SN Computer Science (2021) 2:370
https://doi.org/10.1007/s42979-021-00712-7

ORIGINAL RESEARCH

A Model-Driven Approach for Enforcing Fine-Grained Access Control for SQL Queries

Hoang Nguyen Phuoc Bao[1] · **Manuel Clavel**[1]

Received: 24 March 2021 / Accepted: 17 May 2021 / Published online: 7 July 2021

Abstract

In this paper, we propose a novel, model-driven approach for enforcing fine-grained access control (FGAC) policies when executing SQL queries. More concretely, we define a function SecQuery() that, given an FGAC policy $\mathcal{S}$ and an SQL select-statement q, generates an SQL stored-procedure $\ulcorner$Sec-Query$(\mathcal{S}, q)\urcorner$, such that: if a user u is authorized, according to $\mathcal{S}$, to execute q, then calling $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner(u)$ returns the same result that when u executes q; otherwise, if the user u is not authorized, according to $\mathcal{S}$, to execute q, then calling $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner(u)$ signals an error. The stored-procedure SecQuery$(\mathcal{S}, q)$ implements the appropriate FGAC authorization-checks for executing the query q, according to the policy $\mathcal{S}$. As expected, the execution of the query q takes less time than calling the stored-procedure $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner$. Moreover, evaluating the (sub)-queries corresponding to authorization-checks will take (more or less) time, depending on the "complexity" of the underlying policies. To illustrate this performance-issue, we have included in this paper some experimental results regarding the performance overhead incurred by executing the (secured) stored-procedure corresponding to (unsecured) queries. Finally, we have implemented our model-driven approach for enforcing FGAC policies for SQL queries in an open-source project, called SQL Security Injector (SQLSI).

Keywords Secured SQL queries · Fine-grained access control · Model-driven security

Introduction

Model-driven security (MDS) [1, 2] is a specialization of model-driven engineering for developing secure systems. In MDS, designers specify system models along with their security requirements, and use tools to generate security-related artifacts, such as access control infrastructures. SecureUML [8] is 'de facto' modeling language used in MDS for specifying fine-grained access control policies (FGAC). These are policies that depend not only on static information, namely the assignments of users and permissions to roles, but also on dynamic information, namely the satisfaction of authorization constraints by the current state of the system. The structure query language (SQL) [15] is a special-purpose programming language designed for managing data in relational database management systems (RDBMS). Its scope includes data insert, query, update, and delete, and schema creation and modification. None of the major commercial RDBMS currently supports FGAC policies in a "native" way.

In [12], we have proposed a model-based characterization of FGAC authorization for SQL queries. In our proposal, FGAC policies are modeled using a "dialect" of SecureUML. The challenge we address now is how to effectively enforce FGAC policies when executing SQL queries. Our solution consists of defining a function SecQuery() that, given an FGAC policy $\mathcal{S}$ and an SQL select-statement q, generates an SQL stored-procedure $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner$, such that: if a user u is authorized, according to $\mathcal{S}$, to execute q, then calling $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner(u)$ returns the same result that when u executes q; otherwise, if the user u is not authorized, according to $\mathcal{S}$, to execute q, then calling $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner(u)$ signals an error. Our solution consists of defining a function SecQuery() that, given a SecureUML model $\mathcal{S}$ and an SQL select-statement q, generates an SQL

This article is part of the topical collection "Future Data and Security Engineering 2020" guest edited by Tran Khanh Dang.

✉ Manuel Clavel
manuel.clavel@vgu.edu.vn

Hoang Nguyen Phuoc Bao
ngpbhoang1406@gmail.com

[1] Vietnamese-German University, Thu Dau Mot, Vietnam

stored-procedure, such that: if a user is authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure returns the same result that executing q; otherwise, if a user is not authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure signals an error. Basically, the stored-procedure SecQuery$(\mathcal{S}, q)$ implements the appropriate FGAC authorization-checks for executing the query q, according to the policy $\mathcal{S}$. Informally, we can say that $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner$ is the secure version of the query q with respect to the FGAC policy $\mathcal{S}$, or that $\ulcorner$SecQuery$(\mathcal{S}, q)\urcorner$ secures the query q with respect to the FGAC policy $\mathcal{S}$.

As mentioned before, FGAC policies depend on the satisfaction of authorization constraint by the current state of the system. Thus, executing FGAC-related authorization-checks causes, unavoidably, a performance overhead at run-time, which will be greater or lesser depending on the "complexity" of the underlying policy. We report on some preliminary experiments that illustrate well this performance-issue.

Organization

The rest of the paper is organized as follows. In the next section, we motivate with examples some of the problems we aim to address when securing SQL queries. In the following section, we introduce our modeling language for specifying FGAC policies. This section provides background material, which is needed for the rest of the paper. In the next two consecutive sections, we propose our novel model-driven approach for enforcing FGAC policies for SQL queries, and illustrate it with non-trivial examples. In the following section, we report on experimental results regarding the performance overhead incurred by our approach. In the next section, we present SQLSI, a Java application that implements our solution. Finally, in the last sections, we discuss related and future work.

Motivation

Informally, enforcing an FGAC policy when executing an SQL query means guaranteeing that the execution of the query does not leak confidential information. Interestingly, this implies much more than simply checking that the final result satisfies the applicable FGAC policy, Indeed, a clever attacker can devise a query, such that the simple fact that a final result is obtained will reveal by itself some additional information, which may be confidential. To illustrate this problem, we introduce a simple example of information leakage resulting from allowing users to execute "unsecured" queries.

Let us consider a simple database `UniversityDB` containing three tables: `Lecturer`, for representing lecturers; `Student`, for representing students; and `Enrollment`, for representing the links between the students and their lecturers.

Query#1	`SELECT email FROM Lecturer WHERE Lecturer_id = 'Huong';`

Query#2	`SELECT DISTINCT email FROM Lecturer`
	`JOIN (SELECT * FROM Enrollment`
	`WHERE students = 'Thanh'`
	`AND lecturers = 'Huong' ) AS TEMP`
	`ON TEMP.lecturers = Lecturer_id;`

Query#3	`SELECT DISTINCT email FROM Lecturer`
	`JOIN (SELECT e1.lecturers as lecturers`
	`FROM (SELECT * FROM Enrollment`
	`WHERE lecturers = 'Huong' ) AS e1`
	`JOIN (SELECT * FROM Enrollment`
	`WHERE lecturers = 'Manuel' ) AS e2`
	`ON e1.students = e2.students ) AS TEMP`
	`ON TEMP.lecturers = Lecturer_id;`

Fig. 1 Example. Queries 1–3

The tables `Lecturer` and `Student` have the columns `Lecturer_id` and `Student_id` as their respective primary keys. The table `Enrollment` has two columns, namely, `lecturers` and `students`, which are foreign keys, associated, respectively, to `Lecturer_id` and `Student_id`. Finally, both tables `Lecturer` and `Student` have columns `name` and `email`.

Consider now the select-statements in Fig. 1. For the sake of this example, suppose that, for a given scenario, the three of statements return the same final result, namely, a non-empty string, representing an email that is not confidential. On a closer examination, however, we can realize that, for each of these statements, the final result is revealing additional information, which may happen to be confidential. In particular

- `Query#1` reveals that the resulting email belongs to Huong.
- `Query#2` reveals not only that the resulting email belongs to Huong, but also that Thanh is enrolled in a course that Huong is teaching.
- `Query#3` reveals that the email belongs to Huong, and that Huong and Manuel are "colleagues", in the sense that there some students for whom both Huong and Manuel are lecturers.

As the above example shows, to enforce an FGAC policy, it is not enough to check that displaying the final result is policy-compliant. In fact, we claim that any information used to reach this final result (in particular, when solving subqueries, where-clauses, and on-clauses) should be also checked for policy-compliance. Accordingly, if a user is not authorized to know whether Huong is Thanh's lecturer, when attempting to execute `Query#2`, he/she should receive an authorization-error, even when he/she may be authorized to access Huong's email. Similarly, if a user is not authorized to know whether Huong and Manuel are "colleagues",

then, when executing `Query#3`, he/she should receive an authorization-error, even when he/she may be authorized to access lecturers' emails.

Modeling Fine-Grained Access Control Policies

Our approach for enforcing FGAC policies for SQL queries is model-driven. This means, first of all, that policies are specified using models, and, second, that the corresponding policy-enforcement artifacts are generated from these models. Next, we introduce the language SecureUML for modeling FGAC policies. In the next section, we will introduce the policy-enforcement artifacts that can be generated from SecureUML models for executing securely SQL queries.

SecureUML [8] is a modeling language for specifying FGAC policies. It is an extension of role-based access control (RBAC) [6]. As it is well known, in RBAC, permissions are assigned to roles, and roles are assigned to users. However, in SecureUML, one can model access control decisions that depend on two kinds of information: namely, static information, i.e., the assignments of users and permissions to roles; and dynamic information, i.e., the satisfaction of authorization constraints by the current state of the system.

SecureUML leaves open the nature of the protected resources—i.e., whether these resources are data, business objects, processes, controller states, etc.—and, consequently, the nature of the corresponding controlled actions. These are to be declared in a so-called SecureUML dialect. In particular, in our approach, the data that is protected is modeled using classes and associations—as in standard UML class diagrams—while the actions that are controlled are read-actions on class attributes and association-ends. Authorization constraints are specified in SecureUML models using OCL expressions.

Data Models and Object Models

Data models specify the resources to be protected. Object models (also called scenarios) are instances of data models.

Definition 1 (*Data models*) Let $\mathcal{T}$ be a set of predefined types. A data model $\mathcal{D}$ is a tuple $\langle C, AT, AS\rangle$, where

- C is a set of classes c.
- AT is a set of attributes at, $at = \langle ati, c, t\rangle$, where ati is the attribute's identifier; c is the class of the attribute; and t is the type of the values of the attribute, with $t \in \mathcal{T}$ or $t \in C$.
- AS is a set of associations as, $as = \langle asi, ase_1, c_1, ase_r, c_r\rangle$, where: asi is the association's identifier; ase_1 and ase_r are the ends of the association as; c_1 is the class of the objects at the association-end ase_1; and c_r is the class of the objects at the association-end ase_r.

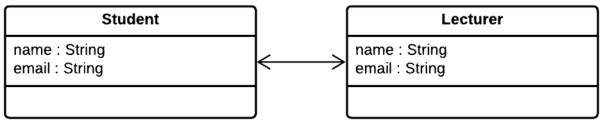

Fig. 2 The `University` model

Definition 2 (*Object models*) Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. An object model $\mathcal{O}$ of $\mathcal{D}$ is a tuple $\langle OC, OAT, OAS\rangle$ where:

- OC is set of objects o, $o = \langle oi, c\rangle$, where oi is the identifier of the object o, and $c \in C$ is the class of the object o.
- OAT is a set of attribute values atv, $atv = \langle\langle ati, c, t\rangle, \langle oi, c\rangle, vl\rangle$, where $\langle ati, c, t\rangle \in AT$, $\langle oi, c\rangle \in OC$, and vl is a value of the type t.
- OAS is a set of association links asl, $asl = \langle\langle asi, ase_1, c_1, ase_r, c_r\rangle, \langle oi_1, c_1\rangle, \langle oi_r, c_r\rangle\rangle$, where $\langle asi, ase_1, c_1, ase_r, c_r\rangle \in AS$, $\langle oi_1, c_1\rangle \in OC$, and $\langle oi_r, c_r\rangle \in OC$.

Without loss of generality, we assume that every object has a unique identifier.

Example 1 We introduce in Fig. 2 the data model `University`, which basically corresponds to the database `UniversityDB` considered in the previous section. The data model `University` contains two classes, `Student` and `Lecturer`, and one association `Enrollment` between both of them. Both classes, `Student` and `Lecturer`, have attributes `name` and `email`.

The class `Student` represents the students of the university, with their names and emails. The class `Lecturer` represents the lecturers of the university, with their names and emails. The association `Enrollment` represents the links between the students (denoted by `students`) and their lecturers (denoted by `lecturers`).

In the following sections, we will consider the following two scenarios of the data model `University`.

Example 2 We introduce in Fig. 3 the scenario `VGU#1`. It contains five students: `An`, `Chau`, `Hoang`, `Thanh`, and `Nam`, with the expected names and emails (*name*`@vgu.edu.vn`). The scenario `VGU#1` also contains three lecturers: `Huong`, `Manuel`, `Hieu`, again with the expected names and emails.

Moreover, `VGU#1` contains `Enrollment`-links between the lecturer `Manuel` and the students `An`, `Chau`, and

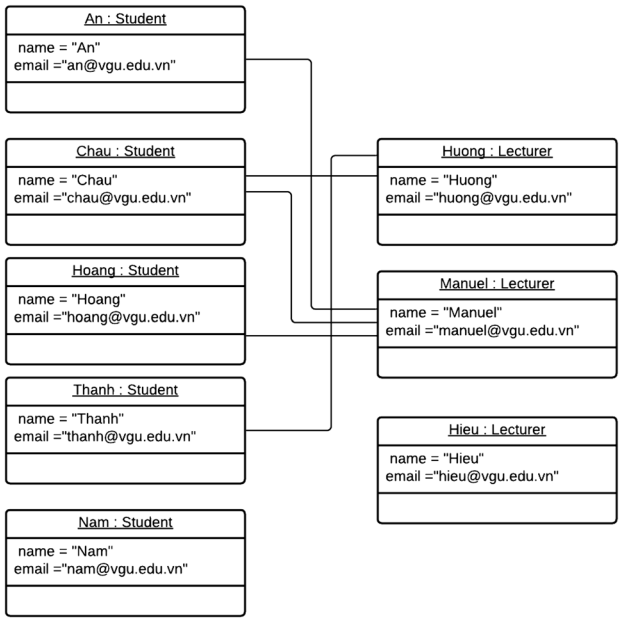

Fig. 3 The VGU#1 scenario

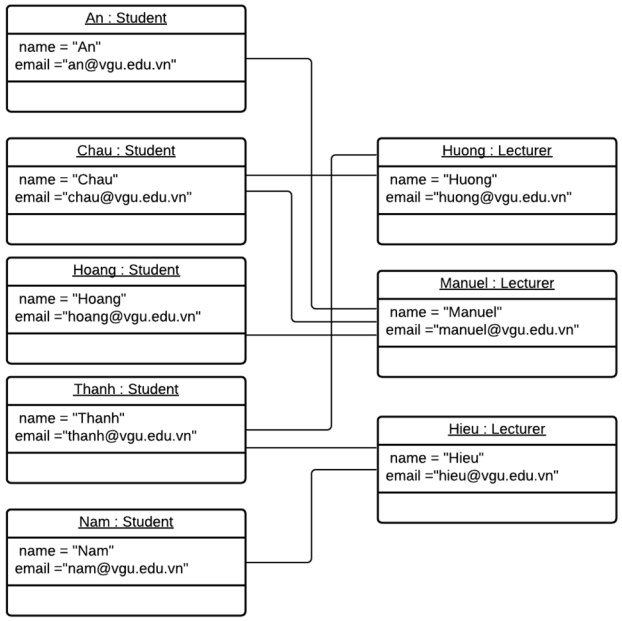

Fig. 4 The VGU#2 scenario

Hoang, and also between the lecturer Huong and the students Chau and Thanh.

Example 3 We introduce in Fig. 4 the scenario VGU#2. that is exactly as VGU#1 except that includes two additional Enrollment-links: one between the lecturer Hieu and the student Thanh, and the other link between the lecturer Hieu and the student Nam.

Object Constraint Language (OCL)

OCL [13] is a language for specifying constraints and queries using a textual notation. Every OCL expression is written in the context of a model (called the contextual model). OCL is a strongly typed language. Expressions either have a primitive type, a class type, a tuple type, or a collection type. OCL provides standard operators on primitive types, tuples, and collections. For example, the operator `includes` checks whether an element is inside a collection. OCL also provides a dot-operator to access the value of an attribute of an object, or the collection of objects linked with another object at the end of an association. OCL also provides operators to iterate over collections, such as `forAll`, `exists`, `select`, `reject`, and `collect`. Collections can be sets, bags, ordered sets and sequences, and can be parameterized by any type, including other collection types. Finally, to represent undefinedness, OCL provides two constants: `null` and `invalid`. Intuitively, `null` represents an unknown or undefined value, whereas `invalid` represents an error or an exception.

Let $\mathcal{D}$ be a data model. We denote by $\text{Exp}(\mathcal{D})$ the set of OCL expressions whose contextual model is $\mathcal{D}$.

Let $\mathcal{O}$ be an instance of $\mathcal{D}$, and let e be an OCL expression in $\text{Exp}(\mathcal{D})$. Then, we denote by $\text{Eval}(\mathcal{O}, e)$ the result of evaluating e in $\mathcal{O}$ according to the semantics of OCL.

Example 4 Let e be the OCL expression `Thanh.email`. Then, $\text{Eval}(\texttt{VGU\#1}, e) = \varepsilon\texttt{thanh@vgu.edu.vn}\varepsilon$, and $\text{Eval}(\texttt{VGU\#2}, e) = \varepsilon\texttt{thanh@vgu.edu.vn}\varepsilon$.

Let e be the OCL expression `Thanh.lecturers`. Then, $\text{Eval}(\texttt{VGU\#1}, e) = \{\texttt{Huong}\}$, while $\text{Eval}(\texttt{VGU\#2}, e) = \{\texttt{Huong}, \texttt{Hieu}\}$.

Let e be the OCL expression `Thanh.lecturers → includes(Hieu)`. Then, $\text{Eval}(\texttt{VGU\#1}, e) = \texttt{false}$, while $\text{Eval}(\texttt{VGU\#2}, e) = \texttt{true}$.

FGAC-Security Models

FGAC-security models specify fine-grained access control policies for executing actions on protected resources. We first define the specific actions that can be protected in our approach. Then, we give a precise definition of FGAC-security models, and of their meaning, i.e., which actions are authorized to be executed for which users, with which roles, and under which conditions.

Definition 3 Let $\mathcal{D}$ be a data model $\mathcal{D} = \langle C, AT, AS \rangle$. Then, we denote by $\text{Act}(\mathcal{D})$ the following read-actions:

- For every attribute $at \in AT$, $\text{read}(at) \in \text{Act}(\mathcal{D})$.
- For every association $as \in AS$, $\text{read}(as) \in \text{Act}(\mathcal{D})$.

Definition 4 Let $\mathcal{D}$ be a data model. Then, a security model $\mathcal{S}$ for $\mathcal{D}$ is a tuple $\mathcal{S} = (R, \text{auth})$, where R is a set of roles, and $\text{auth} : R \times \text{Act(D)} \longrightarrow \text{Exp}(\mathcal{D})$ is a function that assigns to each role $r \in R$ and each action $a \in \text{Act}(\mathcal{D})$ an authorization constraint $e \in \text{Exp}(\mathcal{D})$.

In our approach, we consider authorization constraints whose satisfaction depends on information related to: (i) the user who is attempting to perform a read-action; (ii) the object whose attribute is attempted to be read; and, (iii) the objects between which a link is attempted to be read. By convention, we denote (i) by the keyword *caller*; we denote (ii) by the keyword *self*; and we denote (iii) using as keywords the corresponding association-ends.

Next, we provide three examples of FGAC-security models specifying three different FGAC policies for accessing `University`-data.

Example 5 Consider the following clause for accessing lists of students.

– A lecturer can know the list of his/her own students.

Also, consider also the following clause for accessing emails of lectures and students.

– A lecturer can know his/her own email, as well as the emails of his/her students.

The model `SecVGU#A` precisely specifies the above policy using SecureUML

−roles = {`Lecturer`}.
−auth() =

$$\begin{cases} (\texttt{Lecturer}, \text{read}(\texttt{Enrollment})) \mapsto & (\textit{caller} = \textit{lecturers}). \\ (\texttt{Lecturer}, \text{read}(\texttt{Student : email})) \mapsto & \\ \quad (\textit{caller}.\texttt{students} \rightarrow \texttt{includes}(\textit{self})). & \\ (\texttt{Lecturer}, \text{read}(\texttt{Lecturer : email})) \mapsto & (\textit{caller} = \textit{self}). \end{cases}$$

Example 6 Consider a policy that is exactly as `SecVGU#A` except that it includes the following additional clause:

– A lecturer can know its colleagues' emails. For the sake of these examples, two lecturers are "colleagues" if there is at least one student enrolled with both of them.

The model `SecVGU#B` precisely specifies the above policy using SecureUML

−roles = {`Lecturer`}.
−auth() =

$$\begin{cases} (\texttt{Lecturer}, \text{read}(\texttt{Enrollment})) \text{ as in } \texttt{SecVGU\#A}. \\ (\texttt{Lecturer}, \text{read}(\texttt{Student : email})) \text{ as in } \texttt{SecVGU\#A}. \\ (\texttt{Lecturer}, \text{read}(\texttt{Lecturer : email})) \mapsto \\ \quad (\textit{caller} = \textit{self})\ \texttt{or} \\ \quad (\textit{caller}.\texttt{students} \rightarrow \textit{exists} \\ \quad (s \mid s.\textit{lecturers} \rightarrow \textit{includes}(\textit{self}))). \end{cases}$$

Example 7 Consider a policy that is exactly as `SecVGU#B` except that it includes the following additional clause:

– A lecturer can know the list of lecturers of his/her own students.

The model `SecVGU#C` precisely specifies the above policy using SecureUML

−roles = {`Lecturer`}.
−auth() =

$$\begin{cases} (\texttt{Lecturer}, \text{read}(\texttt{Enrollment})) \mapsto \\ \quad (\textit{caller} = \textit{lecturers})\ \textit{or} \\ \quad (\textit{caller}.\texttt{students} \rightarrow \texttt{includes}(\textit{students})). \\ (\texttt{Lecturer}, \text{read}(\texttt{Student : email}))\ \text{as in } \texttt{SecVGU\#B}. \\ (\texttt{Lecturer}, \text{read}(\texttt{Lecturer : email}))\ \text{as in } \texttt{SecVGU\#B}. \end{cases}$$

Definition 5 Let $\mathcal{D}$ be a data model. Let $\mathcal{S} = \langle R, \text{auth}\rangle$ be an FGAC-security model for $\mathcal{D}$. Let $\mathcal{O} = \langle OC, OAT, OAS\rangle$ be an object model of $\mathcal{D}$. Then

– A user u with role $r \in R$ is authorized, according to $\mathcal{S}$, to read the value of an attribute $at = \langle ati, c, t\rangle$, $at \in AT$ of an object $o \in OC$ if and only if

$$\text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(ati)[\texttt{self} \leftarrow o; \texttt{caller} \leftarrow u]) = \texttt{true}.$$

– A user u with role $r \in R$ is authorized, according to $\mathcal{S}$, to read whether an association $as = \langle asi, ase_{\text{l}}, c_{\text{l}}, ase_{\text{r}}, c_{\text{r}}\rangle$, $as \in AS$, $as \in AS$ links two objects o_{l} and o_{r}, if an only if

$$\text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(asi)[as_{\text{l}} \leftarrow o_{\text{l}}; as_{\text{r}} \leftarrow o_{\text{r}}; caller \leftarrow u]) = \texttt{true}.$$

By convention, the function auth() may take as an extra argument the security model to which it belongs, when the latter is not clear from the context.

Example 8 Suppose that `Manuel` has role `Lecturer`. Then, according to `SecVGU#A`, in the scenario `VGU#1`:

– Manuel is authorized to know whether An, Hoang, Chau, Thanh, and Nam are his students. Recall that

auth(SecVGU#A, Lecturer, read(Enrollment))
= (*caller* = *lecturers*),

Notice that, for *std* ∈ {An, Hoang, Chau, Thanh, Nam}, it holds that

Eval(VGU#1, auth(SecVGU#A, Lecturer,
read(Enrollment))[*caller* ← Manuel,
lecturers ← Manuel, *students*
← *std*) = true

– Manuel is not authorized to know whether An, Hoang, Chau, Thanh or Nam are students of Huong. Notice that in this case

Eval(VGU#1, auth(SecVGU#A, Lecturer,
read(Enrollment))[*caller* ← Manuel,
lecturers ← Huong, *students*
← *std*) = false.

Example 9 Suppose that Manuel has role Lecturer. Then, according to SecVGU#C, in the scenario VGU#1:

– Manuel is authorized to know whether An, Hoang, Chau, Thanh, and Nam are his students. Recall that SecVGU#C specifies that

auth(SecVGU#C, Lecturer, read(Enrollment)) =
(caller = lecturers) or
(*caller*.students → includes(*students*)).

Notice that, for *std* ∈ {An, Hoang, Chau, Thanh, Nam}, it holds that

Eval(VGU#1, auth(SecVGU#C, Lecturer,
read(Enrollment))[*caller* ← Manuel,
lecturers ← Manuel, *students*
← *std*) = true

– Manuel is authorized to know whether An, Hoang, Chau are students of Huong. Notice that, for *std* ∈ {An, Hoang, Chau}, it holds that

Eval(VGU#1, auth(SecVGU#C, Lecturer,
read(Enrollment))[*caller* ← Manuel,
lecturers ← Huong, *students*
← *std*) = true

– Manuel is not authorized to know whether Thanh or Nam are students of Huong. Notice that in this case, for *std* ∈ {Thanh, Nam}, it holds that

Eval(VGU#1, auth(SecVGU#C, Lecturer,
read(Enrollment))[*caller* ← Manuel,
lecturers ← Huong, *students* ← *std*) = false.

Enforcing FGAC Policies for SQL Queries

In [12], we formally defined the conditions that need to be satisfied for a user u, with role r, to be authorized to execute a SQL query q according to an FGAC policy $\mathcal{S}$. In this section, we present our solution for enforcing these conditions when executing queries in SQL. In a nutshell, we define a function SecQuery() that, given an FGAC policy S and an SQL query q, it generates an SQL stored-procedure, which takes two arguments, caller and role, representing, respectively, the user u attempting to execute the query q and the role r with which he/she attempts to execute q. This stored-procedure creates a list of temporary tables, corresponding to the different conditions that need to be satisfied for the user u, with role r, to be authorized to execute the query q, according to $\mathcal{S}$. The definition of each temporary table is such that, when attempting to create the table, if the corresponding condition is not satisfied, then an error will be signaled and the table will not be created. If all temporary tables can be successfully created, then the stored-procedure generated by SecQuery() will execute q; if any of the temporary tables cannot be created, then an error will be signaled. The reason for using temporary tables instead of subqueries is to prevent the SQL optimizer for "skipping" (by "silently" rewriting the subqueries) some of the conditions that SecQuery() must introduce to guarantee that a query is executed securely. The definition of SecQuery() assumes that the policies' underlying data models are implemented in SQL following a specific mapping. Notice that other mappings from data models to SQL are also possible [5]. As expected, if a different mapping from data models to SQL is chosen, then our enforcement of FGAC policies for SQL queries should be changed accordingly. The definition of SecQuery() also assumes that the enforcing mechanism is allowed to access the information needed to perform in each case the corresponding authorization checks.

Mapping Data and Object Models to Databases

Next, we define the specific mappings from data models and object models to SQL that we use in our solution for enforcing FGAC policies when executing SQL queries.

Definition 6 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Our mapping of $\mathcal{D}$ to SQL, denoted by $\overline{\mathcal{D}}$, is defined as follows:

– For every $c \in C$,

```
CREATE TABLE c (c_id varchar PRIMARY KEY);
```

– For every attribute $at \in AT$, $at = \langle ati, c, t \rangle$,

```
ALTER TABLE c ADD COLUMN ati SqlType(t);
```

where

- if $t =$ `Integer`, then SqlType(t) = `int`.
- if $t =$ `String`, then SqlType(t) = `varchar`.
- if $t \in C$, then SqlType(t) = `varchar`.

Moreover, if $t \in C$, then

```
ALTER TABLE c ADD FOREIGN KEY fk_c_ati(ati) REFERENCES t(t_id);
```

– For every association $as \in AS$, $as = \langle asi, ase_1, c_1, ase_r, c_r \rangle \in AS$,

```
CREATE TABLE asi (
  ase_1 varchar NOT NULL,
  ase_r varchar NOT NULL,
  FOREIGN KEY fk_c_1_ase_1(ase_1) REFERENCES c_1(c_1_id),
  FOREIGN KEY fk_c_r_ase_r(ase_r) REFERENCES c_r(c_r_id));
```

Moreover

```
ALTER TABLE asi ADD UNIQUE unique_link(ase_1, ase_r);
```

Definition 7 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{O} = \langle OC, OAT, OAS \rangle$ be an object model of $\mathcal{D}$. Our mapping of $\mathcal{O}$ to SQL, denoted by $\overline{\mathcal{O}}$, is defined as follows:

– For every object $o \in OC$, $o = \langle oi, c \rangle$,

```
INSERT INTO c(c_id) VALUES (oi);
```

– For every attribute value $atv \in OAT$, $atv = \langle \langle ati, c, t \rangle, \langle oi, c \rangle, vl \rangle$,

```
UPDATE c SET ati = vl WHERE c_id = oi;
```

– For every association link $asl \in OAS$, $asl = \langle \langle asi, ase_1, c_1, ase_r, c_r \rangle, \langle oi_1, c_1 \rangle, \langle oi_r, c_r \rangle \rangle$,

```
INSERT INTO asi(ase_1, ase_r) VALUES (oi_1, oi_r);
```

Secure SQL Queries

Next, we introduce the key component in our model-driven solution for enforcing FGAC policies when executing SQL queries.

The Function SecQuery()

Given an FGAC policy $\mathcal{S}$ and an SQL select-statement q, the function SecQuery() generates a SQL stored-procedure satisfying the following: if a user is authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure returns the same result that executing q; otherwise, if a user is not authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure signals an error.

By convention, we denote by $\ulcorner \text{SecQuery}(\mathcal{S}, q) \urcorner$ the name of the stored-procedure generated by SecQuery, for an

FGAC policy $\mathcal{S}$ and a query q. SecQuery() uses the auxiliary function SecQueryAux() that is defined in the next section.

Definition 8 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = (R, \text{auth})$ be a security model for $\mathcal{D}$. Let q be an SQL query in $\overline{\mathcal{D}}$. Then, SecQuery($\mathcal{S}, q$) generates the following stored-procedure:

```
CREATE PROCEDURE ⌜SecQuery(S, q)⌝ (
    caller varchar(250), role varchar(250))
BEGIN
DECLARE _rollback int DEFAULT 0;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
  % If an error is signalled, then set _rollback to 1 and
  % return the error message.
  SET _rollback = 1;
  GET STACKED DIAGNOSTICS CONDITION 1
    @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;
  SELECT @p1, @p2;
  ROLLBACK;
END;
START TRANSACTION;
    % For each authorization condition applicable to the original query,
    % create the corresponding temporary table.

    SecQueryAux(S, q)

    % If after creating all the temporary tables, no error has
    % been signalled yet, i.e., _rollback has still value 0,
    % then execute the original query.
IF _rollback = 0
  THEN q;
END IF;
END
```

SecQueryAux() assumes that the policies' underlying data models, as well as its object models, are implemented in SQL following the mapping introduced in the section "Mapping data and object models to databases". According to this mapping, the rows in the association-tables only represent the links of the given association that exist between objects. In other words, if a link does not exist, this information is not stored anywhere. Thus, when checking if a user is authorized to know the links of a given association, we should not only perform the appropriate checks on the rows contained in the corresponding association-table, but also on the rows contained in its (virtual) complement, i.e., on the table whose rows represent the links that do not exist between objects. For this reason, in the definition of SecQueryAux() below, when handling read-access authorization for tables representing associations, we consider the Cartesian product of the two class-tables involved in the given association, checking read-access authorization for all the rows in the Cartesian product.

The Function SecQueryAux

The function SecQuery() uses the function SecQueryAux to create a temporary table corresponding to each authorization condition applicable when executing a query. As expected, our definition of SecQueryAux() proceeds recursively. By convention, we denote by ⌜TempTable(q, exp)⌝ the name of the temporary table generated by SecQuery, for a query q and a (sub-)expression exp.

Before going further, a word of caution is in order. A subtle, but important point in our definition of SecQueryAux() has to do with our way of handling read-access authorization for tables representing associations. The definition of

Next, we introduce the different cases in the recursive definition of the function SecQueryAux(). For each case, we informally introduce the authorization conditions that need to be satisfied. As mentioned before, we have formally defined these conditions in [12]. According to these conditions, any data that are used when executing a query (in particular, data used by subqueries, where-clauses, and on-clauses) must be checked for policy-compliance (and not only the data that appears in the final result). To this end, the function SecQueryAux() uses the function SecAtt() to add the corresponding authorization-checks to any expression accessing specific attribute values, and the function SecAs() to add the corresponding authorization-checks to access association links. These functions will be introduced in the next section. The function SecAttList(), also used by SecQueryAux(), simply applies SecAtt() to each of the expressions in an expression list. Finally, in the definitions below, we denote by RepExp() the result of replacing, within an expression, each occurrence of the association's association-ends by the corresponding association-ends' class-identifier.

Case $q =$ `SELECT` *selitems* `FROM` c `WHERE` *exp*. *To execute* q, *the following conditions must be satisfied*:

- The user is authorized to access the information required to evaluate the where-clause *exp*.
- The user is authorized to access the information referred to by *selitems*, but only for the objects/rows that satisfy the where-clause *exp*.

For this case, SecQueryAux() returns the following create-statements:

```
CREATE TEMPORARY TABLE ⌜TempTable(q, exp)⌝ AS (
  SELECT * FROM c WHERE SecAtt(S, exp));
CREATE TEMPORARY TABLE ⌜TempTable(q, selitems)⌝ AS (
  SELECT SecAttList(S, selitems) FROM ⌜TempTable(q, exp)⌝);
```

Case $q =$ `SELECT` *selitems* `FROM` *as* `WHERE` *exp*. To execute q, the following conditions must be satisfied:

- The user is authorized to access the information referred to by both association-ends, but only for the rows contained in the Cartesian product between the classes involved in the association that satisfy the where-clause *exp*.

For this case, SecQueryAux() returns the following create-statements:

```
CREATE TEMPORARY TABLE ⌜TempTable(q, exp)⌝ AS (
  SELECT c_l_id as ase_l, c_r_id as ase_r FROM c_l, c_r
    WHERE RepExp(exp, as) );
CREATE TEMPORARY TABLE ⌜TempTable(q, selitems)⌝ AS (
  SELECT * FROM ⌜TempTable(q, exp)⌝ WHERE SecAs(S, as) );
```

Case $q =$ `SELECT` *selitems* `FROM` *subselect* `WHERE` *exp*. To execute q, the following conditions must be satisfied:

- The user is authorized to execute the subquery *subselect*.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, *subselect*)

Case $q =$ `SELECT` *selitems* `FROM` c `JOIN` *as* `ON` *exp* `WHERE` *exp*$'$. To execute q, the following conditions must be satisfied:

- The user is authorized to access the information referred to by both association-ends.
- The user is authorized to access the information required to evaluate the on-clause *exp*.
- The user is authorized to access the information required to evaluate the where-clause *exp*$'$, but only for the objects/rows and links/rows that satisfy the on-clause *exp*.
- The user is authorized to access the information referred to by *selitems*, but only for the objects/rows and links/rows that satisfy the on-clause *exp* and the where-clause *exp*$'$.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, `SELECT` $*$ `FROM` *as*)
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*)⌝ `AS (`
`SELECT * FROM` c `JOIN` *as* `ON` SecAtt($\mathcal{S}$, *exp*)`);`
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*$'$)⌝ `AS (`
`SELECT * FROM` ⌜TempTable⌝(q, *exp*) `WHERE` SecAtt($\mathcal{S}$, *exp*$'$) `);`
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *selitems*)⌝ `AS (`
`SELECT` SecAttList($\mathcal{S}$, *selitems*) `FROM` ⌜TempTable(q, *exp*$'$)⌝ `);`

Case $q =$ `SELECT` *selitems* `FROM` c `JOIN` *subselect* `ON` *exp* `WHERE` *exp*$'$. To execute q, the following conditions must be satisfied:

- The user is authorized to execute the subquery *subselect* .
- The user is authorized to access the information required to evaluate the on-clause *exp*.
- The user is authorized to access the information required to evaluate the where-clause *exp*$'$; but only for the objects/rows and links/rows that satisfy the on-clause *exp*.
- The user is authorized to access the information referred to by *selitems*, but only for the objects/rows and links/rows that satisfy the on-clause *exp* and the where-clause *exp*$'$.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, *subselect*)
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*)⌝ `AS (`
`SELECT * FROM` c `JOIN` *subselect* `ON` SecAtt($\mathcal{S}$, *exp*)`);`
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*$'$)⌝ `AS (`
`SELECT * FROM` ⌜TempTable(q, *exp*)⌝ `WHERE` SecAtt($\mathcal{S}$, *exp*$'$) `);`
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *selitems*)⌝ `AS (`
`SELECT` SecAttList($\mathcal{S}$, *selitems*) `FROM` ⌜TempTable(q, *exp*$'$)⌝ `);`

Case $q =$ `SELECT` *selitems* `FROM` *as* `JOIN` *subselect* `ON` *exp* `WHERE` *exp*$'$. We must consider three cases:

First, the case when ase_{l} appears in *exp*, but ase_{r} does not appear in *exp*. Let *col* be the column in *subselect* that ase_{l} is related to in *exp*. To execute *q*, the following conditions must be satisfied:

- The user is authorized to execute the subquery subselect.
- The user is authorized to access the information referred to by both association-ends, but only for the rows contained in the Cartesian product between the classes involved in the association that satisfy the where-clause *exp*.

For this case, SecQueryAux() returns the following create-statements:

$$\begin{array}{l}\text{SecQueryAux}(\mathcal{S}, \textit{subselect})\\ \texttt{CREATE TEMPORARY TABLE } \ulcorner\text{TempTable}(q, \textit{exp})\urcorner \texttt{ AS (}\\ \quad \texttt{SELECT } c_{\text{l}}\texttt{\_id as } ase_{\text{l}}, \textit{col} \texttt{ as } ase_{\text{r}} \texttt{ FROM } c_{\text{l}}, \textit{subselect}\\ \qquad \texttt{ON } \text{RepExp}(\textit{exp}, \textit{as}) \texttt{ WHERE } \text{RepExp}(\textit{exp}', \textit{as}) \texttt{);}\\ \texttt{CREATE TEMPORARY TABLE } \ulcorner\text{TempTable}(q, \textit{as})\urcorner \texttt{ AS (}\\ \quad \texttt{SELECT * FROM } \ulcorner\text{TempTable}(q, \textit{exp})\urcorner \texttt{ WHERE } \text{SecAs}(\mathcal{S}, \textit{as}) \texttt{);}\end{array}$$

Second, the case when ase_{r} appears in *exp*, but ase_{l} does not appear in *exp*.

This case is resolved analogously to the previous case.

Third, the case when both ase_{r} and ase_{l} appear in *exp*. To execute *q*, the following conditions must be satisfied:

- The user is authorized to execute the subquery subselect.
- The user is authorized to access the information referred to by both association-ends.

For this case, SecQueryAux() returns the following create-statements:

$$\begin{array}{c}\text{SecQueryAux}(\mathcal{S}, \textit{subselect})\\ \text{SecQueryAux}(\mathcal{S}, \texttt{SELECT} * \texttt{FROM } \textit{as})\end{array}$$

Case $q =$ `SELECT` *selitems* `FROM` $\textit{subselect}_1$ `JOIN` $\textit{subselect}_2$ `ON` *exp* `WHERE` *exp*$'$. To execute *q*, the following conditions must be satisfied:

- The user is authorized to execute the subqueries $\textit{subselect}_1$ and $\textit{subselect}_2$.

For this case, SecQueryAux() returns the following create-statements:

$$\begin{array}{c}\text{SecQueryAux}(\mathcal{S}, \textit{subselect}_1)\\ \text{SecQueryAux}(\mathcal{S}, \textit{subselect}_2)\end{array}$$

Table 1 Examples

Caller	Query	SecVGU#A		SecVGU#B		SecVGU#C	
		VGU#1	VGU#2	VGU#1	VGU#2	VGU#1	VGU#2
Manuel	Query#1	✗	✗	✓	✓	✓	✓
Huong		✓	✓	✓	✓	✓	✓
Hieu		✗	✗	✗	✓	✗	✓
Manuel	Query#2	✗	✗	✗	✗	✗	✗
Huong		✓	✓	✓	✓	✓	✓
Hieu		✗	✗	✗	✗	✗	✓
Manuel	Query#3	✗	✗	✗	✗	✗	✗
Huong		✗	✗	✗	✗	✗	✗
Hieu		✗	✗	✗	✗	✗	✗

Calling stored-procedures generated by SecQuery() for different queries and policies, with different users and scenarios

The Function SecAtt()

The function SecQueryAux() uses SecAtt() to wrap any access to a protected attribute *at* into a case expression. The value of this case expression is a call to a function AuthFunc() that implements that authorization-checks required for accessing the corresponding attribute. If the result of this function-call is `TRUE`, then the case expression will return the requested resource; otherwise, it will signal an error. The function AuthFunc() is defined in the following section. By convention, we denote by $\ulcorner \text{AuthFunc}(\mathcal{S}, at) \urcorner$ the name of the function generated by SecQuery() for a policy $\mathcal{S}$ an attribute *at*; when the argument $\mathcal{S}$ is clear from the context, we may omit it.

Definition 9 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = (R, \text{auth})$ be a security model for $\mathcal{D}$. Let *exp* be an SQL expression in $\overline{\mathcal{D}}$. We denote by $\text{SecAtt}(\mathcal{S}, exp)$ the SQL expression in $\overline{\mathcal{D}}$ that results from replacing each attribute $at = \langle ati, c, t \rangle$ in *exp* by the following case expression:

```
CASE ⌜AuthFunc(at)⌝ (c_id, caller, role)
  WHEN 1 THEN at
  ELSE throw_error() END as at.
```

where the function throw error is defined as follows:

```
CREATE FUNCTION throw_error()
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Unauthorized access';
  RETURN (0);
END
```

Query#4	SELECT COUNT(students) FROM Enrollment WHERE lecturers = 'Hieu';

Fig. 5 Example. Query 4

The Function SecAs()

The function SecQueryAux() uses SecAs() to wrap any access to a protected association *as* into a where case expression. The value of this case expression is a call to the function AuthFunc() that, in this case, implements the authorization-checks required for accessing the corresponding association-ends. If the result of this function-call is `TRUE`, then the case expression will return also `TRUE`; otherwise, it will signal an error. The function AuthFunc() is defined in the following section. By convention, we denote by $\ulcorner$AuthFunc($\mathcal{S}, as$)$\urcorner$ the name of the function generated by SecQuery() for a policy $\mathcal{S}$ an association *as*; when the argument $\mathcal{S}$ is clear from the context, we may omit it.

Definition 10 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = (R, \text{auth})$ be a security model for $\mathcal{D}$. Let *as* be an association class in $\mathcal{D}$. Let ase_l and ase_r be the association-ends of *as*. We denote by SecAs($\mathcal{S}, as$) the SQL expression in $\overline{\mathcal{D}}$ that results by the following case expression:

```
CASE ⌜AuthFunc(as)⌝ (ase_l, ase_r, caller, role)
  WHEN 1 THEN TRUE
  ELSE throw_error() END
```

where the function `throw_error()` is defined as above.

The Function AuthFunc()

The functions SecAtt() and SecAs() use this function to check that the access to a specific protected resource is authorized. For each protected resource, the required authorization-checks depend on the role of the user attempting to access this resource. Accordingly, for each role, the function AuthFunc() calls a function AuthFuncRole() that implements the authorization-checks required for a user with that role to access a specific protected resource. The function AuthFuncRole() will be introduced in the next section. By convention, we denote by $\ulcorner$AuthFuncRole($\mathcal{S}, rs, r$)$\urcorner$ the name of the function generated by SecQuery() for a policy $\mathcal{S}$, a resource *rs*, and a role *r*; when the argument $\mathcal{S}$ is clear from the context, we may omit it.

Definition 11 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = (R, \text{auth})$ be a security model for $\mathcal{D}$, with $R = \{r_1, r_2, \ldots, r_n\}$. Let *at* be an attribute in *AT*. Then, AuthFunc(*at*) generates the following SQL function:

```
CREATE FUNCTION ⌜AuthFunc(at)⌝ ( self varchar(250),
  caller varchar(250), role varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  IF (role = r1)
    THEN RETURN ⌜AuthFuncRole(at, r1)⌝(self, caller)
  ELSE IF (role = r2)
    THEN RETURN ⌜AuthFuncRole(at, r2)⌝(self, caller)
  ⋮
  ELSE IF (role = rn)
    THEN RETURN ⌜AuthFuncRole(at, rn)⌝(self, caller)
  ELSE RETURN 0
  END IF;
  ⋮
  END IF;
  END IF;
END
```

Similarly, let as be an association in AS. Then, AuthFunc(as) generates the following SQL function:

```
CREATE FUNCTION ⌜AuthFunc(as)⌝ ( left varchar(250),
  right varchar(250), caller varchar(250), role varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  IF (role = r1)
    THEN RETURN ⌜AuthFuncRole⌝(as, r1) (left, right, caller)
  ELSE IF (role = r2)
    THEN RETURN ⌜AuthFuncRole(as, r2)⌝ (left, right, caller)
  ⋮
  ELSE IF (role = rn)
    THEN RETURN ⌜AuthFuncRole(as, rn)⌝ (left, right, caller)
  ELSE RETURN 0
  END IF;
  ⋮
  END IF;
  END IF;
END
```

The Function AuthFuncRole()

The function AuthFuncRole() implements the authorization constraints associated with the permission for users of

Query#5	SELECT COUNT(*) FROM Student WHERE age > 18
Query#6	SELECT COUNT(students) from Enrollment

Fig. 6 Example. Queries 5–6

a given role for executing a given read-action on a specific resource.

Of course, there are many different ways of implementing in SQL an OCL authorization constraint. In our definition of the function AuthFuncRole(), we only assume that there exists a function map() that, for each authorization constraint of interest, it returns its preferred SQL implementation. Without loss of generality, we also assume that this implementation, when executed, will return a SQL Boolean.[1]

Definition 12 Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = (R, \text{auth})$ be a security model for $\mathcal{D}$. Let r be a role in R. Let $at = \langle ati, c, t \rangle$ be an attribute in AT. Then, AuthFuncRole(at, r) generates the following SQL function:

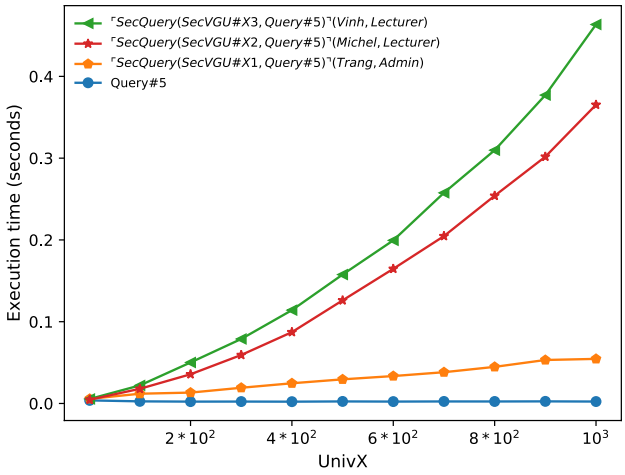

Fig. 7 Experiments: Query#5

Examples

To illustrate our definition of the function SecQuery(), we show in Table 1 the results of calling the stored-procedures

```
CREATE FUNCTION ⌜AuthFuncRole(at, r)⌝ ( self varchar(250),
  caller varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SELECT * INTO result FROM map(auth(r, read(at))) AS TEMP;
  RETURN result; END
```

Similarly, let $as = \langle asi, ase_l, c_l, ase_r, c_r \rangle \in AS$, be an association in AS. Then, AuthFuncRole(as, r) generates the following SQL function:

generated by this function for the queries Query#1, Query#2, and Query#3 (in Fig. 1), on the scenarios VGU#1 and VGU#2, for the policies SecVGU#A, SecVGU#B, and SecVGU#C, when the callers are the

```
CREATE FUNCTION ⌜AuthFuncRole(as, r)⌝ ( left varchar(250),
  right varchar(250), caller varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SELECT * INTO result FROM map(auth(r, read(as))) AS TEMP;
  RETURN result;
END
```

[1] Our mapping OCL2PSQL [11] can certainly be used as map() -function. However, our current experiments suggest that, for non-trivial authorization constraints, manually written implementations significantly outperforms those automatically generated by OCL2PSQL, when checking FGAC authorization in large databases.

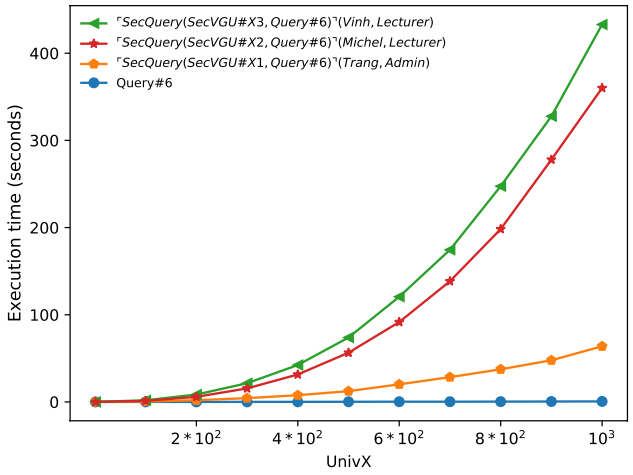

Fig. 8 Experiments: `Query#6`

lecturers `Manuel`, `Huong` and `Hieu`. The mark ✓ indicates that the caller is authorized to execute the query (and therefore the expected result is returned), while the mark ✗ indicates that the caller is not authorized to execute the query (and therefore an error is signaled). Notice in particular that:

- `Manuel` is not authorized to execute `Query#2` for any of the scenarios `VGU#1` and `VGU#2`, according to the policy `SecVGU#C`. This is to be expected, since `Thanh` is not a student of `Manuel` in any of these scenarios, and, therefore, `Manuel` is not authorized to know that `Thanh` is a student of `Huong`.
- `Hieu` is not authorized to execute `Query#2` for the scenario `VGU#1`, according to the policy `SecVGU#C`. This is to be expected, since `Thanh` is not a student of `Hieu` in this scenario, and, therefore, `Hieu` is not authorized to know that `Thanh` is a student of `Huong`.

  However, in the scenario `VGU#2`, `Thanh` is in fact a student of Hieu, and, therefore, `Hieu` is authorized to know that `Thanh` is also a student of `Huong`, according to the policy `SecVGU#C`.
- `Huong` is not authorized to execute `Query#3` for any of the scenarios `VGU#1` and `VGU#2`, according to the policies `SecVGU#C`. This is to be expected, since for each of these scenarios, there is at least one student who is a student of `Manuel`, but he/she is not a student of `Huong`, and therefore, `Huong` is not authorized to know that he/she is in fact a student of `Manuel`, according to the policy `SecVGU#C`.

Our next example serves to illustrate the issue of handling read-access authorization for associations. Recall that, in our implementation of objects models in SQL, the rows in these tables only represent existing links between objects. Consider now the select-statement `Query#4` in Fig. 5. Recall that in the scenario `VGU#1`, `Hieu` has no students. Notice that, according to the policy `SecVGU#C`, for the scenario `VGU#1`, `Huong` is not authorized to execute this query, since she is not authorized to know the students of `Hieu` (unless they happen to be her own students, which is not the case in this scenario). Consider now a naive implementation of read-access authorization for tables representing associations that will only perform authorization-checks on the rows contained in these tables. Since in the scenario `VGU#1`, there are no links between Hieu and students, if we follow this naive implementation, `Huong` will be authorized to know that `Hieu` has no students in the scenario `VGU#1`, and conclude, logically, that neither An nor Nam, for example, are students of `Hieu`. To avoid this undesired leakage of information, when handling read-access authorization for associations, our implementation not only performs the appropriate checks on the rows contained in the corresponding association-table, but also on the rows contained in its (virtual) complement, i.e., on the table whose rows represent the links that do not exist between objects. As expected then, when calling the stored-procedure generated by SecQuery() for the query `Query#4` on the scenario `VGU#1`, we obtain the following results:

- `Huong` is not be authorized to execute `Query#4` for the scenario `VGU#1`, according to the policy `SecVGU#C`, because she is not authorized to know that `An`, `Nam`, and `Hoang` are not students of `Hieu` in this scenario.
- `Hieu` is authorized to execute `Query#4` or the scenario `VGU#1`, according to the policy `SecVGU#C`, because he is authorized to know if a student is or not his student.

Performance

Fine-grained access control (FGAC) policies depend not only on static information, namely the assignments of users and permissions to roles, but also on dynamic information, namely the satisfaction of authorization constraint on the current state of the system. Thus, executing FGAC-related authorization-checks will unavoidably cause a performance overhead at execution-time, which is greater or lesser depending on the "size" of the database and the "complexity" of the authorization-checks. In this section, we conduct several experiments to analyze the performance-overhead incurred when executing securely queries by calling the corresponding stored-procedures generated by our function SecQuery().

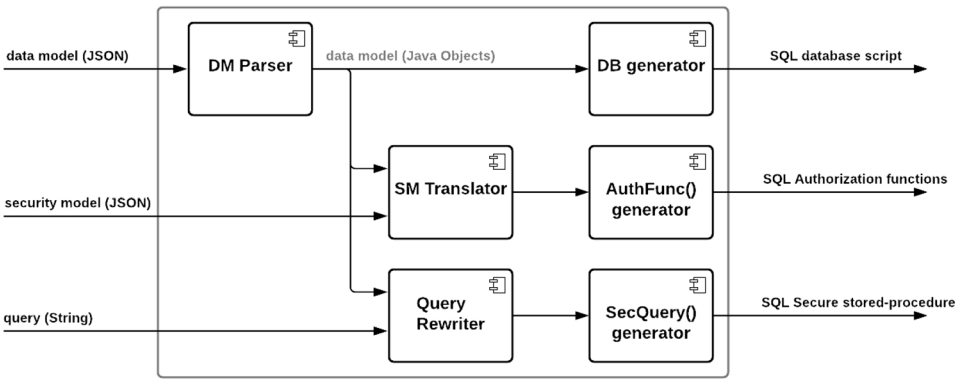

Fig. 9 An overview of the SQLSI component diagram

Experimental Setup

The following experiments were conducted on an MySQL server (5.7.25.1) running on a server computer with Intel(R) Xeon(R) CPU E5-2620 v3, 2.40 GHz, and 16 GB RAM. For each experiment, the execution-time that we report actually corresponds to the arithmetic mean of 10 different executions.

Next, we introduce the data model, object models, FGAC-security models, and queries that we consider in our experiments.

Data Model

We simply extend the data model `University` introduced in the section "Modeling fine-grained access control policies" by adding an attribute `age` to both classes `Student` and `Lecturer`. We call this data model `UniversityX`.

Object Models

For the sake of simplicity, we consider scenarios with the same number of students and lecturers, and in which every student is a student of every lecture. More specifically, for $n > 0$, we denote by `UnivX`(n) an instance of the data model `UniversityX`, such that

- There are exactly n students. Students have unique name.
- There are exactly n lecturers. Lecturers have a unique name.
- Every lecturer has every student as his/her student. Thus, the number of enrollments is n^2.

FGAC-Security Models

We consider the following FGAC-security models for the data model `UniversityX`.

SecVGU#X1 Consider the following clause for accessing the age of students.

- An admin can know the age of any student.

Consider also the following clause for accessing the list of students.

- An admin can know the students of any lecturer.

The following model precisely specifies the above policy using SecureUML:

−roles = {`Admin`}.
−auth(`Admin`, read(`Enrollment`))
= auth(`Admin`, read(`Student : age`))
= (`true`).

For our experiments, we define the function map(), which implements the above authorization-constraints in SQL, as follows:

map(auth(`Lecturer`, read(`Enrollment`)))
= map(auth(`Lecturer`, read(`Student : age`)))
= `TRUE`.

SecVGU#X2 Consider the following clause for accessing the age of students.

- A lecturer can know the age of any student, if no other lecturer is older than he/she is.

Consider also the following clause for accessing lists of students.

- A lecturer can know the students of any lecturer if no other lecturer is older than he/she is..

The following model SecVGU#X2 precisely specifies the above policy using SecureUML:

−roles = {Lecturer}.
−auth(Lecturer, read(Enrollment))
= auth(Lecturer, read(Student : age))
= Lecturer.allInstances()
→ select(l | l.age >*caller*.age) → isEmpty().

For our experiments, we define the function map() as below. Recall that map() is called by the function AuthFuncRole() that takes *caller* as one of its arguments, both for the case of the association Enrollment and the attribute age

map(auth(Lecturer, read(Enrollment)))
= map(auth(Lecturer, read(Student : age)))
=
((SELECT MAX(age) FROM Lecturer)
= (SELECT age FROM Lecturer WHERE Lecturer_id = *caller*)).

arguments, and, for the case of the attribute age, it takes *caller* and *self* as its arguments

map(auth(Lecturer, read(Enrollment)))
= EXISTS (SELECT 1 FROM Enrollment
WHEREEnrollment.lecturers = *caller*
AND Enrollment.students = *students*).

map(auth(Lecturer, read(Student : age)))
EXISTS (SELECT 1 FROM Enrollment
WHERE Enrollment.lecturers = *caller*
AND Enrollment.students = *self*).

SecVGU#X3 Consider the following clause for accessing the age of students.

– A lecturer can know the age of any student, if the student is his/her student.

Consider also the following clause for accessing lists of students.

– A lecturer can know the students of any lecturer if the student is his/her student.

The following model SecVGU#X3 precisely specifies the above policy using SecureUML:

−roles = {Lecturer}.
−auth(Lecturer, read(Enrollment))
= auth(Lecturer, read(Student : age))
= *caller*.students → (s | s = *students*).

For our experiments, we define the function map() as below. Recall that the function map() is called by the function AuthFuncRole() that, for the case of the association Enrollment, it takes *caller*, *students*, and *lecturers* as its

Queries

In our experiments, we will use the queries Query#5 and Query#6 shown in Fig. 6, which return, respectively, the number of students whose age is greater than 18, and the number of enrollments.

We will consider three different users/callers: namely, Trang, with role Admin; and Michel and Vinh, both with role Lecturer. For the sake of our experiments, no other lecturer is older than Michel, and every student is a student of every lecturer.

Results

Query#5

To understand the execution-time lines in Fig. 7, notice that, for a policy $\mathcal{S} \in \{\texttt{SecVGU\#Xi} \mid 1 \leq i \leq 3\}$, the body of $\ulcorner \text{SecQuery}(\mathcal{S}, \texttt{Query\#5})() \urcorner$ contains the following statement:

```
CREATE TEMPORARY TABLE ⌜TempTable(age > 18)⌝ AS (
    SELECT * FROM Student
    WHERE (CASE ⌜AuthFunc(S, age)⌝ (Student_id, caller, role)
      WHEN 1 THEN age
      ELSE throw_error() END as age) > 18) );
```

In particular, notice that, to create the table ⌜TempTable(age > 18)⌝, for every tuple contained in the table Student, the function ⌜AuthFunc($\mathcal{S}$, age)⌝() is called. Logically, the execution-time for SecQuery($\mathcal{S}$, Query#5) increases depending of the "size" of the table Student. Recall also that, depending on the role r of each caller, for every student contained in the table Student, the function ⌜AuthFunc($\mathcal{S}$, age)⌝() calls the function ⌜AuthFuncRole($\mathcal{S}$, age, r)⌝(), which in turn calls the function map(auth($\mathcal{S}$, r, read(age))). Then, the execution-time for SecQuery($\mathcal{S}$, Query#5) depends also on the "complexity" of the query map(auth($\mathcal{S}$, r, read(age))) that implements auth($\mathcal{S}$, r, read(age)) in SQL, since this query will be executed for every student in the table Student. In particular, notice that, in the case of the scenario UnivX(10^3), to execute

⌜SecQuery(SecVGU#X3, Query#5)⌝({*Vinh*}, Lecturer), (1)

the query map(auth(SecVGU#X3, Lecturer, read(age))), that is

```
EXISTS ( SELECT 1 FROM Enrollment e
  WHERE e.lecturers = caller
    AND e.students = self ).
```
(2)

will be executed 10^3 times, each time with *caller* replaced by }Vinh} and *self* replaced by a different student in the table Student. Notice also that, each time the query (2) is executed, the clause

```
WHERE e.lecturers = caller
  AND e.students = self
```

will search in a table Enrollment that contains 10^6 rows.

Not surprisingly, the execution of the (secured) call (1) in the scenario UnivX(10^3) takes around 0.5 s more than the execution of the (unsecured) query Query#5.

To understand the execution-time lines in Fig. 8, notice that, for a policy $\mathcal{S} \in$ {SecVGU#Xi | $1 \leq i \leq 3$}, the body of ⌜SecQuery($\mathcal{S}$, Query#6)()⌝ contains the following create-statements:

```
CREATE TEMPORARY TABLE ⌜TempTable(True)⌝ AS (
  SELECT Student_id AS students, Lecturer_id AS lecturers
  FROM Student, Lecturer
  WHERE TRUE);
CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
  SELECT * FROM ⌜TempTable(Query#6, True)⌝ WHERE
  ( CASE ⌜AuthFunc(S, Enrollment)⌝ (students, lecturers,
        caller, role)
    WHEN 1 THEN TRUE
    ELSE throw_error() END as students );
```

In particular, notice that, to create the table ⌜TempTable(students)⌝, for every tuple contained in the table ⌜TempTable(students)⌝, which is the Cartesian product of the tables Student and Lecturer, the function ⌜AuthFunc($\mathcal{S}$, Enrollment)⌝() is called. Logically, the execution-time for SecQuery($\mathcal{S}$, Query#6) increases depending of the "size" of the tables Student and Lecturer. Recall also that, depending on the role r of each caller, for every pair of a student and a lecturer contained in ⌜TempTable(Query#6, students)⌝, the function ⌜AuthFunc($\mathcal{S}$, Enrollment)⌝() calls ⌜AuthFuncRole($\mathcal{S}$, Enrollment, r)⌝(), which in turn calls the function map(auth($\mathcal{S}$, r, read(Enrollment))). Then, the execution-time for SecQuery($\mathcal{S}$, Query#6) depends also on the "complexity" of the query map(auth($\mathcal{S}$, r, read(Enrollment))) implementing auth($\mathcal{S}$, r, read(Enrollment)) in SQL, since this query will be executed for every pair in the Cartesian product of the tables Student and Lecturer. In particular, notice that, in the case of the scenario UnivX(10^3), to execute

⌜SecQuery(SecVGU#X3, Query#6)⌝({*Vinh*}, Lecturer), (3)

the query map(auth(SecVGU#X3, Lecturer, read(Enrollment))), that is

```
EXISTS ( SELECT 1 FROM Enrollment e
  WHERE e.lecturers = caller                    (4)
    AND e.students = students ).
```

will be executed 10^6 times, each time with *caller* replaced by }*Vinh*} and *students* replaced by the student-element of a different pair of a student and a lecturer in the table `Enrollment`. Notice also that, each time the query (4) is executed, the where-clause

```
WHERE e.lecturers = caller
  AND e.students = students
```

will search in a table `Enrollment` that contains 10^6 rows. Not surprisingly, the execution of the (secured) call (3) in the scenario `UnivX`(10^3) takes around 500 s more than the execution of the (unsecured) query `Query#6`.

Scalability As mentioned before, enforcing FGAC policy for SQL queries implies performing authorization-checks at execution-time, with the consequent loss in performance. There are however situations in which we know that the required authorization-check are in fact unnecessary, because they will always return true. In our experiments, for example, in the case of the FGAC policy `SecVGU#3`, if a lecturer attempts to execute the query `Query#6` on any scenario `UnivX`(n), it is certainly unnecessary to perform any authorization-check at execution-time, because we know that every student is a student of every lecture.

Similarly, in the case of the FGAC policy `SecVGU#2`, if the lecturer `Michel` attempts to execute the query `Query#6` on any scenario `UnivX`(n), it is also unnecessary to perform any authorization-check at execution-time, because we know that there is no other lecturer older than `Michel`.

We leave as future work to develop a formal, model-based methodology for optimizing the stored-procedures generated by the function SecQuery(), based on the intended scenarios.

The SQLSI Project

The SQL Security Injector (SQLSI) is a Java application implementing our solution for enforcing FGAC policies when executing SQL queries. Figure 9 shows an overview of the SQLSI component diagram. In a nutshell, SQLSI takes three inputs, namely, a data model, an FGAC policy, and an SQL query, and returns a database schema script (generated from the input data model, according to the mapping defined in the section "Mapping data and object models to databases"), a list of authorization functions (generated from the input data model and the input FGAC policy, according to the functions AuthFunc() and AuthFuncRole() defined in the section "Enforcing FGAC policies for SQL queries"), and a secure stored-procedure (generated from the input data model and the input SQL query, according to the function SecQuery() defined in the section "Enforcing FGAC policies for SQL queries").

SQLSI is an open-source project, available at:
https://github.com/SE-at-VGU/SQLSI.

SQLSI is also available as a prototype as a multi-container Docker web-application at:
https://github.com/SE-at-VGU/SQLSI-Docker.

Related Work

Based on our model-based characterization of FGAC authorization for SQL queries [12], we have proposed here a novel model-driven approach for enforcing FGAC policies when executing SQL queries. A key feature of this approach is that it does not modify the underlying database, except for adding the stored-procedures that configure our FGAC-enforcement mechanism. This is in clear contrast with the solutions currently offered by the major commercial RDBMS. which recommend—like in the case of MySQL or MariaDB [10]—to manually create appropriate views, and to modify the queries so as to referencing these views, or request—like Oracle [3], PostgreSQL [14], and IBM [4]—to use other non-standard, proprietary enforcement mechanisms. As we have argued in [12], the solutions currently offered by the major RDBMS are far from ideal: in fact, they are time-consuming, error-prone, and scale poorly.

The second key feature of our model-driven approach is that FGAC policies and SQL queries are kept independent of each other, except for the fact that they refer to the same underlying data model. This means, in particular, that FGAC policies can be specified without knowing which SQL queries will be executed, and vice versa. This is in clear contrast with the solution recently proposed in [9] where the FGAC policies must be (re-)written depending on the SQL queries that are executed. Nevertheless, our model-driven approach certainly shares with [9], as well as with other previous approaches like [7], the idea of enforcing FGAC policies by rewriting the SQL queries, instead of by modifying the underlying databases or using non-standard, proprietary RDBMS features.

The third key-feature of our model-driven approach is that the enforcement mechanism can be automatically generated from the FGAC policies, using available mappings from OCL to SQL—for example [11]—to implement the authorization constraints appearing in the FGAC policies. In practice, however, for the sake of execution-time performance, manually implementing in SQL the authorization constraints appearing in the FGAC policies is to be preferred over using the implementations generated by the available mappings from OCL to SQL.

Conclusions and Future Work

In this paper, we have proposed a novel, model-driven approach for enforcing fine-grained access control (FGAC) policies when executing SQL queries. It is characteristic of FGAC policies that access control decisions depend on dynamic information: namely, whether the current state of the system satisfies some authorization constraints. In our approach, FGAC policies are modeled using the SecureUML language [8], in which authorization constraints are specified using the object constraint language (OCL) [13].

In a nutshell, to enforce FGAC policies when executing SQL queries we define a function SecQuery() that, given a policy $\mathcal{S}$ and a select-statement q, generates an SQL stored-procedure, such that: if a user is authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure will return the same result that executing q; otherwise, if a user is not authorized, according to $\mathcal{S}$, to execute q, then calling the stored-procedure will signal an error.

To illustrate our approach we have provided a number of non-trivial examples, involving different FGAC policies, queries, and scenarios, and we have evaluated the performance overhead incurred when executing the stored-procedured generated by SecQuery().

Finally, we have also implemented our approach in a Java application, called SQLSI, which is currently available as open-source project.

We recognize that there is still work to be done. First, we need to formally prove the correctness of the function SecQuery(), with respect to our model-based characterization of FGAC authorization for SQL queries [12]. This proof will certainly involve the formal semantics of both OCL and SQL, since authorization constraints are specified in OCL and SecQuery() generates SQL stored-procedures. Second, we need to develop a formal, model-based methodology for optimizing the stored-procedures generated by the function SecQuery(), based on the intended scenarios. In particular, whenever "safe", subqueries should be favored over temporary tables, to allow the SQL optimizer to do its job. The decision of whether it is "safe" or not to use subqueries instead of temporary tables ultimately depends on the underlying security model, and more particularly on the authorization constraints responsible in each case of the case-statements generated by SecQuery(). If these authorization constraints can be proved to be trivial, or if they can be proved to be always satisfied given the invariants of the underlying data model, then the case-statements do not need to be generated, and the corresponding temporary tables can be safely replaced by subqueries. Third, we need to extend our definition of SecQuery() to cover as much as possible of the SQL language, including, in particular, left/right-joins and group-by clauses. Last but not least, we want to provide a more abstract characterization of our approach, including a formal definition of our attacker model. In this context, we will discuss more formally how our approach relates to the traditional distinction between Truman and Non-Truman models for secure database access (our approach clearly leaning towards the latter).

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Basin DA, Clavel M, Egea M. A decade of model-driven security. In: Breu R, Crampton J, Lobo J, editors. 16th ACM symposium on access control models and technologies, SACMAT 2011, Innsbruck, Austria, June 15–17, 2011, Proceedings. ACM; 2011. p. 1–10. https://doi.org/10.1145/1998441.1998443.
2. Basin DA, Doser J, Lodderstedt T. Model driven security: from UML models to access control infrastructures. ACM Trans Softw Eng Methodol. 2006;15(1):39–91. https://doi.org/10.1145/1125808.1125810.
3. Browder K, Davidson MA. The virtual private database in Oracle9iR2. Tech. rep., Oracle Corporation; 2002. https://www.cgisecurity.com/-database/oracle/pdf/VPD9ir2twp.pdf.
4. Row and column access control support in IBM DB2 for i. Tech. rep. International Business Machines Corporation; 2014. https://www.redbooks.ibm.com/redpapers/pdfs/redp5110.pdf/.
5. Demuth B, Hußmann H, Loecher S. OCL as a specification language for business rules in database applications. In: Gogolla M, Kobryn C, editors. UML, LNCS, vol 2185. Springer; 2001. p. 104–17.
6. Ferraiolo DF, Sandhu R, Gavrila S, Kuhn DR, Chandramouli R. Proposed NIST standard for role-based access control. ACM Trans Inf Syst Secur. 2001;4(3):224–74. https://doi.org/10.1145/501978.501980.
7. LeFevre K, Agrawal R, Ercegovac V, Ramakrishnan R, Xu Y, DeWitt D. Limiting disclosure in Hippocratic databases. In: Proceedings of the thirtieth international conference on very large data bases, VLDB '04, vol 30. VLDB Endowment; 2004. p. 108–19.
8. Lodderstedt T, Basin DA, Doser J. SecureUML: a UML-based modeling language for model-driven security. In: Jézéquel J, Hußmann H, Cook S, editors. UML 2002—the unified modeling language, 5th international conference, Dresden, Germany, September 30–October 4, 2002, Proceedings, Lecture Notes in Computer Science, vol 2460. Springer; 2002. p. 426–41. https://doi.org/10.1007/3-540-45800-X_33.
9. Mehta A, Elnikety E, Harvey K, Garg D, Druschel P. Qapla: policy compliance for database-backed systems. In: Proceedings of the 26th USENIX conference on security symposium, SEC '17. USENIX Association; 2017. p. 1463–79.
10. Montee G. Row-level security in MariaDB 10: protect your data. 2015. https://mariadb.com/resources/blog/.
11. Nguyen HPB, Clavel M. OCL2PSQL: an OCL-to-SQL code-generator for model-driven engineering. In: Dang TK, Küng J, Takizawa M, Bui SH, editors. Future data and security engineering—6th international conference, FDSE 2019, proceedings, lecture notes in computer science, vol 11814. Springer; 2019. p. 185–203.

12. Nguyen HPB, Clavel M. Model-based characterization of fine-grained access control authorization for SQL queries. J Object Technol. 2020;19(3).
13. Object Constraint Language specification version 2.4. Tech. rep. Object Management Group; 2014. https://www.omg.org/spec/OCL/.
14. PostgreSQL 12.2. Part II. SQL The Language. Chapter 5. Data Definition. 5.8. Row Security Policies. 2017. https://www.postgresql.org/docs/10/ddl.html.
15. ISO/IEC 9075-(1–10) Information technology—database languages—SQL. Tech. rep. International Organization for Standardization; 2011. http://www.iso.org/iso/.