSERS project [10] with some obligations and a critical attribute marked as “MustBePresent=true”. The continue-a policy is taken and converted from [100]. And the synthetic-360 is our randomly generated policy using 80% equality operator and 20% inequality operators. We also select random mixture of all combining algorithms.

We see that most target expressions in continue-a policy are trivial with either empty or a few match expressions in a level. The synthetic-360 policy is more complex: each Target, AnyOf and AllOf elements contain from 0 to 4 children. It is the reason why the X-MIDD generated from the synthetic-360 is more complex than from continue-a.

In our testbed, we generate requests randomly following uniform distribution for each datasets. They are then evaluated by two engines.

Table 5.1: Sample Policy Datasets

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Policy levels</th>
<th>#Policy sets</th>
<th>#Policies</th>
<th>#Rules</th>
<th>Attributes</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEYSERS [10]</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>33</td>
<td>3</td>
<td>=</td>
</tr>
<tr>
<td>Continue-a [100]</td>
<td>6</td>
<td>111</td>
<td>266</td>
<td>298</td>
<td>14</td>
<td>=</td>
</tr>
<tr>
<td>Synthetic-360 [47]</td>
<td>4</td>
<td>31</td>
<td>72</td>
<td>360</td>
<td>10</td>
<td>= (80%), complex(20%)</td>
</tr>
</tbody>
</table>

5.5.2 Validation

Our empirical validation is as follows:

- Experiments are performed on the three given datasets in Table 5.1, being a public policy in related work [100], our synthetic policy [47] and a project policy [10].
- For each dataset, 1000 random requests are generated uniformly distributed over attribute value ranges.
- Generated requests are evaluated by two engines: the standard, popular XACML engine [82] and our XACML engine [85]. Outputs of two engines are expected to be the same.
- We run experiments multiple times.

Results of these empirical experiments confirm the correctness of our approach and implementation.

5.5.3 Performance Analysis

We run the testbed ten times, each evaluates a million requests by the SNE-XACML and a thousand ones by the SunXACML. The average response times of two engines
are shown in Figure 5.3. We observe that our engine is about three orders of magnitude faster than the SunXACML. Its response times are almost the same in three datasets, while the basic engine’s performance is heavily dependent on the complexity of policies’ logical expressions. Figure 5.4 shows the standard deviation of evaluation response time. With all datasets, SunXACML has greater variation in response times than ours. This illustrates our analysis of evaluation time complexity, which is linear in attribute sizes and logarithmic complexity in attribute values.

Figure 5.5 shows our micro-benchmark results. The pre-processing time and X-MIDD size are highly dependent on policy complexity, which is the number of attribute $n$ and number of attribute values $k$. The X-MIDD of continue-a policy with 14 attributes has less nodes than the diagram of synthetic-360 policy with 10 attributes. Because the later is randomly generated, therefore it contains higher $\{k_i\}$ values.

The request evaluation time is composed from three parts: time to extract attribute values from XACML requests, time to evaluate these values on the X-MIDD representing the policy and time to create XACML response messages. These time fractions are shown in Figure 5.6. We can see that the response conversion time fraction is negligible, while the request conversion time is quite remarkable compared to the X-MIDD evaluation time. In most cases, response messages only contain decisions, thus their sizes are usually small. However, sizes of request messages in the XML format depend on the number of attributes and attribute values. Therefore, request messages are much larger than response messages. The experiment shows that the request conversion times increase proportionally to the
number of attributes in three datasets.

Figure 5.6 also illustrates that the X-MIDD evaluation fraction times are remarkable in the total response time when the number of attributes is small, e.g., in GEYSERS's dataset. For datasets with more attributes, the X-MIDD evaluation fraction times are in the range about 50%-60%. We can conclude that the conversion overhead is the bottleneck in ours due to the inefficient of XML parsing compared to the optimized policy evaluation. Note that the conversion overheads in SNE-XACML and other engines are quite similar because they often use popular XML processing mechanisms such as JAXB.

5.6 Conclusions

In this chapter, we applied our analysis results in Chapter 4 to design and develop a high performance XACML policy evaluation engine. It not only boosts the system performance, but also preserves original evaluation semantics, which is missed in prior work. The proposed solution could handle the complicated logical expressions defined in policies' predicates, correctness of combining algorithms semantics, critical attribute setting, obligations and advices handling. The evaluation and analysis show that the presented approach has the efficient performance in time complexity while having the reasonable space complexity. Experiments prove that our solution implemented in the open sourced SNE-XACML engine [85] has both
the significant performance compared to the referenced SunXACML engine [82] and validated evaluation results.