







Rewriting Graph-DB Queries to Enforce Attribute-Based Access Control

Daniel Hofer^{1,2} , Aya Mohamed^{1,2} , Dagmar Auer^{1,2} ,
Stefan Nadschläger¹, and Josef Küng^{1,2} 

¹ Institute for Application-oriented Knowledge Processing (FAW),
Johannes Kepler University (JKU) Linz, Linz, Austria
{daniel.hofer, aya.mohamed, dagmar.auer,
stefan.nadschlaeger, josef.kueng}@jku.at

² LIT Secure and Correct Systems Lab, Linz Institute of Technology (LIT),
Johannes Kepler University (JKU) Linz, Linz, Austria

Abstract. To provide Attribute-Based Access Control (ABAC) in a data-store, we can either rely on built-in features or, especially if they are not present, implement access control as a service (ACaaS) on top of the database. We address the latter, in particular for graph databases, by rewriting queries which are violating access control conditions. We intercept the insecure queries right before sending them to the database to add additional filters. Thus, the database returns only authorized data and implicitly enforces ABAC beyond its own access control features. Our contributions are an authorization policy model influenced by XACML and a query rewriting algorithm for enforcing the defined authorizations with respect to this model. Our concept is application- and database-independent and operates on simple freely formulated queries, i.e. the queries do not have to follow a predefined structure. A proof-of-concept prototype has been implemented for Neo4j and its query language Cypher.

Keywords: query rewriting · attribute-based access control (ABAC) · graph databases · database security · Cypher

1 Introduction

To enforce access control on a database with limited or even no access control features, like the community version of Neo4j, we have various options. The approach we chose is rewriting *insecure* queries before they are handed over to the database as *secure* queries [1], including authorization-specific filters. Our approach even supports attribute-based access control (ABAC) [9] by operating on data stored in the database. We already motivated this approach in our previous work [6]. However, our current work does not rely on a predefined query structure, but can handle freely formulated queries.

Authorization requirements are expressed in terms of rules in the authorization policy, referencing *filter templates* to be used in the query rewriting. A *filter*

template defines authorization-specific constraints to be added to the insecure query. Graph database query languages like Cypher distinguish between nodes and relationships. For this work, we refer to both using the term *element*. To rewrite the insecure query applying authorization-specific constraints, we consider the following research questions:

- RQ1 Which elements of a query influence the result?
- RQ2 What information must be provided in the authorization policy?
- RQ3 How can we find mappings between a policy and a query?
- RQ4 How do we apply access control filter templates on queries?

Our contributions are (1) identifying the influencing elements and how they impact the query result in Sect. 3, (2) a policy model influenced by XACML as a policy having a set of rules with conditions and references to filter templates in Sect. 4, (3) a query rewriting approach to extend the insecure query with filters encoding authorization requirements in Sect. 5, and (4) a proof-of-concept prototype¹ using Cypher and a preliminary evaluation in Sect. 6. Related works and a summary including an outlook on future work are provided in Sects. 2 and 7 respectively.

2 Related Work

The idea of protecting data by query rewriting is influenced by Browder et. al. and their work about per-user views in Oracle databases [3]. Another influence comes from Bogaerts et. al. [2] as they propose entity-based access control, taking not only attributes but also the relations between entities into account. While their focus is on relational databases, we primarily consider graph databases and thus attributes on nodes and edges. The dynamic rewriting approach was already proposed by Jarman et. al. [7], however, on relational databases and role-based access control. Our policy model is highly influenced by XACML (Ramli et. al. [10]), although we reduced the features to a subset suitable for our requirements. Colombo et. al. proposed an approach similar to ours in [4], as they generate authorized views to replace the original collection in the query. However, their focus is on document-oriented stores with focus on IoT data analysis. Access control by query rewriting for RDF and SPARQL was also proposed by Kirrane in [8]. A slightly different approach is presented by Shay et. al. [11] which checks queries against a policy and blocks them altogether if necessary. The current work is also based on our previous work, especially [5] for query parsing and modification and [9] for XACML policies for graphs.

3 Relevant Information in the Insecure Query

To answer RQ1, we start with checking the elements of the query pattern and identify the relevant elements influencing the query result. The pattern for example in the Cypher query `MATCH (a:L1)-[c]->(b:L2) WHERE a.id=8 RETURN`

¹ <https://github.com/jku-lit-scs/CypherRewritingCore>.

b” is “(a:L1)-[c]->(b:L2)”. While a confidential node in the *RETURN* clause of a query clearly reveals information, other cases are less obvious. For example, we have a graph database with information about students and their grades. A node stores all student data and links to a node with the student’s grade for a certain exam. To protect grades, we block returning the grade-nodes. However, a malicious user could return a student’s node and include a *WHERE* clause filtering only for a name (which is not confidential) and a specific grade. By only returning the data-node, no access violation is detected, but it implicitly confirms the *guessed* grade. Therefore, an element which has a filter applied might still lead to information leaks although it is not directly returned. On the other hand aggregating functions (e.g. average) prevent access to individual elements (e.g., a student’s grade). Thus, we check the combination of filter and return status for each element (see example in Table 1). The filter status is (1) filtered or (2) unfiltered and the return status is (1) aggregated, (2) direct value or (3) not included in the return clause.

Table 1. Influencing factors in **MATCH** (a:L1)-[c]->(b:L2) **WHERE** a.id=8 **RETURN** b.

| Element in pattern | Filter | Return | Influencing |
|--------------------|--------|--------------------|-------------|
| a | yes | no | yes |
| b | no | yes (direct value) | yes |
| c | no | no | no |

4 Policy Model

The purpose of the policy is to specify all authorization-relevant information. A policy P describes a pattern of elements E (i.e., nodes and relationships) and a set of rules R . The function $\Phi(e_{policy}, e_{query}) \rightarrow \{true, false\}$ decides whether an element of the policy pattern (e_{policy}) can be mapped to an element of the insecure query pattern (e_{query}). Let one policy be:

$$\begin{aligned}
 P &= (E, R, \Phi) \\
 E &= \langle e_1, e_2, \dots, e_n \rangle \\
 R &= (e, C, f) \\
 C &= \{c_1, c_2, \dots, c_n\}
 \end{aligned}$$

Each rule $r \in R$ references a single element of the policy pattern ($e \in E$) and specifies one or more boolean combined conditions C on the pattern elements and references a filter template f to be applied to e . A condition $c \in C$ checks whether filter and return properties (cp. Sect. 3) are satisfied by any element of the policy pattern $e' \in E$.

Filter templates F are used to exclude unauthorized results in the secure query. They define authorization-relevant constraints to be added to the insecure query. We define a filter template f with placeholders for runtime-specific information as follows:

$$\begin{aligned} F &= \{f_1, f_2, \dots, f_n\} \\ f &= (t, A) \\ A &= \langle a_1, a_2, \dots, a_n \rangle \end{aligned}$$

Every filter template $f \in F$ includes a query fragment t containing placeholders. For each placeholder in t , its kind a (e.g., `ruleElement` or `username`) is given in A . The kind `ruleElement` indicates that the placeholder stands for the element in the rule which references this filter template.

5 Query Processing

For a policy and its rules to be applicable, each element defined in the policy e_{policy} is mapped to an equivalent one in the query e_{query} based on its labels and pattern structure. To find a mapping (cp. RQ3), we define a function `getPaths` returning a set of paths from the pattern. Each path consists of a start node, a relationship and an end node ($e_{start}, e_{relationship}, e_{end}$). The relationship and end node can be empty if the start node is isolated.

$$\begin{aligned} &getPaths(E) \rightarrow E^* \\ E^* &= \{(e_{start}, e_{relationship}, e_{end}), \dots\} \end{aligned}$$

This step converts the patterns of policy and query into a common and comparable structure. We search for mappings using the function $map(e_{policy}) \rightarrow e_{query}$:

$$\begin{aligned} map(e_{policy}) \rightarrow e_{query} &\Leftrightarrow \forall (a, b, c) \in E_{policy}^* \exists (x, y, z) \in E_{query}^* : \\ &\Phi(a, x) \wedge \Phi(b, y) \wedge \Phi(c, z) \wedge (a = e_{policy} \wedge x = e_{query}) \wedge \\ &((b = \emptyset \wedge c = \emptyset) \vee (b = e_{policy} \wedge y = e_{query} \wedge c = e_{policy} \wedge z = e_{query})) \end{aligned}$$

The overall mapping is valid if (1) the path elements of the policy and the insecure query are successfully mapped using the function Φ (e.g., $\Phi(a, x)$), (2) the start nodes are matched, and (3) the relationships and end nodes are either empty or matched. Accordingly, we evaluate all conditions C in all rules R for a policy P . We generate a set S of 2-tuples (e_i, f_i) denoting an element from the query e_i and a filter template f_i to be applied on the insecure query q .

$$\begin{aligned} S &= \{(e_i, f_i) \mid \exists (E, R, \Phi) \in P, (e, C, f) \in R, e_i \in q, f_i \in F \forall c \in C : \\ &\Gamma(q, c) \wedge map(e) = e_i \wedge f_i = f\} \end{aligned}$$

The function $\Gamma(q, c) \rightarrow \{true, false\}$ checks whether a condition c in the rule's conditions C is satisfied by the insecure query q . Further, the element from the rule e must map to the element in the insecure query e_i and the applied filter template f_i is the same as the one f in the rule. To apply the filters of the matched rules on the insecure query (RQ4), we use the following function:

$$\Xi(q, S) \rightarrow q'$$

It takes an insecure query q and for each assignment $(e_i, f_i) \in S$, it instantiates $f_i \rightarrow f_{i_q}$ according to Section 4. This f_{i_q} can then be added to e_i or it extends existing filters using boolean *AND*. With all filters in place, we have rewritten an insecure query q to a secure version q' .

6 Evaluation

We evaluate our query rewriting approach by implementing a proof-of-concept prototype² using Cypher, ANTLR, Spring Boot and Kotlin. We rewrite the insecure Cypher query based on the specified policy. The secure query and information about the applied rules are returned. In our prototypical implementation, we only support reading queries with one *MATCH* clause. In experiments with a set of queries, we tested all currently supported features and visually confirmed that the filters were applied correctly. However, in our prototype we did not consider potential vulnerabilities or attack vectors not addressed by ABAC.

When measuring the performance of the query rewriting (no database access), we noticed the standard deviation to be higher than the average rewriting time (≈ 0.2 ms on a HP ELITEBOOK 850 G6 with 32 GB, CPU i7-8665U). Therefore, we assume the performance overhead to be negligible.

7 Conclusion

In this paper, we proposed a runtime rewriting approach for freely formulated graph-DB queries to enforce ABAC independent of the underlying database and application. First, we defined how various elements of a query contribute to its result (RQ1). We introduced the strategy of categorizing the elements based on whether they have a filter applied and how they are used for returning data. Next, we introduced a policy model encoding our authorization requirements. Then, we formally defined a policy model including a filter template. The policy consists of a pattern and rules to decide if access control constraints apply to an element of the insecure query and which filter template to use (RQ2). The policy pattern is a sequence of elements, which is used in the query processing.

The policy and the insecure query are processed by first splitting their patterns into tuples representing either paths or isolated nodes. Accordingly, we mapped the policy elements with their respective ones in the insecure query

² <https://github.com/jku-lit-scs/CypherRewritingCore>.

(RQ3). A mapping is valid if each path tuple of the insecure query matches one of the policy. In this case, if all conditions of a rule are successfully evaluated, its filter template is instantiated replacing its placeholders with runtime values from the insecure query and/or user information. The last step of query processing is enhancing the insecure query with these access control filters (RQ4).

As we only consider one policy, we plan to support policy sets according to the XACML policy language model in the future. This further demands for combining algorithms. Additionally, we currently support reading queries only with one `MATCH` clause. Thus, we not only need to increase the supported number of `MATCH` clauses, but also the types of supported queries. This could be added using additional conditions or dedicated rules for reading and writing access. Above all, intensive evaluation is needed especially with complex authorization policies and large graph models. Finally, we need to identify possible potential security vulnerabilities.

Acknowledgements. This research has been partly supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria and 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

1. Bao, H.N.P., Clavel, M.: A model-driven approach for enforcing fine-grained access control for SQL queries. *SN Comput. Sci.* **2**(5), 370 (2021)
2. Bogaerts, J., Decat, M., Lagaisse, B., Joosen, W.: Entity-based access control: supporting more expressive access control policies. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp. 291–300 (2015)
3. Browder, K., Davidson, M.A.: The virtual private database in oracle9ir2. Oracle Tech. White Paper, Oracle Corporat. **500**(280) (2002)
4. Colombo, P., Ferrari, E.: Fine-grained access control within NoSQL document-oriented datastores. *Data Sci. Eng.* **1**(3), 127–138 (2016)
5. Hofer, D., Mohamed, A., Nadschläger, S., Auer, D.: An intermediate representation for rewriting cypher queries. In: Submitted to Workshop (2023)
6. Hofer, D., Nadschläger, S., Mohamed, A., Küng, J.: Extending authorization capabilities of object relational/graph mappers by request manipulation. In: Database and Expert Systems Applications: 33rd International Conference, DEXA 2022, Vienna, Austria, 22–24 August 2022, Proceedings, Part II, vol. 13427, pp. 71–83. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-12426-6_6
7. Jarman, J., McCart, J.A., Berndt, D., Ligatti, J., et al.: A dynamic query-rewriting mechanism for role-based access control in databases (2008)
8. Kirrane, S.: Linked data with access control. Diss. National University of Ireland, Galway (2015)
9. Mohamed, A., Auer, D., Hofer, D., Küng, J.: Extended authorization policy for graph-structured data. *SN Comput. Sci.* **2**(5), 351 (2021)
10. Ramli, C.D.P.K., Nielson, H.R., Nielson, F.: The logic of XACML. *Sci. Comput. Program.* **83**, 80–105 (2014)
11. Shay, R., Blumenthal, U., Gadepally, V., Hamlin, A., Mitchell, J.D., Cunningham, R.K.: Don't even ask: database access control through query control. *ACM SIGMOD Rec.* **47**(3), 17–22 (2019)