Comparison of Access Control Approaches for Graph-Structured Data

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, ABAC, ReBAC, Graph-Structured Data, Property Graph.

Abstract: Access control is the enforcement of the authorization policy, which defines subjects, resources, and access rights. Graph-structured data requires advanced, flexible, and fine-grained access control due to its complex structure as sequences of alternating vertices and edges. Several research works focus on protecting property graph-structured data, enforcing fine-grained access control, and proving the feasibility and applicability of their concept. However, they differ conceptually and technically. To gain a profound overview of the current state of research, we study works from our systematic literature review on authorization and access control for different database models in addition to recent ones. Based on defined criteria, we exclude research works which do not protect graph-structured data, have coarse-grained approaches, consider models other than the property graph model, or have no proof-of-concept implementation. The latest version of the remaining works are discussed in detail in terms of their access control approach as well as authorization policy definition and enforcement. Finally, we analyze the strengths and limitations of the selected works and provide a comparison with respect to different aspects, including the base access control model, open/closed policy, negative permission support, and datastore-independent enforcement.

1 INTRODUCTION

Access control ensures data security by protecting assets and private information against unauthorized access. It refers to enforcing authorization policies, which specify access rights in terms of accessing subject, requested resource and performed action.

In graphs, data are structured as vertices connected by edges to represent the relationships between objects. Vertices and edges are stored as entities in graph databases, optionally including properties as key-value pairs. Fine-grained access control for graph-structured data refers to protecting vertices and edges at the attribute level. Established graph databases currently support role-based access control (RBAC), while the latest research works address relationships in graph-structured data, including attributes on vertices and edges.

With this paper, we aim to discuss these latest research works to get a profound overview of current research approaches. The results provide guidance for our further research. We select the following access control approaches for graph-structured data from our systematic literature review (SLR) on access control for different database models, and include further recent works. Rizvi and Fong (2018) extended the relationship-based access control model (ReBAC) to support attributes in policy specification and enforcement. Mohamed et al. (2023a) presented a flexible, fine-grained authorization policy specification for graph-structured data and datastore-independent enforcement. Furthermore, Hofer et al. (2023a,b) proposed an approach to enforce fine-grained access control by rewriting Cypher queries using an Abstract Syntax Tree (AST). Bereksi Reguig et al. (2024) recently introduced an approach based on the Neo4j access control model to apply attribute-based access control (ABAC). We study the works on conceptual and technical level by answering the following research questions:

RQ1. What are the current access control approaches in the context of graph-structured data?

RQ2. How are the authorization policies defined and enforced in each work?

RQ3. How are the feasibility and applicability of the proposed approaches proved?

[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

Mohamed, A., Auer, D., Hofer, D. and Küng, J.
Comparison of Access Control Approaches for Graph-Structured Data.
DOI: 10.5220/0012861500003767

In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 576-583
ISBN: 978-989-758-709-2; ISSN: 2184-7711

RQ4. What are the strengths and limitations for each approach?

The paper is structured as follows. In Section 2, we describe our method for selecting related works. In Section 3 to 6, we discuss the conceptual approach, definition and enforcement of authorization policy, and implementation details for the selected works respectively. We compare and analyze the approaches in Section 7. The paper concludes with a summary and an outlook on future work in Section 8.

2 SELECTION METHOD

In our SLR of authorization and access control for different database models (Mohamed et al., 2023b), we identified a list of research works related to the graph database model. We additionally include recent works as listed in Table 1. For each work, we indicate whether or not it is included according to our selection criteria: protecting property graph-structured data, fine-grained authorization policy, and applied in graph datastore.

Table 1: Research works in access control for graphs.

Research works	Protect property graph	Fine-grained	Applied
Rizvi and Fong (2018)	✓	✓	✓
Morgado et al. (2018)	×	×	✓
Bertolissi et al. (2019)	×	✓	✓
Jin and Kaja (2019)	×	✓	✓
Mohamed et al. (2020)	✓	✓	✓
Valzelli. et al. (2020)	✓	×	✓
Chabin et al. (2021)	×[1]	✓[2]	✓
Clark et al. (2022)	✓	✓[2]	×
Hofer et al. (2023a)	✓	✓	✓
Bereksi Reguig et al. (2024)	✓	✓	✓

[1] Based on the RDF graph model [2] For vertices but not edges

Morgado et al. (2018) present a model-based approach using metadata with authorization rules to control access in applications that use a graph database. Their security model uses a predefined schema for the vertices as a plugin in Neo4j. This approach is not fine-grained, as it only considers protection on vertex level. The focus is on the meta model and architecture rather than applying it to protect graph-structured data.

Bertolissi et al. (2019) introduce an access control framework for secure data fusion in cooperative systems. Authorization policies are represented in a provenance graph, which relates artefacts (vertices) to depict their provenance through a system. They provide the formal semantics of the proposed language and use the extensible access control markup language (XACML) to demonstrate provenance-based access constraints in ABAC policies. Jin and Kaja (2019) use graphs to represent the XACML authorization policy language model. An authorization policy graph is constructed by parsing XACML policy files. In addition duplicates and conflicts are handled before generating the Cypher query statements. XACML requests are processed as queries to Neo4j, which stores the policy graph. Both approaches do not consider protecting graph-structured data.

Chabin et al. (2021) propose an access control system for graph-based models with schema constraints, authorization rules to protect the data and user context rules. To enforce the constraints, modules for rewriting, planning and executing queries in parallel are provided. Roles can have fine-grained access rules on vertices, and clearance levels can be associated to roles and resources. However, each user has only a single role. Furthermore, only type and direction, but no attributes can be specified for edges in rules. This work is not considered in our comparison as it is not designed to deal with property graphs, but rather the resource description framework (RDF) graph model.

Valzelli. et al. (2020) introduce a property graph model, which combines discretionary access control, mandatory access control and RBAC to protect knowledge graphs. Positive and negative authorizations can be specified using authorization edges between subjects and resources. RDF concepts are translated into a property graph query. It is implemented using Tinkerpop. We excluded RBAC approaches because it is not fine-grained and supported in many graph databases, such as Neo4j, ArangoDB and Azure Cosmos DB.

Clark et al. (2022) define the formal language and unified framework ReLOG to encode ReBAC policies, which can be viewed as graph queries. Although the work is a fine-grained ReBAC extension, we excluded it because only the theoretical feasibility of the approach is investigated. Neither a concrete implementation nor demo cases are provided.

In the following, we discuss the latest research results of the remaining works in detail, which are Rizvi and Fong (2018), Mohamed et al. (2023a), Hofer et al. (2023a) and Bereksi Reguig et al. (2024).

3 ACCESS CONTROL APPROACH

We identified two main categories of access control approaches: permit-deny access and filtered results. With *permit-deny* access is requested given the subjects, resources, actions, and other optional conditions. The request is checked against the authorization policy and a decision is returned indicating whether the access is authorized or not. This can be used

to control the type of actions to be performed on a specific resource. The *filtered results* category refers to returning authorized data. The result of unauthorized access requests is empty. It is commonly used in databases, where views or query rewriting are applied to filter the result according to the policy and retrieve a list of authorized resources at once. Although there are currently several works that focus on protecting data in graph databases, they have different objectives and use cases. In the rest of this section, we discuss the conceptual approach for each work with respect to their access control model and approach category.

AReBAC. The *Attribute-supporting ReBAC Model (AReBAC)* (Rizvi and Fong, 2018) is an extension to *ReBAC* (Bruns et al., 2012), considering attributes on vertices and edges along relationships between subjects and resources to provide fine-grained access control in property graphs. It relies on a proprietary graph pattern format to specify queries and access control policies. A subset of Cypher, called Nano-Cypher, is defined as an equivalent language. An authorization request is a pair (m,s), with a requested method m (i.e., a database query with additions) and subject s. Authorization policies are defined in graph patterns. For a request (m,s) by a user s, the matching policies for the user's query m are identified and integrated with the query to filter the results.

XACML4G. The work of Mohamed et al. (2020) aims to specify and enforce fine-grained, dynamic authorizations for graph-structured data. There can be several paths from a subject to a resource, but not all of them are authorized. A subject can only access a protected resource through authorized paths considering the context of resources, as it can reveal confidential information. The ABAC model is extended to specify flexible graph patterns as a path from the subject to the resource, defining fine-grained authorization constraints for vertices and edges and pattern-related conditions to join and compare path elements. The XACML policy language model and reference architecture are extended as a proof-of-concept for the proposed approach, which is called *XACML for graphs (XACML4G)*, to consider specification and evaluation of graph patterns. XACML4G is a permit-deny access model since the result is a decision.

GQRA[1]**.** Hofer et al. (2023a,b) present a graph query rewriting-based approach (GQRA) to enforce fine-grained authorizations dynamically for Cypher queries. Insecure queries are rewritten to include authorization-specific filters. Thus, only authorized data is returned. The query rewriting algorithm uses an AST for the source Cypher query. Based on the defined policy, authorization-specific filters are added to the syntax tree and the query is reformulated accordingly. This approach implicitly enforces ABAC in graph databases, especially Cypher-based datastores, beyond their supported access control features.

ABAC for Neo4j[1]**.** Bereksi Reguig et al. (2024) extend the RBAC model in Neo4j to support attributes. In Neo4j, the enterprise edition supports granting and denying traverse, read and match privileges to node labels and relationship types along with their properties. However, the authorization policy rules are associated to roles and apply to all nodes or relationships with the specified label or type. Therefore, they extend this access control model to support fine-grained conditions to filter out further nodes or relationships from being traversed or read.

To answer RQ1, we provided an overview and classified the selected works into two main categories (i.e., permit-deny access and filtered results).

4 POLICY DEFINITION

In this section, we explain the authorization policy language model for the selected works along with an overview of the access request and response (if any).

AReBAC. Rizvi and Fong (2018) define authorization policy in the context of AReBAC as "... a graph pattern used for specifying authorization requirements for accessing resources.". It can be considered as a graph restricted by mutual exclusion constraints and attribute requirements. AReBAC assumes an underlying open policy. Database queries are categorized. Each category has at most one authorization policy. Categories are used for the request and policy matching. With graph patterns, a language for declarative authorization policy specification is provided, as an easier to use alternative to logic formulas and regular expressions (Rizvi and Fong, 2018). A graph pattern contains a set of vertices, a set of edges, mutual exclusion constraints on vertices, a set of attributes of vertices and edges, a category, a mapping of selected vertices and the return values. The graph pattern also contains placeholders which will be replaced during request processing and policy matching. To specify an authorization policy, a Nano-Cypher query with an associated category and mapping can also be used instead of a graph pattern.

XACML4G. A preliminary policy specification is defined in a proprietary JSON format in Mohamed et al. (2020). Then, the XACML4G elements are defined using XSD and applied in XACML (Mohamed et al., 2023a). The meta element defines vertex labels and edge types of the source graph relevant for policy

[1]We defined this abbrev. to use throughout our paper.

evaluation. The pattern is a path with composite structure, consisting of a vertex, an edge, and either another vertex (i.e., the base case) or an entire path. Each vertex has an identifier, a label, its own category URI and a sequence of attributes. Edges additionally include an optional direction, a type instead of a label and a range (i.e., minimum and maximum length) for flexible patterns. The pattern condition element is structured like the XACML condition, but its identifier attribute references a vertex or an edge variable in the pattern. Furthermore, the function attribute should be one of the supported functions. Access request is extended to additionally include path-related attributes, specifying vertices and edges as resources. Based on the input request, the policy is matched and evaluated, returning an access decision.

GQRA. The authorization policy model defined in Hofer et al. (2023a) is influenced by XACML. Authorization requirements are described in a policy having a set of rules. The policy specifies the path from the subject to the resource. The matching elements can be referenced in conditions and filter templates, which require runtime information. Each rule references an element of the policy pattern and specifies one or more boolean combined conditions. A condition checks whether *filter* and *return* properties are satisfied by any element of the policy pattern. The filter indicates whether the element is filtered according to its label/type, its properties or not at all, while return states if it is included in the return clause of the query (i.e., directly or aggregated).

ABAC for Neo4j. The policy rules in Bereksi Reguig et al. (2024) are expressed like in Neo4j with an introduced `WHERE` statement to specify one or more conditions. Each condition consists of an attribute, an operator or a built-in Cypher function, and a value. The attribute belongs to an entity in the graph, user or context. Thus, the policy extension is fine-grained at the vertex and edge attributes. Currently, conditions are only supported with the *traverse* privilege. The *read* rules are specified without conditions to avoid circular permissions. The *match* is not directly supported, but can be expressed by combining traverse and read.

To summarize, AReBAC, GQRA and ABAC for Neo4j define their policies as a query or filter template to restrict the result set. AReBAC uses a category with the query to match it with the authorization policy. XACML4G extends the XACML structure to define paths in the graph. XACML4G and ABAC for Neo4j are based on established policy formats such as XACML and the Neo4j policy definition, while AReBAC and GQRA define their own proprietary formats.

5 POLICY ENFORCEMENT

We now present the policy enforcement approach for each of the selected works, including processing, matching and conflict resolution (if applicable).

AReBAC. Rizvi and Fong (2018) propose an enforcement based on query rewriting. Nano-Cypher queries are not directly executed in Neo4j, but the evaluation rather relies on the internal graph pattern format and the evaluation algorithm *GP-Eval*. When a user sends an authorization request, the query is translated to a graph pattern. The policy is matched using the method's category. Categories can be organized in a hierarchy, which defines a refinement relationship, e.g., $c_1 \geq c_2$. Accordingly, the policy for methods with category c_1 must be at least as restrictive as the one for methods with category c_2. All graph patterns along the hierarchy are combined into a single graph pattern. The integrated policy and the query are merged resulting in a single graph pattern. Rizvi and Fong (2018) call this process *weaving* and not *query rewriting*. The integrated graph pattern is evaluated by *GP-Eval*, which only returns authorized data. As the AReBAC approach relies on its graph pattern format, it is datastore-independent, at least at its core. The database is accessed during the evaluation of the graph pattern to retrieve candidates for the result set.

XACML4G. In Mohamed et al. (2023a), the XACML conceptual components are extended to match the request path with the patterns in policy rules and evaluate the pattern conditions. To allow for a source datastore-independent policy evaluation, a subset of the overall source data is managed in an independent graph, which is called source-subset graph. This graph is optionally used to evaluate the policy without relying on a specific datastore and can be constructed from multiple data sources. The policy administration point (PAP) is extended to process the XACML4G policy and create the source-subset graph according to the specified entities in the meta element. In the extended context handler, XACML4G requests are parsed to extract the path attributes. Then, the extended policy decision point (PDP) adds a specific condition for evaluating the pattern if it exists in the matched policy rule. Based on the pattern and its conditions, a Cypher pattern and a `WHERE` statement are dynamically generated and the query is executed to evaluate path constraints as an extension in the policy information point (PIP). Conflicts are resolved in XACML at the level of policies and rules using custom or predefined combining algorithms, such as first applicable and (ordered) deny/permit overrides.

GQRA. Hofer et al. (2023a,b) enforces fine-grained authorizations by rewriting Cypher queries at run-

time. Firstly, the query is processed by mapping each element in the query to the corresponding one in the policy based on the pattern structure and the element label(s). For that, the pattern is mapped to a set of paths, each one represented as a tuple with start vertex, and optional edge and end vertex. The resulting comparable structure of the query and policy patterns are needed for policy matching. A policy is applicable if each path tuple of the insecure query matches one of its rule patterns and satisfies the rule's conditions. A simplified AST is built, excluding syntactical details as well as omitting `CREATE` and `WITH` clauses. It describes the semantics of the query. Filters of the matched policy rules are added to the `WHERE` part of the AST. Nodes without labels could cause ambiguity when multiple rules are matched. Thus, an optimization approach is needed to avoid overlapping filters or an empty result, due to non-matching constraints. Since only positive permissions are supported, no conflicts occur. The extended AST is written to a Cypher statement and executed in the database to retrieve the authorized result set.

ABAC for Neo4j. Bereksi Reguig et al. (2024) proposed an algorithm called *SafeCypher* to rewrite a user query according to the defined authorization policy along with the user role(s) and return a safe one (if applicable). First, the authorization graph is constructed such that each policy component (i.e., role, privilege, permission and entity) is represented as a vertex connected by directed edges, e.g., user role to privilege (i.e., traverse or read), privilege to permission, and permission to entity (i.e., node label, relationship type and attribute). A variable is assigned to each node and relationship in the source query and the query elements (e.g., `MATCH`, `WHERE` and `RETURN` statements) are extracted. In the case of a nested query, the subqueries are separately rewritten and the results are appended to the main query. Then, the corresponding access conditions are added to the `WHERE` statement and the attributes being returned are checked. Conflicts are resolved using the authorization policy graph using the algorithm deny-override like in Neo4j. Finally, the safe query is optimized before execution.

We answered RQ2 in Section 4 and 5 by discussing the policy definition and enforcement mechanism for the selected works on the conceptual level. In the following section, we present the technical details for each work and the provided use cases (if any).

6 IMPLEMENTATION

In this section, we address how the proposed access control approaches are feasible and applicable (refer to RQ3). For each work, we demonstrate which frameworks and graph databases have been used in addition to where in the architecture it has been implemented, e.g., within the database or an independent layer. Furthermore, we give an overview of the evaluation methods and results.

AReBAC. AReBAC is a proprietary Java implementation and uses Neo4j[2]. Graph pattern is an internal representation of queries and policies. Concerning the architecture, access control is implemented between the application and the database. Rizvi and Fong (2018) use the *Census-Income (KDD) Dataset* from the UCI Machine Learning Repository[3] to demonstrate the feasibility and performance of AReBAC. The GP-Eval algorithm implementations provide significantly better performance than Neo4j's Cypher evaluation engine, but the performance of semantically equivalent Nano-Cypher queries may differ.

XACML4G. Technically, XACML is an authorization service between the datasource and the application. Thus, XACML4G (Mohamed et al., 2023a) is considered as an independent layer. The approach is implemented in Java as a source-code extension to Balana[4], an open-source framework based on Sun's XACML implementation. Furthermore, the XACML4G policies and requests are validated using the XML schema for the language extensions. The optional source-subset graph is created and stored in an embedded Neo4j database. Currently, two Java classes are implemented to import graph data for Neo4j and ArangoDB. Access requests are parsed to extract relevant attributes for policy matching and pattern evaluation. Then, the PDP evaluates the matched policy rules to determine the access decision, including the XACML4G pattern and its conditions. A Cypher query is generated from the pattern constraints in the policy and path attributes in the extended request. The pattern evaluation is successful if the query returns a result. Two demo cases are presented with various access control scenarios and data models in different graph database systems. Paths with different lengths (1 to 5) are evaluated. In contrast to previous results, the latest XACML4G prototypical implementation showed better performance and the constant overhead due to policy processing is eliminated. The overall overhead is reduced from ≈ 25 (excluding processing overhead) to 21 ms.

GQRA. The query rewriting approach in Hofer et al. (2023a,b) is implemented in Java as an extension to the *Neo4j Object Graph Mapper (OGM)*. The class *Session* is extended to enforce authorizations not only

[2]https://github.com/szrrizvi/arebac/

[3]https://archive.ics.uci.edu/

[4]https://github.com/wso2/balana

when executing Cypher queries, but also in loading, saving and deleting objects. A query parser is built from a subset of Cypher's grammar using ANTLR. The function `getPaths` matches the query pattern with those of the policy. Authorization-related filters are added to the query and placeholders from the templates are replaced with runtime values. The resulting secure query is executed, and authorized resources are returned. The prototypical implementation is applied in the research project *SyMSpace*[5]. Authorization rules are defined in a configuration class. According to the graph model's structure, permissions can be assigned directly to users or via roles. Experiments with representative queries show that the average time needed for query rewriting is ≈ 0.2 ms.

ABAC for Neo4j. Bereksi Reguig et al. (2024) implemented their proposed concept in Java within Neo4j. Their Cypher query rewriting algorithm is executed within the Neo4j browser. The defined authorization policy is processed from its textual representation to an access graph and stored in Neo4j. User-defined procedures and functions are invoked from Cypher to extend Neo4j using customized Java code. The evaluation with respect to efficiency and scalability uses the *Stack Exchange* dataset. The experiments show that introducing conditions in authorization rules has minor performance overhead. It scales well with large datasets in addition to complex Cypher queries and authorization policies.

Although all selected works provide proof-of-concept implementations, they are still research results. AReBAC is an independent framework with a proprietary implementation for specifying authorization policies in Nano-Cypher and enforcing them using their GP-Eval algorithm. XACML4G and GQRA are implemented as extensions to Balana and OGM, respectively, to prove the feasibility of the concepts without being limited to these frameworks. Finally, ABAC for Neo4j is implemented using custom procedures, which can be invoked in Cypher and perform operations on the database directly.

7 DISCUSSION

After discussing the selected approaches, we compare them according to the following criteria. The results are summarized in Table 2.

Access Control Approach. Most of the selected approaches rely on *filtered results* to perform access control close to the database layer, by reducing the result set to only permitted data. Therefore, the same access request by different users may return different result sets, depending on their authorizations. XACML4G belongs to the classic *permit-deny* access, where the access request is checked and a decision is returned.

Table 2: Comparison of the selected works.

Criteria	AReBAC	XACML4G	GQRA	ABAC for Neo4j
Access control approach	filtered results	permit-deny access	filtered results	filtered results
Base AC model	ReBAC	ABAC	ABAC	RBAC
Open policy	✓	×	✓	✓
Negative permissions	×	✓	×	✓
Runtime policy processing	✓	✓	✓	×
Datastore-independent	✓	✓	✓[1]	×

[1] Cypher-based datastores only

Base Access Control Model. Fine-grained access control in the context of graphs is considered in all approaches, but with different access control models as a basis. Rizvi et al. (2015) already did research in ReBAC for relational databases, but without considering attributes. Even though ReBAC concepts can be mapped to the graph model, AReBAC was developed to further support ABAC. The base model of XACML4G and GQRA is ABAC, which allows flexible, fine-grained authorization policies, but does not consider restrictions on paths. The XACML4G policy structure, contains components to define pattern constraints and meta data for policy evaluation. GQRA supports path-specific requirements in their filter templates. ABAC for Neo4j in contrast is based on RBAC like access control in the Neo4j Enterprise edition, but is extended to support fine-grained conditions.

Open/Closed Policy. All filtered results approaches, i.e., AReBAC, GQRA, and ABAC for Neo4j rely on an open policy, which authorizes subjects by default(Samarati and de Vimercati, 2001). Authorizations limit the result. XACML4G assumes a closed policy, i.e., only access with positive permission.

Positive and/or Negative Permission Support. AReBAC and GQRA policies define filters to specify the permitted result set. However, XACML4G and ABAC for Neo4j policies must explicitly define, whether matching the policy conditions results is granting or denying access.

Runtime Policy Processing. This aspect considers if policies need processing before evaluation. With ABAC for Neo4j, policies are defined in text and then transformed into an access graph as a pre-processing step for query rewriting. All other approaches evaluate policies directly with each authorization request, even though, pre-processing could enhance performance, e.g., translating AReBAC policies defined in Nano-Cypher to graph patterns in advance.

[5]https://web.symspace.lcm.at

Datastore-Independent Enforcement. The AReBAC concept is datastore independent, as the internal representation of queries and policies is graph patterns. Thus, any graph pattern-equivalent language could be used. However, to replace the underlying Neo4j database, an abstraction layer for the database access is needed. The query rewriting approach in Hofer et al. (2023a) can be applied in Cypher-based datastores, while XACML4G is dynamic, applies external authorization, and deals with property graph-compatible datastores. In contrast, the work of Bereksi Reguig et al. (2024) is datastore specific, i.e., restricted to Neo4j. Although they overcome the limitations of the Neo4j access control model which is not fine-grained and the rules are statically defined in the system database, the authorizations are still linked to roles and are not flexibly assigned to users according to attribute-based criteria.

Each of the approaches has its strengths and weaknesses, and their relevance heavily depends on the specific requirements. We summarize the main results to answer RQ4 in the following. The three filtered results approaches (AReBAC, GQRA, and ABAC for Neo4j) share many of the conceptual details such as an open access control approach, but differ in the chosen basic access control model. ABAC for Neo4j supports negative permissions like the authorization model of Neo4j. Furthermore, it follows the Neo4j policy definition format unlike AReBAC and GQRA, which have proprietary formats. GQRA is restricted to Cypher-based datastores as it enforces the policies by rewriting Cypher queries, whereas AReBAC "weaves" the database query and policies specified in the proprietary graph pattern format, and is thus conceptually independent of the Cypher language. XACML4G in contrast differs in fundamental criteria, as it is a *permit-deny* access control approach with a closed policy. However, like GQRA, it is based on ABAC. Furthermore, XACML4G is datastore-independent, as it can be enforced on any property graph-compatible datastore.

8 CONCLUSIONS

To compare and analyze fine-grained access control approaches to protect property graph-structured data, we selected Rizvi and Fong (2018), Mohamed et al. (2020), Hofer et al. (2023a), and Bereksi Reguig et al. (2024). The proposed access control approaches belong to the *permit-deny access* or *filtered results* category (RQ1). We answered RQ2 and RQ3 by discussing each work in terms of access control model, authorization policy definition, policy enforcement approach and implementation details. In AReBAC, the policy is either specified using their graph pattern representation or a Nano-Cypher statement, which is translated to a graph pattern. The enforcement algorithm evaluates graph patterns with a minimal number of database accesses and amount of data retrieved. The concept is implemented as an independent layer on top of Neo4j. XACML4G defines graph-specific authorization constraints on vertices and edges as a recursively structured path pattern. Conditions for comparing and joining pattern elements can be specified and edges are also considered as resources. Regarding policy enforcement, dynamic queries are generated for the matched policies, taking additional attribute values from the request, and then executed in the source database or the one storing the source-subset graph. The approach is implemented as an extension to the XACML policy language and architecture. The authorization policy in GQRA is a set of rules with paths having placeholders to be replaced with runtime values. To enforce the policy, an AST equivalent to the source query is generated for adding authorization-specific conditions, and then translated back to a Cypher query. OGM is extended to apply the proposed query rewriting approach. In ABAC for Neo4j, the policy has a textual representation based on the Neo4j access control model, but stored in a graph after processing. For the policy enforcement, conditions for each clause are appended to the query, which is optimized after resolving conflicts. A user-defined procedure is implemented in Neo4j taking the source query as input to be rewritten and executed.

For the last research question (RQ4), we defined additional criteria to distinguish between the selected works, which already satisfy our initial selection criteria (i.e., protect property graph-structured data, fine-grained, and applied in graph datastores). Only XACML4G is designed differently with respect to the *permit-deny* access control approach and closed policy, while the rest belongs to the *filtered results* category with open policy. Concerning the base access control model, XACML4G and GQRA rely on ABAC, while AReBAC and ABAC for Neo4j rely on ReBAC and RBAC respectively. XACML4G and ABAC for Neo4j support negative permissions, unlike the other works. ABAC for Neo4j is the only work that supports neither datastore-independent policy enforcement nor runtime policy processing, as the approach is specific to Neo4j and a policy graph needs to be constructed in advance. The enforcement approach in XACML4G can work with property-graph compatible datastores, whereas GQRA is considered for Cypher-based datastores.

The current work has highlighted further ques-

tions concerning the differences and use cases of the two categories for access control approaches. For example, the *filtered results* category is typically used in retrieving authorized resources based on the defined policy. The *permit-deny access* category is less commonly used in access control for databases, but still considered in applications to allow authorized users to perform specific actions (e.g., edit or delete) on particular resources and vice versa. A detailed comparison of the prototypes (if available) on a common scenario is considered in the future. Furthermore, we plan to add more aspects to our comparison and include other works focusing on protecting property graph-structured data that have been excluded due to coarse-grained access control or a lack of application.

ACKNOWLEDGEMENTS

This work has been partly supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria and the Linz Institute of Technology. This work has also been supported by the COMET-K2 Center of the Linz Center of Mechatronics (LCM) funded by the Austrian federal government and the federal state of Upper Austria.

REFERENCES

Bereksi Reguig, A. A., Mahfoud, H., and Imine, A. (2024). Towards an effective attribute-based access control model for neo4j. In Mosbah, M., Kechadi, T., Bellatreche, L., and Gargouri, F., editors, *Model and Data Engineering*, pages 352–366. Springer, Cham.

Bertolissi, C., den Hartog, J., and Zannone, N. (2019). Using provenance for secure data fusion in cooperative systems. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, SACMAT '19, page 185–194. ACM.

Bruns, G., Fong, P. W. L., Siahaan, I., and Huth, M. (2012). Relationship-based access control. In Bertino, E. and Sandhu, R. S., editors, *Proceedings of the 2nd ACM conference on Data and Application Security and Privacy*, pages 117–124. ACM.

Chabin, J., Ciferri, C. D. A., Halfeld-Ferrari, M., Hara, C. S., and Penteado, R. R. M. (2021). Role-based access control on graph databases. In Bureš, T., Dondi, R., Gamper, J., Guerrini, G., Jurdziński, T., Pahl, C., Sikora, F., and Wong, P. W., editors, *SOFSEM 2021: Theory and Practice of Computer Science*, pages 519–534. Springer, Cham.

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*, page 303–315. Springer, Cham.

Hofer, D., Mohamed, A., Auer, D., Nadschläger, S., and Küng, J. (2023a). Rewriting graph-db queries to enforce attribute-based access control. In Strauss, C., Amagasa, T., Kotsis, G., Tjoa, A. M., and Khalil, I., editors, *Database and Expert Systems Applications*, pages 431–436. Springer, Cham.

Hofer, D., Mohamed, A., Nadschläger, S., and Auer, D. (2023b). An intermediate representation for rewriting cypher queries. In Kotsis, G., Tjoa, A. M., Khalil, I., Moser, B., Mashkoor, A., Sametinger, J., and Khan, M., editors, *Database and Expert Systems Applications - DEXA 2023 Workshops*, pages 86–90. Springer.

Jin, Y. and Kaja, K. (2019). Xacml implementation based on graph databases. In *Proceedings of the 34th International Conference on Computers and Their Applications*, pages 65–74.

Mohamed, A., Auer, D., Hofer, D., and Küng, J. (2020). Authorization policy extension for graph databases. In Dang, T. K., Küng, J., Takizawa, M., and Chung, T. M., editors, *Future Data and Security Engineering*, pages 47–66. Springer, Cham.

Mohamed, A., Auer, D., Hofer, D., and Küng, J. (2023a). Xacml extension for graphs: Flexible authorization policy specification and datastore-independent enforcement. In *Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT*, pages 442–449. INSTICC, SciTePress.

Mohamed, A., Auer, D., Hofer, D., and Küng, J. (2023b). A systematic literature review of authorization and access control requirements and current state of the art for different database models. *International Journal of Web Information Systems*.

Morgado, C., Busichia Baioco, G., Basso, T., and Moraes, R. (2018). A security model for access control in graph-oriented databases. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 135–142.

Rizvi, S. Z. R. and Fong, P. W. L. (2018). Efficient authorization of graph database queries in an attribute-supporting rebac model. In Zhao, Z., Ahn, G.-J., Krishnan, R., and Ghinita, G., editors, *CODASPY'18*, pages 204–211. ACM.

Rizvi, S. Z. R., Fong, P. W. L., Crampton, J., and Sellwood, J. (2015). Relationship-based access control for an open-source medical records system. In Weippl, E., Kerschbaum, F., and Lee, A. J., editors, *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pages 113–124. ACM.

Samarati, P. and de Vimercati, S. C. (2001). Access control: Policies, models, and mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, pages 137–196. Springer Berlin.

Valzelli., M., Maurino., A., and Palmonari., M. (2020). A fine-grained access control model for knowledge graphs. In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications - SECRYPT*, pages 595–601. INSTICC, SciTePress.