EPiC Series in Computing
Volume 58, 2019, Pages 65–74
Proceedings of 34th International Conference on Computers and Their Applications

XACML Implementation Based on Graph Database

Ying Jin and Krishna Chaitanya Kaja

Department of Computer Science, California State University, Sacramento
Sacramento, CA 95819-6021, USA
jiny@csus.edu

Abstract

Extensible Access Control Markup Language (XACML) is an OASIS standard for security policy specification. It consists of a policy language to define security authorizations and an access control decision language for requests and responses. The high-level policy specification is independent of underlying implementation. Different from existing approaches, this research uses a graph database for XACML implementation. Once a policy is specified, it will be parsed and the parsing results will be processed by eliminating duplicates and resolving conflicts. The final results are saved as graphs in the persistent storage. When a XACML request is submitted, the request is processed as a query to the graph database. Based on this query result, a XACML response will be produced to permit or deny the user's request. This paper describes the architecture, implementation details, and conflict resolution strategies of our system to implement XACML.

Keywords: Extensible Access Control Markup Language (XACML), Graph Database System, Conflict Resolution

1 Introduction

Extensible Access Control Markup Language (XACML) [6] is a high-level language that allows users to specify access control policies in the form of Extensible Markup Language (XML). XACML has a policy language to define access control decisions. It also allows users to define requests to access resources. Whether a request is granted or not depends on the predefined policy specifications. There are various ways for XACML implementation. This research uses a graph database approach to implement XACML.

Graph database system is one type of NoSQL (Not Only SQL) database systems. Compared to traditional SQL databases, NoSQL systems have different data models. Graph database systems are based on the graph model, in which the basic components are nodes and arcs. Nodes present entities and arcs present relationships between entities. Compared to relational database systems that commonly use JOIN operations, graph databases use graph traversal to handle similar issues.

This research uses the widely used graph database Neo4j [4] for XACML implementation. Specifically, we parse the policies and store the results in Neo4j. When a XACML Request is issued, the request is handled by querying the graph database using Cypher Query Language (CQL) [5] of Neo4j. Our past research has focused on Role-Based Access Control (RBAC) of XACML. This research is not restricted by RBAC, instead, it handles the general Attribute-Based Access Control (ABAC). Because of the nature of ABAC, conflict resolution is one important factor in our design and implementation, which will be detailed in Section 4.

This paper uses an example of medical application to illustrate the concepts, which includes information such as doctors, nurses, and pharmacists, as well as resources such as patient records, lab results, and medication. The example is based on the example of [1].

The rest of the paper is organized as follows. Section 2 is the brief overview of XACML policy structures and related work. Section 3 presents our system architecture, design, and implementation. Section 4 describes our methodology for conflict resolution. Section 5 concludes the paper with a summary of our work.

2 Background and Related Work

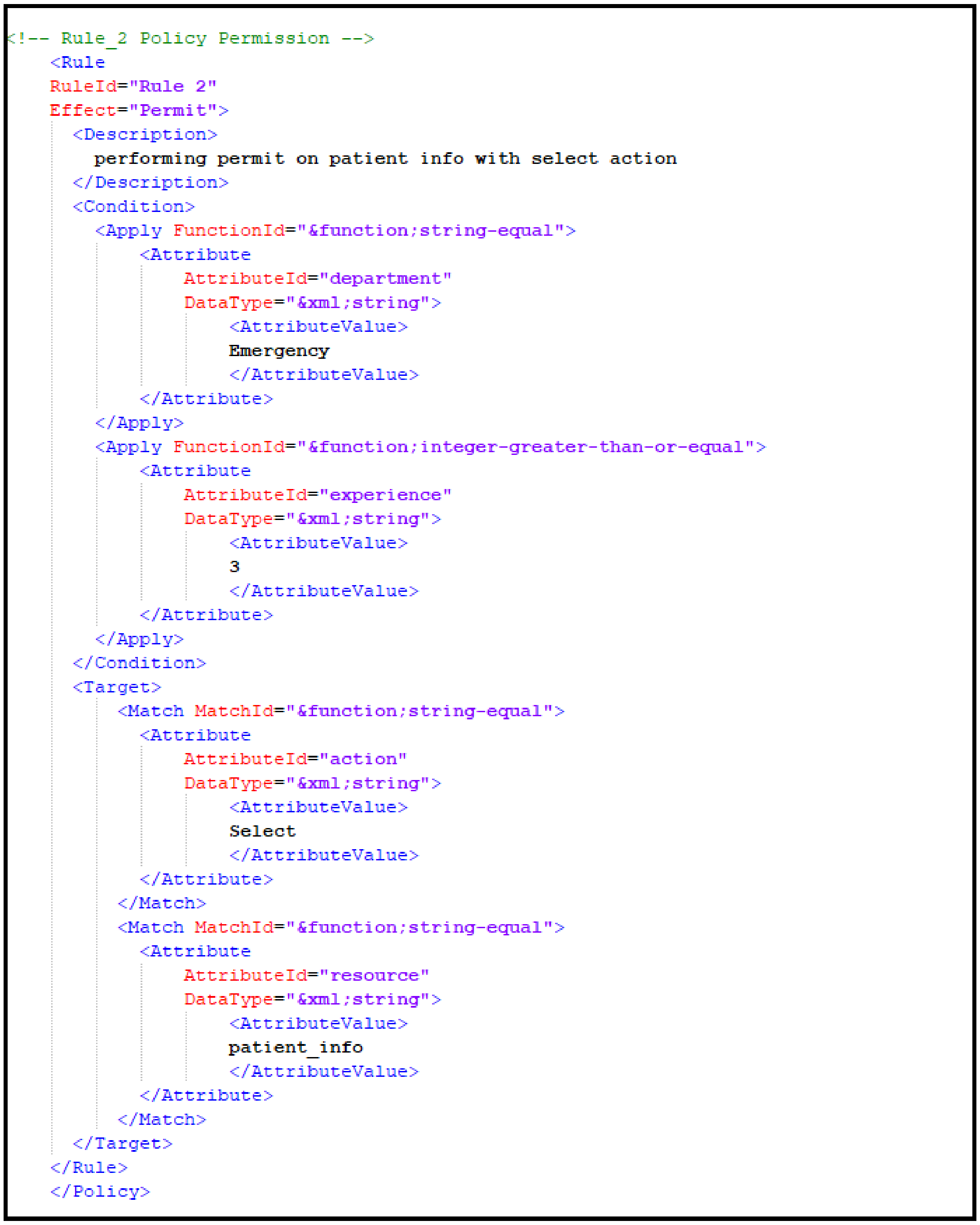
XACML policy language has the structure of policy set, policy, and rule. A policy set consists of one or more policies. One policy consists of one or more rules. Figure 1 shows an example of a rule which specifies the access right from a user to a resource. The “Target” tag contains the “Action” tag with attributes such as Select, Insert, and Update. In this policy, a user from Emergency department with the experience of three or more years will be permitted to perform Select operation on Patient_info.

A XACML request file allows a user to specify a request to access a resource. The answer to the request is formed as a XACML Response file with the access permission of “Permit” or “Deny”.

Various approaches are used to implement XACML. Sun’s XACML [8] open source implementation verifies XACML policies on-the-fly to handle access requests. XEngine [3] is based on the same principle, however, enhancing performance by preprocessing. XACMLight is an Apache Axis2 Web Service [10] to implement a Policy Decision Point (PDP) and a Policy Administration Point (PAP) that are defined in XACML 2.0.

A different type of solution is the database-based approach. MyABDAC system [1] used access-control lists of MySQL relational database to do implementation. It discussed Attribute-Based Access Control (ABAC) of XACML, which is what we are handling in this research.

Our past research has been focused on the Role-Based Access Control (RBAC) of XACML. We have used the role-based access control mechanism that is embedded in a relational database to implement XACML [2, 7]. We also have researched on using Neo4j graph database for RBAC, as reported in [9]. Different from [9], this research is targeting ABAC. Although based on similar high-level architecture as our previous research, the underlying data structure is totally different. One challenge is conflict resolution in ABAC, which will be discussed in detail in Section 4.



```
<!-- Rule_2 Policy Permission -->
    <Rule
    RuleId="Rule 2"
    Effect="Permit">
      <Description>
        performing permit on patient info with select action
      </Description>
      <Condition>
        <Apply FunctionId="&function;string-equal">
            <Attribute
                AttributeId="department"
                DataType="&xml;string">
                    <AttributeValue>
                    Emergency
                    </AttributeValue>
            </Attribute>
        </Apply>
        <Apply FunctionId="&function;integer-greater-than-or-equal">
            <Attribute
                AttributeId="experience"
                DataType="&xml;string">
                    <AttributeValue>
                    3
                    </AttributeValue>
            </Attribute>
        </Apply>
      </Condition>
      <Target>
          <Match MatchId="&function;string-equal">
            <Attribute
                AttributeId="action"
                DataType="&xml;string">
                    <AttributeValue>
                    Select
                    </AttributeValue>
            </Attribute>
          </Match>
          <Match MatchId="&function;string-equal">
            <Attribute
                AttributeId="resource"
                DataType="&xml;string">
                    <AttributeValue>
                    patient_info
                    </AttributeValue>
            </Attribute>
          </Match>
      </Target>
    </Rule>
    </Policy>
```

Figure 1: Example XACML

3 System Architecture and Implementation

Our system consists of two modules. One module is for processing XACML policies, and the other is for handling XACML requests.

Figure 2 illustrates the architecture of the XACML policy module. A Policy Writer produces a XACML policy, which is the input to the Policy Parser Engine. The Parser Engine uses Java libraries of Document Object Model (DOM) to read every tag from the XACML policy. User and resource information that is stored in the database is also retrieved. This step of processing establishes a relationship between a user and a resource, which specifies whether a user can access a resource with particular action, according to the policy's original specification, for example, "Permit" "Bob" to do "Insert" operation on "Patient_Info". The results are key-value pairs stored in a data structure of HashMap. Since this result is produced directly from the policy, conflicts can occur. In addition, ruleId, policyId, and PolicySetId are also saved, so the structure of the XACML policy is maintained. The additional information is also used in the conflict resolution process.

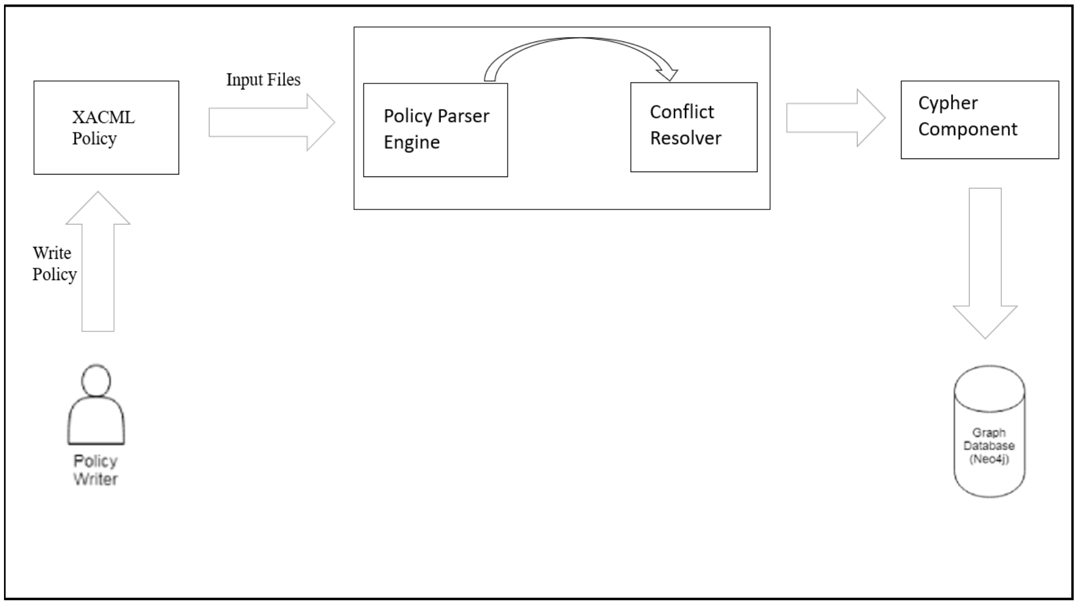


Figure 2: XACML Policy Module

Next, the Conflict Resolver performs conflict resolution to produce refined results, which will be described in details in Section 4. The Cypher Component converts the refined results into Cypher query statements.

For example, the following statement reflects this policy: "Permit" "Bob" to do "Insert" operation on "Patient_Info".

Match(e:Employee {name:"Bob"}),(s:Resource{name: "patient_info"}) create (e)-[:Insert {effect:"permit"}]->(s)

Further, the Cypher Component communicates with Neo4j database and executes all the statements. The successful execution of the above statement stores the policy semantics in the graph database as an arc from the user to the resource, as shown in Figure 3. One property of the arc is "effect", where the "effect" value is "permit". Upon executing all Cypher statements, all the authorizations between users and resources will be established. As a result, the input XACML policies eventually stored as a graph in the persistent storage. An example of the results is shown in Figure 4.

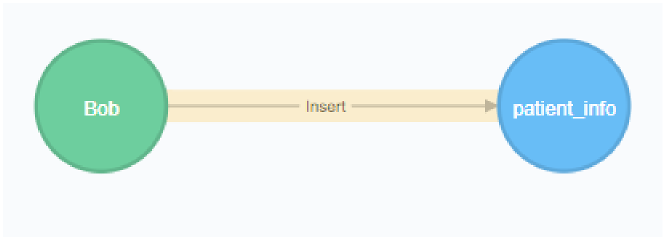
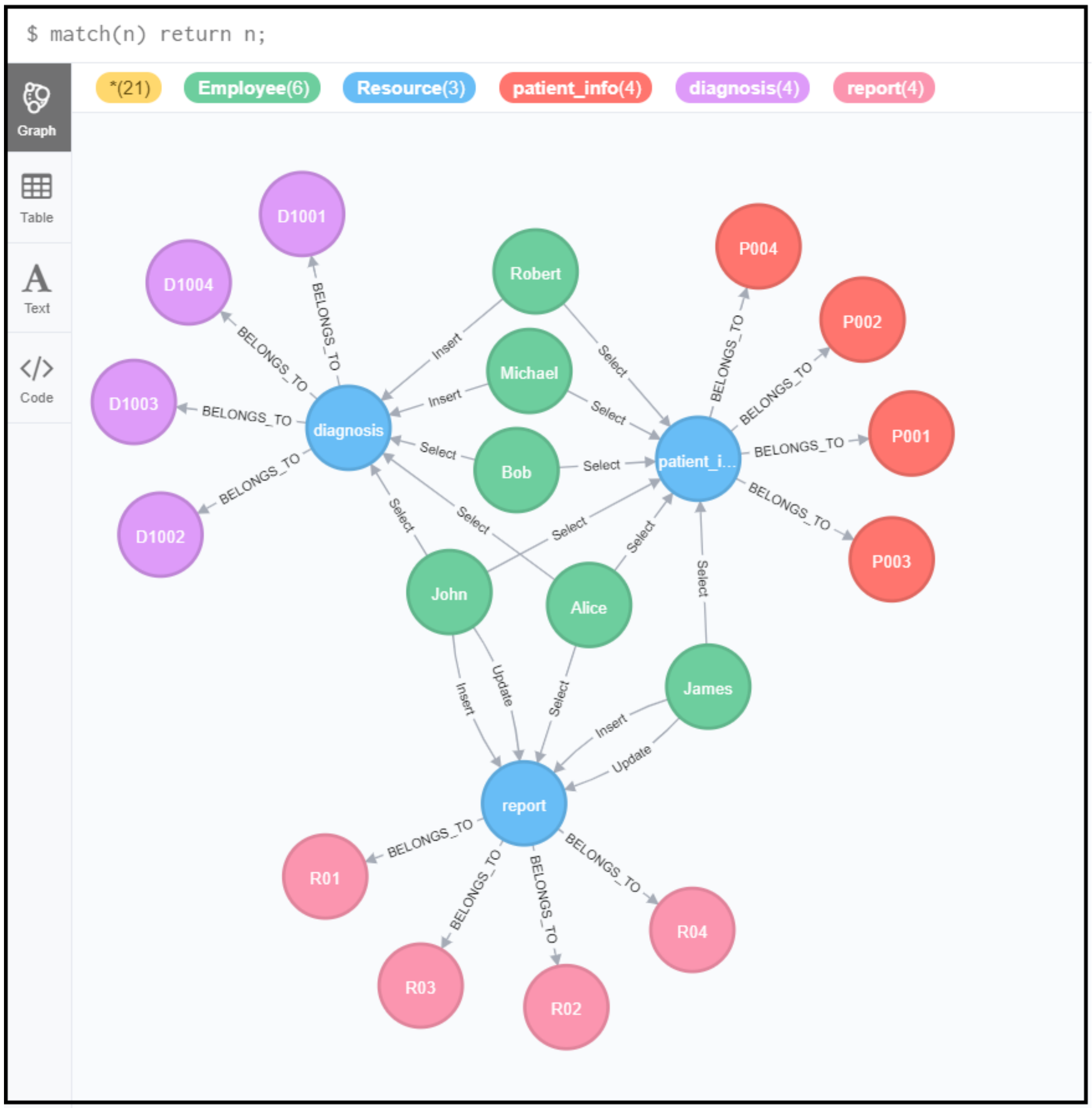


Figure 3: Example of Stored Rule



```
$ match(n) return n;
```

Figure 4: Neo4j Storage of the XACML Policy

The XACML request/response model is to handle users' requests of accessing resources. As shown in Figure 5, users can form a XACML Request file to ask for access authorization. The Request Parser Engine parses the input by examining tags such as "category" and "attributeId" and generates a Cypher query statement. The CQL statement generated is pushed into the Neo4j database to query the database. The answer retrieved from the Neo4j database is converted to a XACML Response file. If the query result is "permit", then "permit" will be returned in the response file. Otherwise, the response file will have "Deny" as the decision. For example, based on the graph in Figure 3, if the request file asks whether "Bob" can do "Insert" on "Patient_info", the Cypher query will return "permit". As a result, the XACML Response file returns with "permit".

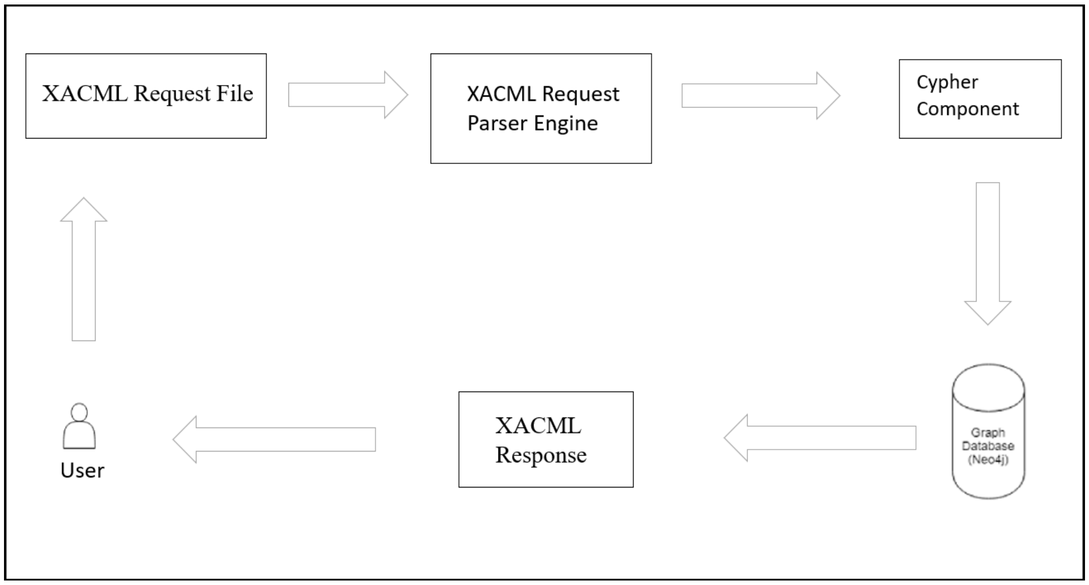


Figure 5: XACML Request/Response Module

Compared to our past research on XACML Role-Based Access Control, the high-level architecture is similar. However, since the underlying data structures to store policies are totally different, the details of parsing and all other processing are very different from our previous work.

4 Conflict Resolution

After parsed a XACML policy, duplicates and conflicts can occur between the same user and the same resource. Our system handles duplicate elimination firstly before performing conflict resolution.

4.1 Duplicate Elimination

Duplicates occur when the same action permission is given to the same user and the same resource by different rules. Figure 6 provides an example, where the rules are simplified without XACML syntax. In R1, "Permit" any users in the Emergency department, whose level is two or more, to do "Select" operation on "Patient_info". In R2, "Permit" any users in the Emergency department, whose experiences are three or more, to do "Select" operation on "Patient_info". Suppose Bob is in the Emergency department, with technical level of 2 and experiences of 3 years. According to Rule 1, Bob has Select permission on Patient_Info. According to Rule 2, he also has Select permission on

Patient_Info. If we save the results to the database directly, we will have two arcs between Bob and Patient_Info, which is redundant information, as shown in Figure 7. To handle this issue, we firstly consider duplicates at the policy level, eliminating all duplicates within the same policy. For example, as shown in Figure 8, either Rule 1 or Rule 2, say Rule 1, will be selected at the policy level for Policy 1, while the other one will be eliminated. Similar operations will be performed on Policy 2, in which Rule 3 is selected. Next, at the policy set level, the winners from Policy 1 and Policy 2 will be compared again, and duplicates will be eliminated at the policy set level. In the case that Rule 1 and Rule 3 are duplicates, we will select one of the rule at the policy set level, say Rule 1.

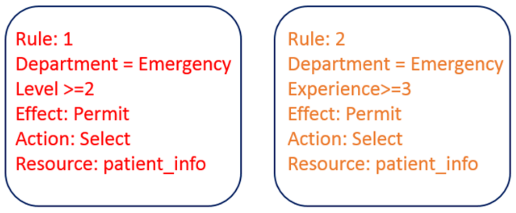


Figure 6: Example Rules with Duplicate Results

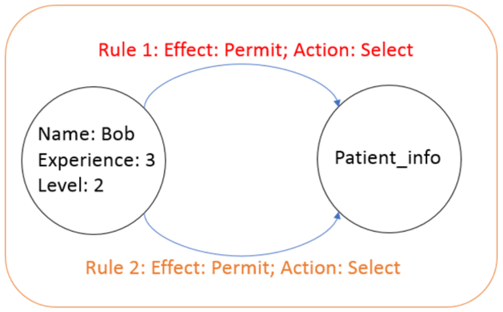

Rule 1: Effect: Permit; Action: Select
Name: Bob
Experience: 3
Level: 2
Patient_info
Rule 2: Effect: Permit; Action: Select

Figure 7: Example Rules in Graph Databases

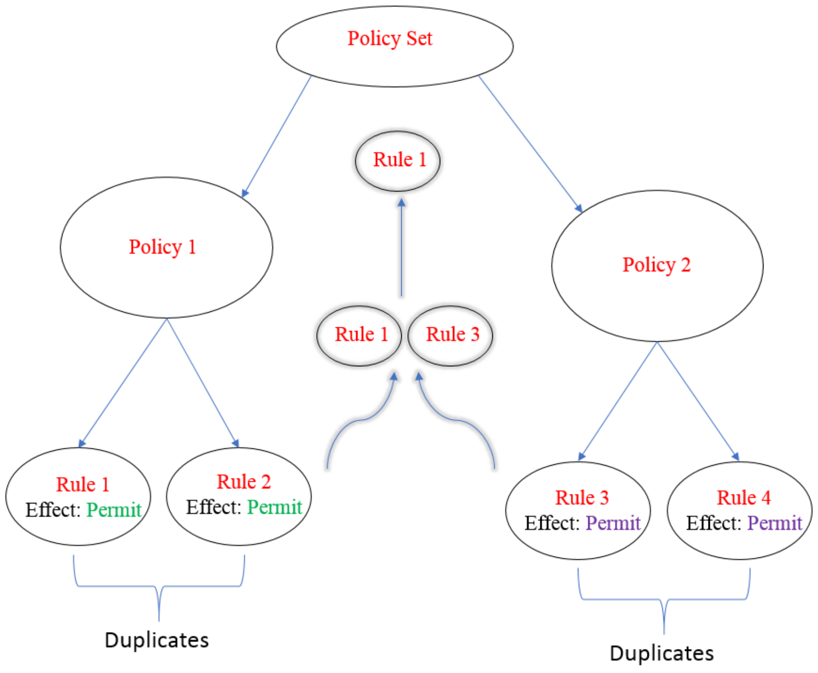


Figure 8: Duplicate Elimination Example

4.2 Conflict Resolution

Conflicts occur when the same user and the same resource have different action permissions, such as Select, Insert, and Delete, after XACML parsing. Figure 9 gives an example of conflicting action permissions between two different rules. If Alice is in Internal Medicine department with level 3 and experience of 5 years, she is permitted to access Patient_info according to Rule 5, but denied according to Rule 6, which causes a conflict. Without conflict resolution, in the graph database, there will be two arcs between the two nodes with contradict authorizations, as shown in Figure 10.

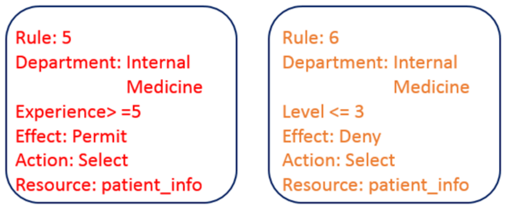


Figure 9: Conflict Rules

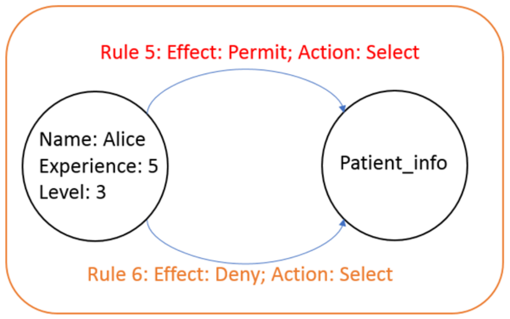

Rule 5: Effect: Permit; Action: Select
Name: Alice
Experience: 5
Level: 3
Patient_info
Rule 6: Effect: Deny; Action: Select

Figure 10: Graph Database Results

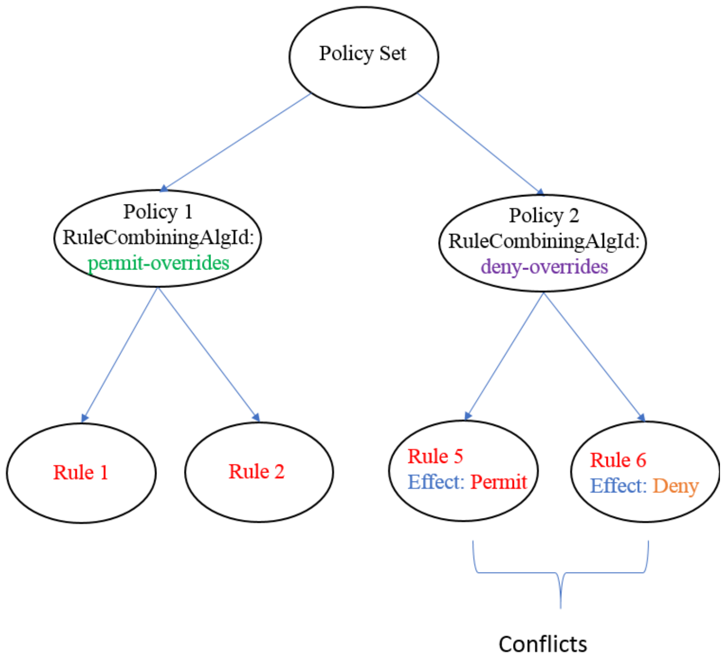


Figure 11: Conflict Resolution Case 1

XACML allows users to define how to handle conflicts by specifying combining algorithms. A combining algorithm defines which rules to select in the case of conflicts. At the policy level, it is called rule combining algorithm. At the policy set level, it is called policy combining algorithm. There are four combining algorithms: permit-overrides, deny-overrides, first-applicable, and only-one-applicable [1]. In the case of permit-overrides algorithm, if a single rule or policy is "Permit", the result is "Permit". Likewise, in the case of deny-overrides, if a single rule or policy is "Deny", then the result is a "Deny". Similar to the reasons listed in [1], our current implementation system handles permit-overrides and deny-overrides.

Our conflict resolver firstly handles conflicts within each policy. Next, the winners from each policy are compared at the policy set level. Resolving conflicts is explained in two different scenarios as follows.

Case 1: This case presents how to handle conflicts at the policy level. Conflicts are resolved according to a rule combining algorithm. For example, as shown in Figure 11, Rule 5 and Rule 6 are considered as conflicts, as shown in Figures 9 and 10. According to the combining algorithm, which is deny-overrides, Rule 6 with the effect of "Deny" will be selected at the policy level for Policy 2.

Case 2: This case presents how to resolve conflicts at both policy level and policy set level. As shown in Figure 12, in Policy 1, Rule 1 and Rule 2 are duplicates. One of them will be selected, say Rule 1. In Policy 2, Rule 4 has the effect of "Deny". Since the combing algorithm in Policy 2 is deny-overrides, Rule 4 is the winner. Next, all winners from each policy, i.e. Rule 1, Rule 4, and Rule 5, are further checked at the policy set level. Among these three rules, Rule 1 (from Policy 1) and the Rule 4 (from Policy 2) are conflicting. Rule 1 has the effect of "Permit". Comparing Rule 1 and Rule 4, Rule 1 is the winner, since the combining algorithm at the policy set level is permit-overrides. As a result, two rules will survive at the policy set level, which are Rule 1 and Rule 5.

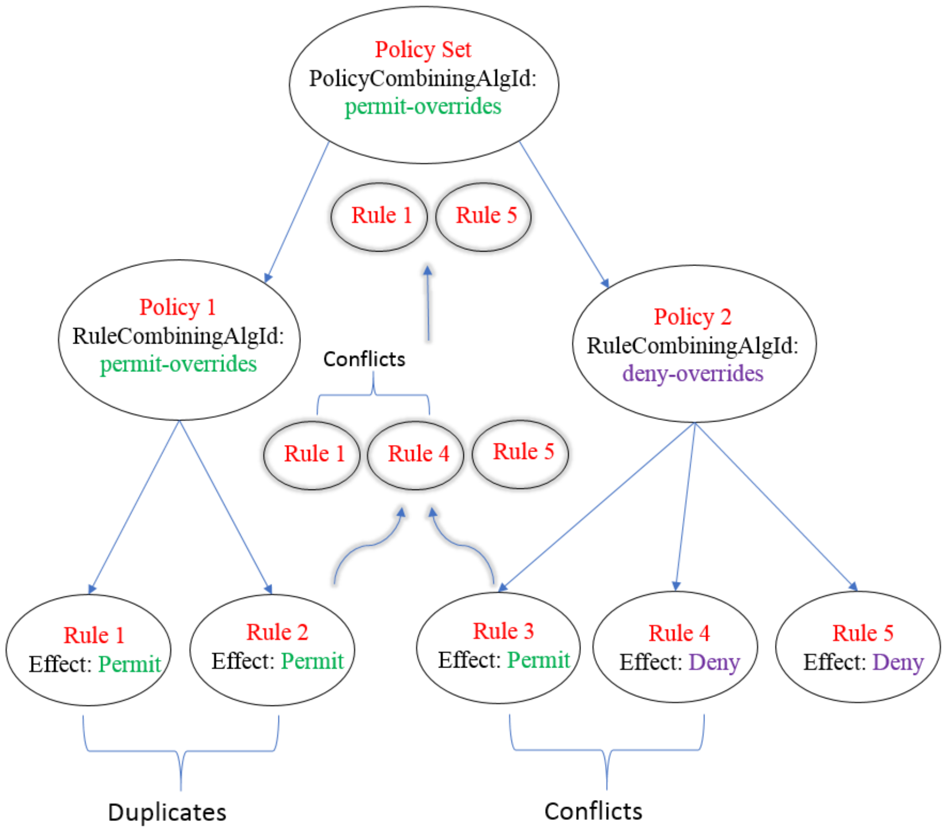


Figure 12: Conflict Resolution Case 2

5 Summary

There are different approaches to implement XACML standard. The research presented in this paper uses the graph database approach for XACML implementation. The current version of the project is implemented using Neo4j. The data is stored in Neo4j with users and resources as nodes, and access authorizations as arcs. The paper described how to process polices using the Policy Module, as well as how to handle requests using the Request/Response Module. Because of the way to authorize access by evaluating attribute values in attribute-based access control, conflict resolution is one critical problem to handle. This paper illustrated our approach and solution. One future direction of this research is to refine our system and performance evaluation.

References

[1] S. Jahid, C. A. Gunter, I. Hoque, H. Okhravi. MyABDAC: Compiling XACML Policies for Attribute-Based Database Access Control. In *proceedings of 1st ACM conference on Data and application security and privacy*, 2011, pages 97-108. New York, NY, USA

[2] Y. Jin, T. Sorley, J. Reyes. A Database-Oriented Approach for XACML Implementation on Role-Based Access Control. In *proceedings of the 22nd International Conference on Software and Data Engineering*, September 25-27, 2013, Los Angeles, CA, USA, pp. 55-59

[3] A.X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A Fast and Scalable XACML Policy Evaluation Engine. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems*, June 2-6, 2008, Annapolis, Maryland, USA.

[4] Neo4j Graph Platform. https://neo4j.com/

[5] Neo4j's Graph Query Language: An Introduction to Cypher. https://neo4j.com/developer/cypher-query-language/

[6] OASIS. XACML Version3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf

[7] S. O'Brien, Y. Jin, T. Sorley. Refinement of XACML Implementation on Role-Based Access Control. In the *proceedings of 27th International Conference on Computer Applications in Industry and Engineering*, October 13-15, 2014, New Orleans, Louisiana, USA, pp. 39-44

[8] SunXACML Implmentation. Sun Microsystems, Inc

[9] A. Wahane and Y. Jin. A Graph Database Approach for XACML Role-Based Access Control Implementation. In the *proceedings of 27th International Conference on Software Engineering and Data Engineering*, October 8-10, 2018, New Orleans, Louisiana, USA.

XACMLight. http://xacmllight.sourceforge.net/