

XACML Extension for Graphs: Flexible Authorization Policy Specification and Datastore-Independent Enforcement

Aya Mohamed^{1,2}^a, Dagmar Auer^{1,2}^b, Daniel Hofer^{1,2}^c and Josef Küng^{1,2}^d

¹*Institute of Application-oriented Knowledge Processing, Johannes Kepler University Linz, Linz, Austria*

²*LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Linz, Austria*

Keywords: Access Control, Authorization Policy, Graph-Structured Data, Graph Database, Cypher, Neo4j, XACML.

Abstract: The increasing use of graph-structured data for business- and privacy-critical applications requires sophisticated, flexible and fine-grained authorization and access control. Currently, role-based access control is supported in graph databases, where access to objects is restricted via roles. This does not take special properties of graphs into account, such as vertices and edges along the path between a given subject and resource. In our previous research iterations, we started to design an authorization policy language and access control model, which considers the specification of graph paths and enforces them in the multi-model database ArangoDB. Since this approach is promising to consider graph characteristics in data protection, we improve the language in this work to provide flexible path definitions and specifying edges as protected resources. Furthermore, we introduce a method for a datastore-independent policy enforcement. Besides discussing the latest work in our XACML4G model, which is an extension to the Extensible Access Control Markup Language (XACML), we demonstrate our prototypical implementation with a real case giving an outlook on performance.

1 INTRODUCTION

With the increasing use of graph databases for business- and privacy-critical applications, not only the continuous growth of data and its complexity must be considered but also advanced, flexible, and fine-grained authorization and access control. Access control protects assets and private information against unauthorized access by potentially malicious parties. Authorization is the process and result of specifying access rights in terms of who (subject) can perform what (action) on which resource (Jøsang, 2017). An authorization policy defines access rights in one or more sets of rules using some policy language. Existing policy languages do not yet consider graph-specific characteristics.

A graph is a set of vertices, which can be related to each other in pairs by edges. In graph databases, both vertices and edges are stored and accessed as entities. Thus, vertices can be also considered in the context of their relationships to other vertices, even over longer


paths, i.e. sequences of alternating vertices and edges.


Consider a knowledge graph with *data objects* and *tasks* as vertex entities. We need to describe a path in the authorization policy with constraints on the attributes of subject and resource *data object* vertices. Moreover, a *task* vertex has to exist somewhere along this path having a connecting edge with certain values for its attributes.


We worked on attribute-level path constraints in previous research iterations (see Section 3.3) resulting in initial versions of our model called *XACML for Graphs (XACML4G)*. The policy language and access control model support constraints on paths, but each element of the path must be described in detail. Furthermore, only graphs in ArangoDB are supported.


Our current research deals with the open issues from these earlier iterations as well as further input from related work. We contribute the following results to the XACML4G model and prototype:

- Flexible path specification in XACML4G authorization policies without defining every vertex and edge in the path pattern.
- Edges are also considered as resources.
- A datastore-independent enforcement model using a source-subset graph.

^a  <https://orcid.org/0000-0001-8972-6251>

^b  <https://orcid.org/0000-0001-5094-2248>

^c  <https://orcid.org/0000-0003-0310-1942>

^d  <https://orcid.org/0000-0002-9858-837X>

- Support path-related attributes in the XACML4G policy and request by extending an established XACML implementation.
- A proof of concept prototype of the extended XACML4G language and architecture, implementing flexible authorization policy specification and datastore-independent enforcement.
- A demo case to show the extended XACML4G policy definition and enforcement as well as handling XACML4G requests.

The rest of the paper is structured as follows. In Section 2, we give an overview of our research method and research questions. Section 3 describes relevant access control models, technologies, and results of our previous research iterations. The policy language and enforcement details of our current XACML4G extension are explained in Section 4. Section 5 gives details about our prototypical implementation and a demo case. In Section 6, we investigate the performance of the implemented prototype compared with XACML and our previous work. The paper concludes with a summary and an outlook on future work in Section 7.

2 RESEARCH METHOD

We follow a design science research (DSR) method (Hevner et al., 2004) and thus, focus on problem solving to enhance human knowledge by designing artifacts. The requirements for authorization and access control in the context of graph-structured data originate from a real-world problem in the domain of IT-supported knowledge work in a patent law firm (Hübscher et al., 2021).

Concepts, prototypes and knowledge are designed, developed and evaluated in an iterative research process in two complementary projects with partners from business and research. While the first two iterations were strongly driven by the needs of our business partner, we now generalize the concept to be flexible, adaptable and datastore-independent.

We discuss the results of our previous iterations, approaches and technologies, i.e., our DSR knowledge base, in Section 3. In the current research iteration, we focus on the following research questions:

- RQ1.** What are the main challenges for a flexible definition and datastore-independent enforcement of XACML4G path constraints?
- RQ2.** What are suitable concepts for designing the identified challenges?
- RQ3.** Can a prototype implementation of the concept be provided and applied to real-world cases?

To evaluate our design, we focus on analytical proofs and feedback circles with our project partners for *RQ1* and *RQ2*. Concerning *RQ3*, we implement a proof of concept prototype, showing its feasibility on a generated dataset from the patent and trademark prosecution domain (see Section 5). We overcome the limitations regarding flexible definition of path constraints and generally applicable enforcement.

3 RELATED WORK

Besides the related access control models to protect graph-structured data, we introduce the models and technological background as well as the main results of our previous research iterations.

3.1 Access Control Models

Current graph and multi-model database systems provide *role-based access control (RBAC)*, e.g., *Neo4j*¹, *Microsoft Azure CosmosDB*², and *ArangoDB*³. However, RBAC is not sufficient as it neither considers protecting data via restrictions on paths in the graph nor takes entity properties into account.

Protecting graph-structured data requires to constrain the path from the subject to the resource by content, not only by properties such as depth, type or trust level as in several relation-based access control (ReBAC) models (Fong, 2011; Cheng et al., 2016). ReBAC is based on evaluating relationships between subjects and resources. The lack of a common definition of ReBAC led to a number of domain-specific models with rather ad-hoc enforcement models and implementations. Clark et al. (2022) consider ReBAC policies as graph queries and formalize their ReBAC query language *ReLOG* according to the language features derived from comparing ReBAC models. Most ReLOG features are already supported in our previous XACML4G versions, such as querying basic graph patterns, mutual exclusion constraints, arbitrary path semantics, path negation, parameterized queries, and the any predicate. Path patterns do not need to describe the overall path from subject to resource in ReLOG, which is an open issue in previous XACML4G versions, but will be considered in the current research. ReLOG is based on *regular property-graph logic* and introduces custom functions to overcome the limited expressiveness of the policy language. Whenever the policy is changed, functions

¹Role-based access control in Neo4j enterprise edition

²Azure role-based access control in Azure Cosmos DB

³Access control in the ArangoGraph Insights Platform

need to be added or changed. But with ReLog, fine-grained access control and an implementation are still open issues. Furthermore, ReLog (like our previous works) does not consider edges as resources worth protecting like vertices, although both are equivalent elements in the graph model.

Neither RBAC nor comprehensive ReBAC models support fine-grained access control applying constraints on the entities to be protected. Attribute-based access control (ABAC) (Hu et al., 2017) overcomes this limitation. Constraints can be defined at the attribute level for subjects, resources, actions to be performed, and environment conditions. Unlike ReBAC, the ABAC model lacks the natural specification of relationships between subject and resource. The comparison of ABAC and ReBAC in Ahmed et al. (2017) shows that ABAC models are more expressive.

3.2 Models and Technologies

Braun et al. (2008) regard XML-based models, like XACML, as the closest to graph-related requirements. Thus, we rely on XACML as it is considered the defacto standard for managing and enforcing fine-grained privileges, e.g., Wu et al. (2006) and the PRIMA system (Lorch et al., 2003). As our current design relies on a graph database to enforce XACML4G policies, we further discuss the graph query language Cypher and the graph database Neo4j.

3.2.1 XACML

XACML is the abbreviation of *eXtensible Access Control Markup Language*. It is an OASIS approved standard for access control established in 2001. The policy language model of XACML is XML-based having the three main components: rule, policy, and policy set. Firstly, *rule* is the basic element having an effect (i.e., permit or deny) as well as an optional target and condition. A *target* is a combination of zero or more subjects, resources, actions, and environment attributes. A *policy* is comprised of zero or more rules, a rule combining algorithm, and a target. A *policy set* is a composite element consisting of a target, a policy combining algorithm besides zero or more policy sets and policies. A rule, policy, or policy set is applicable when its target attributes match those in the request.

XACML is not only a policy language, but also a processing model (i.e., architecture, workflow, and methodology) for evaluating access requests. The data flow between the XACML conceptual units is visualized in Figure 1. Additionally, XACML provides extension points for defining custom combining algorithms, attribute providers, policy providers, data types, and functions.

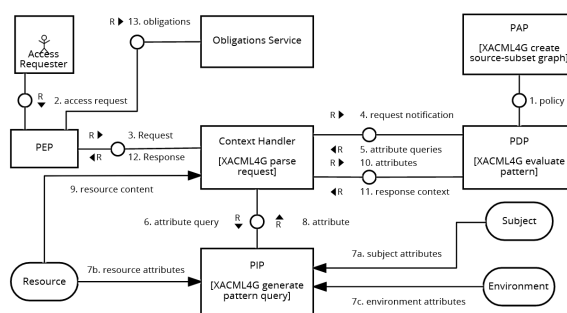


Figure 1: XACML reference architecture and extension.

The *policy administration point (PAP)* manages policies, which will be used in evaluating access requests, with respect to authoring and deployment (1). The *policy enforcement point (PEP)* receives the access request from the user (2), maps it to the XACML request native format and sends it to the context handler (3). Furthermore, the PEP fulfills obligations, i.e., operations carried out during the policy enforcement phase (13). The context handler converts access requests from the native format to the XACML canonical form (4) and vice versa for the response (12). It also acts as an intermediate entity between the *policy decision point (PDP)* and the *policy information point (PIP)*. The PDP requests subject, resource, action, environment, and other custom attributes from the context handler (5). The context handler requests the attributes from the PIP (6), retrieves them from the respective entities (7) and returns them to the context handler (8). The context handler further optionally includes the resource in the context (9). Finally, the results are sent to the PDP (10) for evaluating the policies and making authorization decisions (11).

3.2.2 Cypher Query Language

Several query languages are proposed for property graphs such as PGQL (van Rest et al., 2016), Gremlin⁴, Blueprints⁵, G-CORE (Angles et al., 2018), or Cypher⁶. Furthermore, the *Graph Query Language (GQL)*⁷ ISO standard, comparable to SQL for relational data, is in development.

Cypher is a declarative query language for property graphs and has a syntax inspired by SQL. It allows to easily express graph patterns as well as path queries. Cypher was originally created by Neo4j and contributed to the open-source project openCypher in 2015. It is not only used with Neo4j, but also with other graph databases (e.g., Amazon Neptune and

⁴<https://tinkerpop.apache.org/gremlin.html>
⁵<https://github.com/tinkerpop/blueprints/wiki>
⁶<https://opencypher.org/>
⁷<https://www.gqlstandards.org/>

SAP HANA Graph). Due to its expressiveness, flexibility, and significant contribution to GQL, we decided for Cypher to process and evaluate the patterns in XACML4G authorization policies.

3.2.3 Neo4j

Neo4j⁸ is a native graph database implementing the property graph model in Java. The data is structured in terms of nodes and relationships. It supports schema-free and schema-optional use and is currently available in the open source Community Edition (GPL v3) and the commercial Enterprise Edition. In over 20 years, they established their position as world market leader in graph databases⁹ with significant evolution and an active community.

The Neo4j database is directly accessed and queried using the declarative query language *Cypher*. Furthermore, it provides means to load data from different sources, which is needed for implementing our demo cases. Thus, we decided to use Neo4j to process and evaluate XACML4G policies.

3.3 Previous Results

In Mohamed et al. (2021a), we presented a preliminary approach to define and enforce graph-specific authorizations. We proposed a model for expressing fine-grained constraints on vertex and edge properties along the path between a subject and resource. We introduced a JSON-formatted authorization policy based on the XACML policy structure and provided a proprietary pattern enforcement. The concept and prototype are restricted to the multi-model database *ArangoDB*.

Our subsequent work (Mohamed et al., 2021b) proves the concept in the context of XACML. We provide a formal grammar for the extended XACML policy and request. Furthermore, the reference architecture of XACML is extended to enforce the newly introduced element (i.e., *pattern*) using the extensibility points of the standard functional components. This XACML extension has the expressiveness of standard XACML in addition to considering path constraints for graph-structured data. However, the complete path between a given subject and resource needs to be defined. Moreover, the policy is processed in advance to generate a query for each rule having a pattern as well as a XACML condition associated to the pattern identifier to evaluate the pattern in the request evaluation phase. This is because the pattern is an extension and cannot be evaluated directly by the XACML

model. Requests are also processed to extract the subject and resource from the path attributes to be used in the policy matching by XACML. The other attributes are used for the pattern evaluation. This model is based on ArangoDB along with its query language *AQL* and thus, needs further implementation to be used with other graph databases.

4 APPROACH

In this section, we illustrate the extended XACML4G policy and request language. We explain the proposed extensions of the XACML architecture, whether proprietary or using extensibility points. Moreover, we discuss the policy decision procedures on the conceptual level. We provide the source code of our prototype, an XML schema definition (XSD) of the new elements, a demo case, and the full version of this paper in our documentation¹⁰.

4.1 Challenges

According to the results of our previous research iterations and related work, we identify the following challenges:

- Flexible number of hops between two path vertices without specifying every path element.
- Pattern-related conditions (e.g., joining conditions between path elements).
- A datastore-independent enforcement model of XACML4G policies for property-graph compatible datastores.

4.2 Policy Language Extension

In the following, we explain the policy, rule, and request structure of XACML4G.

4.2.1 XACML4G Policy

The XACML4G policy is extended with a new element *Meta* to specify the entities of vertices and edges used in evaluating the policy. We manage the defined subset of the overall source data in an independent graph, which we call, according to its purpose, *source-subset graph*.

The meta element is composed of *Vertices* and *Edges* as XML tags. The *vertices* element lists at least one node label in the source graph using a tag called *VertexEntity*. The same applies to the relationships types, but using an *EdgeEntity* tag.

⁸<https://neo4j.com/docs/>

⁹<https://db-engines.com/en/ranking/graph+dbms>

¹⁰XACML extension for graphs documentation

4.2.2 XACML4G Rule

The XML schema of the XACML rule is extended by adding two elements, *Pattern* and *PatternCondition*. The pattern element is already introduced in our previous research iterations. It is structured as a recursive path consisting of a vertex, an edge, and either another vertex (i.e., the base case) or an entire path. Although the pattern definition does not change, the elements within the path are enhanced to support flexible patterns. Thus, there is no need to specify every vertex and edge along the path anymore.

The path vertex and edge elements express the pattern constraints by specifying the attributes and their values as a sequence of *AnyOf*, like in the XACML *target*. An optional attribute *Label* is added to specify the entity/collection to which the vertex belongs. We consider the label as part of the vertex definition, not a property to be matched against a value. The vertex has an attribute *Category* indicating whether it is a subject, a resource, or belongs to the path. The subject and resource categories are predefined by XACML while the path category is defined in XACML4G to handle the path vertices differently. An attribute representing the identifier of the vertex element is called *VertexId*. The identifiers are mainly used to join vertices or edges in the pattern condition element.

Constraints of the edge are structured like the vertex. The edge element is identified by an *EdgeId* attribute and can belong to the path or resource category only. The direction of the edge (e.g., inbound, outbound, or any) can be indicated by an attribute called *Direction*. *Length*, *MinLength*, and *MaxLength* are newly added attributes for pattern flexibility by specifying a range for some part of the path. An attribute called *Type* is also introduced to specify the edge type.

Pattern condition is another new element. It describes constraints and joining conditions that are related to the path elements of the pattern within the same rule. Elements of the pattern condition have the same structure as the XACML *Apply* element, but its *AttributeDesignator* is extended to include a *VertexId* or an *EdgeId* attribute representing the variable defined for a vertex or an edge in the rule pattern. Accordingly, the *category* property should be either *xacml4g:1.0:path-category:vertex* or *xacml4g:1.0:path-category:edge*.

Furthermore, the *FunctionId* attribute of the *Apply* should be one of our defined XACML4G functions. For now, we support conjunction, disjunction, comparison operators (i.e., =, !=, <, ≤, >, and ≥), and some string functions (e.g., *contains* and *starts with*). The function URI (e.g., *xacml4g:1.0:function:and*) is then translated to the corresponding operation in Cypher during the dynamic query generation in the

request evaluation phase.

4.2.3 XACML4G Request

The standard XACML access request consists of a sequence of attributes related to the subject, resource, and action. Each request attribute has an attribute id, a data type and a value. In Mohamed et al. (2021b), we extended the XACML request to differentiate between the attributes' types (i.e., action and path) without changing their structure. The path attributes are needed in matching requests against policies, especially subject and resource attributes, as well as during the pattern evaluation in the PIP. Therefore, we define a *path* category to which all vertices other than subject and resource belong.

In this work, we are extending the request with respect to language and its processing. The action and path elements are defined as a sequence of the standard XACML *Attributes* consisting of an *attribute* element having an identifier (i.e., *AttributeId*) as a property and an element for the value (i.e., *AttributeValue*). The definition of the *Attributes* element is extended to optionally include an attribute called *Type* to represent whether it is a vertex or an edge.

For the path attributes, we need to describe not only the category, entity, and value for a vertex (or a resource edge), but also to which property this value belongs. Therefore, we define the property name and value in the *AttributeValue* tag separated by a colon. We also use this format to specify the identifier name and value of a vertex or a resource edge in the request since the naming convention of the identifier property could vary from a data source to another.

Several components of the XACML architecture, i.e., context handler, PAP, PIP and PDP, are extended as illustrated in Figure 1 to handle the language-specific extensions according to the proposed enforcement concept.

4.3 Architecture Extension

To apply the XACML4G language extensions, the policy enforcement model is extended to deal with paths in the XACML4G requests and match them against patterns within rules of the XACML4G policy evaluating the pattern conditions as well. We propose a datastore-independent enforcement concept to evaluate the policy from various data sources without changing the core model. We introduce an optional property graph called *source-subset graph* containing the source data needed to evaluate the policy.

Firstly, the policy administration point (PAP) is extended to parse the policy files and extract the meta

element to create the source-subset graph independently of the request evaluation (refer to *XACML4G create source-subset graph* extension in Figure 1). The values of the *VertexEntity* and *EdgeEntity* elements represent the node labels and relationship types in a graph. The source-subset graph can be created from multiple data sources including flat files or database systems. The only requirement is that the data model can be mapped to the property graph model. Our model can be also configured to directly interact with the source datastore instead of the optional source-subset graph.

The context handler receiving the access requests is extended to parse the path and action attributes, which are used in policy matching by XACML as well as generating a Cypher pattern for the request path attributes by XACML4G. This is done by extending an established open source XACML implementation (see Section 5.1), to extract the path attributes and use them in the policy evaluation phase.

The policy information point (PIP) is extended to automatically handle the policy attributes related to the vertex label and edge type as well as the custom attributes representing the pattern identifier of the rule being evaluated. A Cypher pattern and a *where* statement are dynamically generated from the pattern and pattern condition elements to evaluate the pattern-related attributes (refer to *XACML4G generate pattern query* extension in Figure 1).

When the target of the policy is matched with the subject, resource, action, and environment attributes in the request, the XACML policy decision point (PDP) proceeds with evaluating the policy rules to determine the access decision. Before evaluating the conditions of the matched policy rules, an additional XACML condition is appended for the rules having a pattern. This condition is specific to evaluating the XACML4G pattern and its conditions (see *XACML4G evaluate pattern* in Figure 1). The condition evaluation is successful if the query returns a result (i.e., true). The query is generated in the PIP according to the pattern and its conditions within the rule. It is executed in a Neo4j database, which is the source datastore or the one having the source-subset graph. If the condition fails to evaluate due to the pattern query, an indeterminate decision is returned.

5 DEMONSTRATION CASE

We present the feasibility of our proposed approach by applying our implemented prototype to an access control case from the Knop-2D project.

5.1 Prototype Implementation

Our prototype is implemented using Java and Neo4j. XACML4G is not restricted to a particular database as source datastore. For all datastores except Neo4j, a *source-subset graph* must be created containing all authorization-relevant data. These data are specified in the meta element within the policy and can be retrieved from different datasources via special importer classes. This source-subset graph is optional if Neo4j is the source datastore. We currently provide importer classes for Neo4j and ArangoDB.

The prototype is based on the open-source XACML implementation *Balana*¹¹. Policies and requests are expressed in the XACML4G syntax. The request is evaluated by the Balana framework, which we extended to (1) parse the structured attributes (i.e., action and path) in XACML4G requests and (2) evaluate the XACML4G pattern (if exists) by adding a XACML condition dynamically when evaluating the rule(s) of the matched policy (or policies) against the access request. Our extensions address the open issues stated in Section 4.1 without affecting the overall XACML-specific procedures and results.

5.2 TEAM Model Case

In this demo case, we use just the instance model of the TEAM model (Hübscher et al., 2021). *DataObjects* vertices are related to each other via *dataObjectRelations* edges or to *tasks* via *accessRelations* or *taskDataRelations*. *AccessRelations* express user-to-task relationships, whereas *taskDataRelations* link *tasks* to *dataObjects* or *dataObjectRelations*. The generated dataset contains 11,982 *dataObjects*, 2,559 *tasks*, 3,165 *accessRelations*, 13,271 *dataObjectRelations*, and 53,246 *taskDataRelations*.

Our authorization scenario requires a certain relationship between a user and a resource via a task: "As a user, I can access a data object, if I am allocated to or work/have worked on a task with a path to the resource.". The rule contains: (1) *Subject*: user, (2) *Resource*: data object, (3) *Action*: access, (4) *Pattern*: user → task →⁺ data object, and (5) *Effect*: permit.

The policy defines the vertex and edge collections of the graph model in the *meta* tag to create the source-subset graph. The *user* is a data object having an attribute *typeCode* with value *pmUser*. The resource can be any data object. The subject and resource are the start and end vertices of the pattern. The task is specified as an intermediate vertex connected to the user via an *access relation* edge and a

¹¹<https://github.com/wso2/balana>

maximum of two hops from the resource. The pattern has attribute-based constraints and can specify the range of a sub-path. We define a variable for the *accessRelations* edge and specify the constraint worksOn or allocates for its *typeKind* property.

We specify attributes for subject, resource, and path vertices by name and value separated by a colon (e.g.: `_key:1196742142`). If the request is matched with a policy having a pattern, a XACML condition is added to its rule. The extended PIP generates the pattern query using the pattern and pattern condition in the rule besides the path attributes in the request. It looks for the intersection of the rule pattern of the matched policy (p1) and request pattern generated from the path attributes (p2) as shown in Listing 1.

Listing 1: Pattern query example.

```
MATCH p1 = (s:dataObjects(typeCode:"pmUser"))-
[el:accessRelations]->(:tasks)-[*..2]-(:dataObjects)
MATCH p2 = ({_key:"1196741133"})-[]-
({_key:"1196741778"})-[]-({_key:"1196742142"})
WHERE e1.typeKind="worksOn" OR e1.typeKind="allocates"
AND ALL (x IN nodes(p2) WHERE x IN nodes(p1)) AND ALL
(x IN relationships(p2) WHERE x IN relationships(p1))
RETURN p1 IS NOT NULL AS result
```

The request path is a sequence of vertices only and is matched by its attributes. The direction of the edges in the rule pattern is applied to the request when evaluating the pattern query. The query is successfully evaluated, as we check for an existing path in the dataset satisfying the pattern constraints. The request decision is *permit* according to the rule effect.

6 PRELIMINARY EVALUATION

We provide preliminary performance measurements evaluating access requests with different path lengths. Three prototypes are compared for the same scenario based on the TEAM model. *Prototype 1* is based on the standard XACML and ArangoDB. It has limited expressiveness and scalability because XACML does not support paths. Only subject, object, and action can be defined within the policy and request. Moreover, we manually add custom attributes in the policies and statically write the respective queries to be evaluated in the decision-making phase. Hence, each change in the authorization requirements demands adaptations in the policy as well as the implementation. *Prototype 2* is the initial version of XACML4G (see Section 3.3). These two prototypes are already investigated in Mohamed et al. (2021b), but with a more trivial evaluation design. *Prototype 3* is the latest XACML4G discussed in this paper (see Section 5.1).

The evaluation was performed offline on an Intel(R) Core(TM) i7-6500U CPU @ 2.50 GHz with 24 GB RAM. We investigated the execution time of

100 consecutive requests from processing till receiving the access decision. We performed the experiment three times for each prototype and calculated the average. The results are plotted in Figure 2.

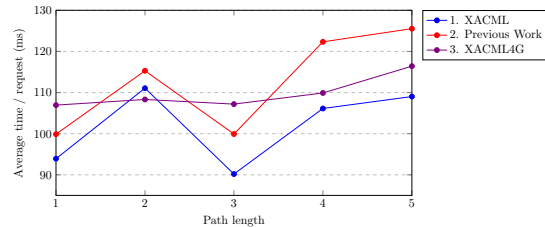


Figure 2: Average request evaluation time for the XACML, previous work, and latest implementation prototypes.

The plot for the prototypes 1 and 2 look like those in Mohamed et al. (2021b), where we proved that the introduced overhead in the extension is constant regardless of the path length. This overhead almost doubled as indicated in the difference between the plots for the path length 4 and 5 vs. shorter paths. This is due to the additional pattern conditions to join specific elements along the path. From the implementation perspective, the two major differences between the prototype presented in this paper and the other ones are the underlying graph database and the pattern evaluation within the reference architecture of XACML. Previously, our approach relied on directly connecting to the source graph database and our implementation was based on ArangoDB and its declarative query language *AQL*. In the latest implementation, our approach is independent of the source datastore. It is currently based on *Cypher* and an embedded Neo4j database within the XACML4G prototype for evaluating patterns and their conditions.

As can be observed in Figure 2, our latest work is only slightly impacted with the increased path length. This is most likely due to the better stability of Neo4j compared with ArangoDB.

7 CONCLUSIONS

We present our latest enhancements to XACML4G, which are flexible constraints on paths, edges as resources, and datastore-independent enforcement.

The main challenges are flexible path specification and an enforcement for property-graph compatible datastores (*RQ1*). Path features include pattern-related conditions and a flexible number of hops between two vertices to no longer define the path completely. To address these challenges in (*RQ2*), we rely on a declarative graph query language supporting the required characteristics (e.g., pattern match-

ing on paths, flexible path length, or incomplete path specifications) and a property graph holding all authorization-relevant data, which we call source-subset graph. The proof-of-concept prototype implements the latest language and architecture of XACML4G and a case for a real knowledge graph (RQ3). The XML schemas are further extended to define the authorization-relevant data, support flexible path specification in the policy, and specify edges as resources. To enforce the XACML4G language, extensibility points in the PIP and proprietary extensions of the XACML architecture (i.e., context handler, PAP, and PDP) are implemented. The prototype extends the open source XACML implementation *Balana* and uses *Neo4j* along with *Cypher* for a datastore-independent enforcement. No more preprocessing of policies and requests is required. Compared to our previous work and a statically implemented XACML prototype, our current prototype has better performance and stability in evaluating paths with different lengths. Additionally, the current approach no longer introduces constant overhead.

This work highlighted further challenges. Patterns are now evaluated within the XACML model as conditions, but no pattern-related errors can be detected. Moreover, multiple labels on vertices and edges have to be considered to match with real-world graph models. The performance comparison can be improved by excluding influencing factors, such as different graph database systems for policy enforcement.

ACKNOWLEDGMENTS

The research reported in this paper has been partly supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria. The work was also funded within the FFG BRIDGE project KnoP-2D (grant no. 871299).

REFERENCES

- Ahmed, T., Sandhu, R., and Park, J. (2017). Classifying and comparing attribute-based and relationship-based access control. In *Proceedings of the 7th Conference on Data and Application Security and Privacy, CODASPY '17*, page 59–70, New York, USA. ACM.
- Angles, R., Arenas, M., Barcelo, P., Boncz, P., Fletcher, G., Gutierrez, C., Lindaaker, T., Paradies, M., Planktikow, S., Sequeda, J., van Rest, O., and Voigt, H. (2018). G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, page 1421–1432, New York, NY, USA. ACM.
- Braun, U., Shinnar, A., and Seltzer, M. (2008). Securing provenance. In *Proceedings of the 3rd Conference on Hot Topics in Security, HOTSEC'08*, USA. USENIX Association.
- Cheng, Y., Park, J., and Sandhu, R. (2016). An access control model for online social networks using user-to-user relationships. *IEEE Transactions on Dependable and Secure Computing*, 13(4):424–436.
- Clark, S., Yakovets, N., Fletcher, G., and Zannone, N. (2022). Relog: A unified framework for relationship-based access control over graph databases. In *Data and Applications Security and Privacy XXXVI: 36th Annual IFIP WG 11.3 Conference, DBSec 2022, Newark, NJ, USA, July 18–20, 2022, Proceedings*, page 303–315, Berlin, Heidelberg. Springer-Verlag.
- Fong, P. W. (2011). Relationship-based access control: Protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy, CODASPY '11*, page 191–202, New York, NY, USA. ACM.
- Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Q*, 28(1):75–105.
- Hu, V. C., Ferraiolo, D. F., Chandramouli, R., and Kuhn, D. R. (2017). *Attribute-Based Access Control*. Artech House information security and privacy series. Artech House, Boston.
- Hübscher, G., Geist, V., Auer, D., Ekelhart, A., Mayer, R., Nadschläger, S., and Küng, J. (2021). Graph-based managing and mining of processes and data in the domain of intellectual property. *Information Systems*, 106:101844.
- Jøsang, A. (2017). A consistent definition of authorization. In Livraga, G. and Mitchell, C., editors, *Security and Trust Management*, pages 134–144, Cham. Springer International Publishing.
- Lorch, M., Adams, D. B., Kafura, D., Koneni, M., Rathi, A., and Shah, S. (2003). The prima system for privilege management, authorization and enforcement in grid environments. In *First Latin American Web Congress*, pages 109–116, Phoenix, AZ, USA. IEEE.
- Mohamed, A., Auer, D., Hofer, D., and Küng, J. (2021a). Extended authorization policy for graph-structured data. *SN Computer Science*, 2(5):1–18.
- Mohamed, A., Auer, D., Hofer, D., and Küng, J. (2021b). Extended xacml language and architecture for access control in graph-structured data. In *The 23rd International Conference on Information Integration and Web Intelligence, iiWAS2021*, page 367–374, New York, NY, USA. Association for Computing Machinery.
- van Rest, O., Hong, S., Kim, J., Meng, X., and Chafi, H. (2016). Pqql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, GRADES '16*, New York, NY, USA. ACM.
- Wu, J., Leangsuksun, C. B., Rampure, V., and Ong, H. (2006). Policy-based access control framework for grid computing. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, volume 1, pages 391–394, Singapore. IEEE.