JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Model-based characterization of fine-grained access control authorization for SQL queries

Hoàng Nguyễn Phước Bảo and Manuel Clavel
Vietnamese-German University, Vietnam

ABSTRACT We propose a model-based characterization of fine-grained access control (FGAC) authorization for SQL queries. More specifically, we define a predicate $\mathrm{AuthQuery}()$ that represents whether a user is authorized by an FGAC-policy to execute a SQL query on a database. 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 proposal, FGAC-policies are modeled using a dialect of SecureUML, and authorization constraints are specified using the Object Constraint Language (OCL). To illustrate our definition of the predicate $\mathrm{AuthQuery}()$, we provide examples of authorization decisions for different SQL queries, attempted by different users, in different scenarios, and with respect to different FGAC-policies. Interestingly, the availability of mappings from OCL to SQL opens up the possibility of implementing $\mathrm{AuthQuery}()$ within the database and, consequently, of enforcing FGAC-policies following a model-driven approach.

KEYWORDS Model-driven security, SQL, Fine-grained access control, Authorization, SecureUML, OCL.

1. Introduction

The ever-growing development and use of information and communication technology is a constant source of security and reliability problems. Clearly, better ways of developing software systems and approaching software engineering as a well-founded engineering discipline is needed.

Model-Driven Engineering is a software development methodology that focuses on creating models of different views of a system, and then automatically generating different system artifacts from these models, such as code and configuration data. Model-Driven Security (MDS) (Basin et al. 2006, 2011) is a specialization of model-driven engineering for developing secure systems. In a nutshell, designers specify system models along with their security requirements and use tools to automatically generate security-related system artifacts, such as access control infrastructures.

SecureUML (Lodderstedt et al. 2002) 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* in the current state of the system. Typically, authorization constraints are specified in SecureUML models using the Object Constraint Language (OCL) (OCL 2014).[1]

The Structure Query Language (SQL) (SQLISO 2011) 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. For data access control, standard RDBMS do not easily support FGAC policies. In fact, to the best of our knowledge, no formal characterization of FGAC authorization for SQL queries has been proposed yet. In this paper, we aim to fill this critical gap by providing a model-based characterization of FGAC authorization for a large class of SQL queries. Concretely, we define a predicate $\mathrm{AuthQuery}()$

JOT reference format:
Hoàng Nguyễn Phước Bảo and Manuel Clavel. *Model-based characterization of fine-grained access control authorization for SQL queries*. Journal of Object Technology. Vol. 19, No. 3, 2020.
http://dx.doi.org/10.5381/jot.2020.19.3.a15

[1] The *context* of an authorization constraint in a SecureUML model is each policy's underlying data model. As such, (history-based) SoD constraints are not easily modelled using SecureUML.

that represents whether a user is authorized to execute a SQL query, according to an FGAC-policy specified in a SecureUML model. To illustrate our definition, we provide examples of authorization decisions for different SQL queries, attempted by different users, in different scenarios, and with respect to different FGAC-policies. We envision the possibility of implementing the predicate AuthQuery() in SQL —by making use of the mapping OCL2PSQL from OCL to SQL (Nguyen & Clavel 2019)— and, consequently, of being able to enforce FGAC-policies in SQL databases, following a model-driven approach.[2]

Organization The rest of the paper is organized as follows. In Section 2 we provide our basic definitions of data models and object models, and a short description of OCL. In Section 3 we define our mappings from data models and object models to SQL. Then, in Section 4 we define our concrete semantics for SecureUML, by providing a predicate Auth() that represents, for each security model, whether a user is authorized to execute an action on an object model. Next, in Section 5, we propose our model-based characterization of FGAC-authorization for SQL queries. Concretely, we define a predicate AuthQuery() that represents, given an FGAC-policy, modeled using SecureUML, whether a user is authorized to execute a SQL query on a database. As expected, the definition of AuthQuery() critically uses the predicate Auth() defined in Section 4. Finally, we conclude, with an extended related work in Section 6 and detailed discussion on future work in Section 7.

2. Modeling data

In modeling access control policies, we use data models to specify the data to be protected. In this section we define our notions of data and object models. We also introduce below the data and object models that will be used in our examples. We end this section with a brief description of OCL.

Definition 1. *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, $as = \langle asi, ase_l, c_l, ase_r, c_r\rangle$, where asi is the association's identifier, ase_l and ase_r are the association's ends, and c_l and c_r are the classes of the objects at the corresponding association's ends.*

For simplicity's sake, we only consider `Integer` and `String` as our predefined types.

[2] Interestingly, this possibility was already foreseen in (Lodderstedt et al. 2002), the seminal paper on MDS and SecureUML: "To begin with, security requirements can be formulated and integrated into system designs at a high level of abstraction. In this way, it becomes possible to develop security aware applications that are designed with the goal of preventing violations of a security policy. For example, a database query can be designed so that users can only retrieve those data records that they are allowed to access".

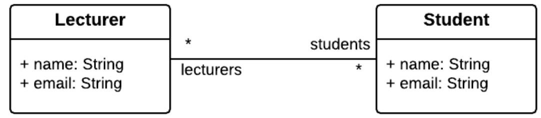

Figure 1 The `University` model

Example 1 (The data model `University`)**.** As a basic example, we introduce in Figure 1 the data model `University`. It contains two classes, `Student` and `Lecturer`, and one association `Enrollment` between both of them. The classes `Student` and `Lecturer` have both attributes `name` and `email`. The class `Student` represents the students of the university, with their name and email. The class `Lecturer` represents the lecturers of the university, with their name and email. The association `Enrollment` represents the relationship between the students (denoted by `students`) and the lecturers (denoted by `lecturers`) of the courses the students have enrolled in. More formally, the data model `University` is a tuple containing the set of classes {`Lecturer`, `Student`}, the set of attributes { ⟨`name`, `Lecturer`, `String`⟩, ⟨`email`, `Lecturer`, `String`⟩, ⟨`name`, `Student`, `String`⟩, ⟨`email`, `Student`, `String`⟩}, and the set of associations { ⟨`Enrollment`, `lecturers`, `Lecturer`, `students`, `Student`⟩}.

Definition 2. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. An* object model *$\mathcal{O}$ of $\mathcal{D}$ (also called an* instance *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 object's identifier and c is the class of the object, where $c \in C$.*
- *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. The attribute value atv denotes the value vl of the attribute $\langle ati, c, t\rangle$ of the object $\langle oi, c\rangle$.*
- *OAS is a set of* association links *asl, $asl = \langle\langle asi, ase_l, c_l, ase_r, c_r\rangle, \langle oi_l, c_l\rangle, \langle oi_r, c_r\rangle\rangle$, where $\langle asi, ase_l, c_l, ase_r, c_r\rangle \in AS$, $\langle oi_l, c_l\rangle \in OC$, and $\langle oi_r, c_r\rangle \in OC$. The association link asl denotes that there is a link of the association $\langle asi, ase_l, c_l, ase_r, c_r\rangle$ between the objects $\langle oi_l, c_l\rangle$ and $\langle oi_r, c_r\rangle$, where the later stands at the end ase_r and the former stands at the end ase_l.*

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

Example 2. Consider the following instance `VGU#1` of the data model `University` (Example 1). It contains five students: `Chau`, `An`, `Thanh`, `Nam`, and `Hoang`, with the expected names and emails (*name*`@vgu.edu.vn`). It contains also three lecturers: `Huong`, `Manuel`, `Hieu`, again with the expected names and emails (*name*`@vgu.edu.vn`). Finally, there are links of the association `Enrollment` between the lecturer `Manuel` and the students `Chau`, `An`, and `Hoang`, and also between the lecturer `Huong` and the students `Chau` and `Thanh`.

Example 3. Consider an instance `VGU#2` of the data model `University` (Example 1), which is exactly as `VGU#1` except for including two additional links of the association `Enrollment`: one between the lecturer `Hieu` and the student `Thanh` and the other between the lecturer `Hieu` and the student `Nam`.

2.1. Object Constraint Language (OCL)

OCL (OCL 2014) 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 strongly typed. 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 to collect the objects linked with an object at the end of an association. For example, suppose that the contextual model includes a class *c* with an attribute *at* and an association-end *ase*. Then, if *o* is an object of the class *c*, the expression *o*.*at* refers to the value of the attribute *at* of the object *o*, and *o*.*ase* refers to the objects linked to the object *o* at the association-end *ase*. OCL 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, namely, `null` and `invalid`. Intuitively, `null` represents an unknown or undefined value, whereas `invalid` represents an error or an exception.

Notation. Let $\mathcal{D}$ be a data model. We denote by $\mathrm{Exp}(\mathcal{D})$ the set of OCL expressions whose contextual model is $\mathcal{D}$. Let $\mathcal{O}$ be an instance of $\mathcal{D}$, and let *exp* be an OCL expression in $\mathrm{Exp}(\mathcal{D})$. Then, we denote by $\mathrm{Eval}(\mathcal{O}, exp)$ the result of evaluating *exp* in $\mathcal{O}$ according to the semantics of OCL.

3. Mapping data and object models to databases

In characterizing access control authorization for SQL queries, we assume that SQL queries are executed on databases that implement the policies' underlying data models, as well as the object models of interest, according to the mappings defined below.[3]

Definition 3. *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 $\mathrm{SqlType}(t)$;

where:

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

Moreover, if $t \in C$, then

`ALTER TABLE` c `ADD FOREIGN KEY fk_`c`_`ati`(`ati`)`
  `REFERENCES` t`(`t`_id);`

– *For every association $as \in AS$, $as = \langle asi, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$,*

`CREATE TABLE` asi `(`
  ase_{l} `varchar NOT NULL,`
  ase_{r} `varchar NOT NULL,`
  `FOREIGN KEY fk_`c_{l}`_`ase_{l}`(`ase_{l}`)`
    `REFERENCES` c_{l}`(`c_{l}`_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_l`,` ase_r`);`

Definition 4. *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 $asl \in OAS$, $asl = \langle\langle asi, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle, \langle oi_{\mathrm{l}}, c_{\mathrm{l}}\rangle, \langle oi_{\mathrm{r}}, c_{\mathrm{r}}\rangle\rangle$,*

`INSERT INTO` asi `(`ase_{l}`,` ase_{r}`) VALUES (`oi_{l}`,` oi_{r}`);`

Notation. Let $\mathcal{D}$ be a data model. Let $\mathcal{O}$ be an object model of $\mathcal{D}$. Let q be a SQL query on $\overline{\mathcal{D}}$. We denote by $\mathrm{Exec}(\overline{\mathcal{O}}, q)$ the result of executing q in $\overline{\mathcal{O}}$ according to the semantics of SQL.

The following remark makes explicit the key property of our mapping from object models to SQL:

Remark 1. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{O} = \langle OC, OAT, OAS\rangle$ be an instance of D. Let $\langle ati, c, t\rangle$ be an attribute in AT, and let $\langle oi, c\rangle$ be an object in OC. Then:*

$$\mathrm{Eval}(\mathcal{O}, oi.ati) \\ = \mathrm{Exec}(\overline{\mathcal{O}}, \texttt{SELECT}\ ati\ \texttt{FROM}\ c\ \texttt{WHERE}\ c\_id = oi).$$

Let $\langle asi, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$ be an association in AS, and let $\langle oi_{\mathrm{l}}, c_{\mathrm{l}}\rangle$ and $\langle oi_{\mathrm{r}}, c_{\mathrm{r}}\rangle$, be objects in OC. Then,

$$\mathrm{Eval}(\mathcal{O}, oi_{\mathrm{l}}.ase_{\mathrm{l}}) \\ = \mathrm{Exec}(\overline{\mathcal{O}}, \texttt{SELECT}\ ase_{\mathrm{l}}\ \texttt{FROM}\ asi\ \texttt{WHERE}\ ase_{\mathrm{r}} = oi_{\mathrm{r}}),$$

and

$$\mathrm{Eval}(\mathcal{O}, oi_{\mathrm{r}}.ase_{\mathrm{r}}) \\ = \mathrm{Exec}(\overline{\mathcal{O}}, \texttt{SELECT}\ ase_{\mathrm{r}}\ \texttt{FROM}\ asi\ \texttt{WHERE}\ ase_{\mathrm{l}} = oi_{\mathrm{l}}).$$

[3] Notice that other mappings from data models to SQL are possible (Demuth et al. 2001). If a different mapping from data models to SQL is chosen, then our characterizing of access control authorization for SQL queries should be changed accordingly.

4. Modeling fine-grained access control policies

In this section, we first introduce SecureUML (Lodderstedt et al. 2002) and then define the meaning of SecureUML models by providing a predicate $\mathrm{Auth}()$ that represents, for each security model, whether a user is authorized to execute an action on an object model. Logically, the predicate $\mathrm{Auth}()$ plays a key role in our characterization of access control authorization for SQL queries with respect to SecureUML models.

SecureUML is a modeling language for specifying access control policies on protected *resources*. In SecureUML, resources are protected by controlling the *actions* that provide access to them. However, SecureUML leaves open the nature of the protected resources, —i.e., whether these resources are data, business objects, processes, controller states, etc.— and, consequently, of the corresponding controlled actions. These are to be declared in a so-called SecureUML dialect. Next we define the actions that we consider in our SecureUML dialect:

Definition 5. *Let $\mathcal{D}$ be a data model $\mathcal{D} = \langle C, AT, AS\rangle$. Then, we denote by* $\mathrm{Act}(\mathcal{D})$ *the following set of* read-actions*:*

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

Definition 6. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{O}$ = $\langle OC, OAT, OAS\rangle$ be an instance of $\mathcal{D}$. Then, we denote by* $\mathrm{Act}(\mathcal{O})$ *the following set of* instance read-actions*:*

– *For every attribute $at = \langle ati, c, t\rangle$, $at \in AT$, and every object $o = \langle oi, c\rangle$, $o \in OC$, the action* $\mathrm{read}(at, o)$ *of reading the value of the attribute at in o.*
– *For every association $as = \langle asi, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$, $as \in AS$, and every pair of objects $o_{\mathrm{l}} = \langle oi_{\mathrm{l}}, c_{\mathrm{l}}\rangle$, $o_{\mathrm{r}} = \langle oi_{\mathrm{r}}, c_{\mathrm{r}}\rangle$, such that $o_{\mathrm{l}}, o_{\mathrm{r}} \in OC$, the action* $\mathrm{read}(as, o_{\mathrm{l}}, o_{\mathrm{r}})$ *of reading if there is a link of the association as between o_l and o_r.*

As a language for specifying access control policies, SecureUML is an extension of Role-Based Access Control (RBAC) (Ferraiolo et al. 2001). 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* in the current state of the system. Authorization constraints are specified in SecureUML models using OCL expressions. Concretely, in our SecureUML dialect, we consider authorization constraints whose satisfaction depend on information related to: (i) the object who is attempting to perform the read-action; (ii) the object whose attribute is attempted to be read; and, (iii) the objects whose association 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) by using as keywords the corresponding association-ends.

Next we define the notion of security models in our SecureUML dialect, and introduce the security models that will be used in our examples.

Definition 7. *Let $\mathcal{D}$ be a data model. Then, a* security model $\mathcal{S}$ *for $\mathcal{D}$ is a tuple $\mathcal{S} = (R, \mathrm{auth})$, where R is a set of roles, and* $\mathrm{auth}\colon R \times \mathrm{Act}(\mathrm{D}) \longrightarrow \mathrm{Exp}(\mathcal{D})$ *is a function that assigns to each role $r \in R$ and each action $a \in \mathrm{Act}(\mathcal{D})$ an authorization constraint $exp \in \mathrm{Exp}(\mathcal{D})$.*

Example 4. Consider the following security model `SecVGU#A` for the data model `University`.

– Roles. There is only one role, namely, the role `Lecturer`. Lecturers are assigned this role.
– Permissions:
  - Any lecturer can know his/her students. Formally, for this model, $\mathrm{auth}(\mathtt{Lecturer}, \mathrm{read}(\mathtt{Enrollment}))$ is

    $\mathtt{lecturers} = \mathtt{caller}$.
  - Any lecturer can know his/her own email, as well as the emails of his/her students. Formally, for this model, $\mathrm{auth}(\mathtt{Lecturer}, \mathrm{read}(\mathtt{email}))$ is

    $(\mathtt{caller} = \mathtt{self})$
    $\mathtt{or}\ (\mathtt{caller.students} \rightarrow \mathtt{includes}(\mathtt{self}))$

Example 5. Consider the security model `SecVGU#B` for the data model `University`, which is exactly as `SecVGU#A` except for including the following additional clauses:

– Permissions:
  - Any lecturer can know his/her colleagues' emails. For the sake of this example, two lecturers are *colleagues* if there is at least one student enrolled with both of them. Formally, for this model, $\mathrm{auth}(\mathtt{Lecturer}, \mathrm{read}(\mathtt{email}))$ is

    $(\mathtt{caller} = \mathtt{self})$
    $\mathtt{or}\ (\mathtt{caller.students} \rightarrow \mathtt{includes}(\mathtt{self}))$
    $\mathtt{or}\ (\mathtt{caller.students} \rightarrow \mathtt{exists}$
    $(\mathtt{s}|\mathtt{s.lecturers} \rightarrow \mathtt{includes}(\mathtt{self})))$.

Example 6. Consider the security model `SecVGU#C` for the data model `University`, which is exactly as `SecVGU#B` except for including the following additional clauses:

– *Permissions*:
  - Any lecturer can know the lecturers of his/her own students. Formally, for this model, $\mathrm{auth}(\mathtt{Lecturer}, \mathrm{read}(\mathtt{Enrollment}))$ is

    $(\mathtt{lecturers} = \mathtt{caller})$
    $\mathtt{or}(\mathtt{caller.students} \rightarrow \mathtt{includes}(\mathtt{students}))$.

Finally, we formalize the semantics of our security models by defining a predicate $\mathrm{Auth}()$ that represents, for a given model, whether a user is authorized to execute a read-action on a scenario.

Definition 8. *Let $\mathcal{D}$ be a data model. Let $\mathcal{S} = \langle R, \mathrm{auth}\rangle$ be a security model for $\mathcal{D}$. Let r be a role in R. Let $\mathcal{O}$ = $\langle OC$, OAT, $OAS\rangle$ be an object model of $\mathcal{D}$. Let u be an object in OC. Then, we define the predicate* Auth *as follows:*

– *For any action* $\mathrm{read}(at, o) \in \mathrm{Act}(\mathcal{O})$,

$$\mathrm{Auth}(\mathcal{S}, \mathcal{O}, u, r, \mathrm{read}(at, o))$$
$$\Longleftrightarrow \mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(at)) \begin{bmatrix} \mathtt{self} \mapsto o \\ \mathtt{caller} \mapsto u \end{bmatrix}).$$

– *For any action* $\mathrm{read}(as, o_{\mathrm{l}}, o_{\mathrm{r}}) \in \mathrm{Act}(\mathcal{O})$,

$$\mathrm{Auth}(\mathcal{S}, \mathcal{O}, u, r, \mathrm{read}(as, o_{\mathrm{l}}, o_{\mathrm{r}}))$$
$$\Longleftrightarrow \mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(as)) \begin{bmatrix} as_{\mathrm{l}} \mapsto o_{\mathrm{l}} \\ as_{\mathrm{r}} \mapsto o_{\mathrm{r}} \\ \mathtt{caller} \mapsto u \end{bmatrix}).$$

To illustrate the definition of the predicate $\mathrm{Auth}()$, we show in Tables 1–2 the different values of this predicate, for the same actions, but with different users (*callers*), on different scenarios, and for different security models.

5. Model-based SQL query authorization

In this section, we propose a model-based characterization of FGAC-authorization for SQL queries. More specifically, we define a predicate $\mathrm{AuthQuery}()$ that checks, given an FGAC-policy modelled using our SecureUML dialect, whether a user is authorized to execute a SQL query on a database.

We organize this section as follows. First, we motivate with examples some of the problems we aim to address when defining the predicate $\mathrm{AuthQuery}()$. Then, we present the list of SQL query-patterns currently covered by our definition. Next, for each of these query-patterns, we define the predicate $\mathrm{AuthQuery}()$, including in each case several examples of authorization decisions that illustrate the intent of the definition. Finally, we revisit the examples used to motivate our definition of the predicate $\mathrm{AuthQuery}()$, with a preliminary case study.

Motivation Informally, a user can be authorized to execute a SQL on a database if the execution of this query does not leak confidential information, according to a given FGAC policy. Interestingly, checking whether a user is authorized to execute a SQL query on a database implies much more than simply checking that the final result satisfies the given FGAC policy, since a clever attacker can devise a query such that the simple fact that a final result is obtained reveals already some additional information, which maybe confidential.

Consider the select-statements in Figures 2–4. For the sake of this example, suppose that, for a given scenario, the three of them return the same final result, namely, a non-empty string, representing an email, which is not confidential. On a closer examination, however, we can realize that, for each of these select-statements, the final result is revealing *additional* information, which may in turn 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.

```
mysql> SELECT email FROM Lecturer
->  WHERE Lecturer_id = 'Huong'
```

Figure 2 Example. `Query#1`.

– `Query#3` reveals that the email belongs to Huong, and that Huong and Manuel are "colleagues", in the sense that there some students who have both Huong and Manuel as their lecturers.

As the above example shows, in order to authorize a user, according to an FGAC policy, to execute a query, it is not enough to simply check that displaying the final result is policy-compliance. On the contrary, we claim that *any* information that is *used* to reach this final result (in particular, information involved in subqueries, where-clauses, and on-clauses) should be also checked for policy-compliance. In this way, for example, if a user is not authorized to know whether Huong is Thanh's lecturer or not, then he/she should not be authorized to execute `Query#2`, 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" or not, then, he/she should not be authorized to execute `Query#3`, even when he/she may be authorized to access lecturers' emails.

```
mysql> SELECT DISTINCT email FROM Lecturer
->JOIN (SELECT * from Enrollment
->      WHERE students = 'Thanh'
->      AND lecturers = 'Huong'
->      ) as TEMP
->ON TEMP.lecturers = Lecturer_id
```

Figure 3 Example. `Query#2`.

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

Figure 4 Example. `Query#3`.

Scope Our definition of the predicate AuthQuery currently covers the following query "patterns", where *c* and *as* denote, respectively, a class and an association in the underlying data model.

– `SELECT` *selitems* `FROM` *c* `WHERE` *exp*.

		SecVGU#A		SecVGU#B		SecVGU#C	
caller	*action*	VGU#1	VGU#2	VGU#1	VGU#2	VGU#1	VGU#2
Manuel	read(email, Manuel)	✓	✓	✓	✓	✓	✓
Manuel	read(email, Huong)	✗	✗	✓	✓	✓	✓
Manuel	read(email, Hieu)	✗	✗	✗	✗	✗	✗
Huong	read(email, Manuel)	✗	✗	✓	✓	✓	✓
Huong	read(email, Huong)	✓	✓	✓	✓	✓	✓
Huong	read(email, Hieu)	✗	✗	✗	✓	✗	✓
Hieu	read(email, Manuel)	✗	✗	✗	✗	✗	✗
Hieu	read(email, Huong)	✗	✗	✗	✓	✗	✓
Hieu	read(email, Hieu)	✓	✓	✓	✓	✓	✓

Table 1 The predicate Auth(): lecturers attempting to read lecturers' emails.

- `SELECT` *selitems* `FROM` *as* `WHERE` *exp*.
- `SELECT` *selitems* `FROM` *subselect* `WHERE` *exp*.
- `SELECT` *selitems* `FROM` *c* `JOIN` *as* `ON` *exp* `WHERE` *exp'* (and, vice versa, `SELECT` *selitems* `FROM` *as* `JOIN` *c* `ON` *exp* `WHERE` *exp'*).
- `SELECT` *selitems* `FROM` *c* `JOIN` *subselect* `ON` *exp* `WHERE` *exp'* (and, vice versa, `SELECT` *selitems* `FROM` *subselect* `JOIN` *c* `ON` *exp* `WHERE` *exp'*).
- `SELECT` *selitems* `FROM` *as* `JOIN` *subselect* `ON` *exp* `WHERE` *exp'* (and, vice versa, `SELECT` *selitems* `FROM` *subselect* `JOIN` *as* `ON` *exp* `WHERE` *exp'*).
- `SELECT` *selitems* `FROM` $subselect_1$ `JOIN` $subselect_2$ `ON` *exp* `WHERE` *exp'*.

Preliminaries In the definition of our predicate $\text{AuthQuery}()$, we use the following auxiliary functions.

- $\text{PropsInSel}(selitems)$: the set of *properties* (i.e., attributes and association-ends) that appear in a list of selected items.
- $\text{PropsInWhe}(exp)$: the set of *properties* (i.e., attributes and association-ends) that appear in a where-expression.
- $\text{PropsInOn}(exp)$: the set of *properties* (i.e., attributes and association-ends) that appear in an on-expression.
- $\text{CompWithInOn}(exp, ase)$: the *property* that is compared with *ase* in an on-expression.

5.0.1. Definition Let $\mathcal{D}$ be a data model. Let $\mathcal{O}$ be an object model of $\mathcal{D}$. Let $\mathcal{S} = (R, \text{auth})$ be a security model for $\mathcal{D}$. Let q be a SQL query in $\overline{\mathcal{D}}$. Let r be a role in R. Let u be a user. Then, we define the predicate $\text{AuthQuery}()$ as follows:

Case q = `SELECT` *selitems* `FROM` *c* `WHERE` *exp*. Then, $\text{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, q)$ holds if and only if:

- For every $o \in \text{Exec}(\overline{\mathcal{O}}$, `SELECT` *c*`_id FROM` *c*),
  - For every attribute $at = \langle ati, c, t\rangle$, such that $ati \in \text{PropsInWhe}(exp)$, it holds that:

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, read(at, o)).$$

- For every $o \in \text{Exec}(\overline{\mathcal{O}}$, `SELECT` *c*`_id FROM` *c* `WHERE` *exp*),
  - For every attribute $at = \langle ati, c, t\rangle$, such that $ati \in \text{PropsInSel}(selitems)$, it holds that:

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, read(at, o)).$$

Example 7. Consider the following SQL query:

```
SELECT Lecturer_id
FROM Lecturer;
```

For any policy `SecVGU#`[`A`|`B`|`C`], and any instance of the data model `VGU`, all lecturers will be authorized to execute this query.

Example 8. Consider the following SQL query:

```
SELECT 1
FROM Lecturer;
```

For any policy `SecVGU#`[`A`|`B`|`C`], and any instance of the data model `VGU`, all lecturers will be authorized to execute this query.

Example 9. Consider the following SQL query:

```
SELECT email
FROM Lecturer;
```

For policy `SecVGU#A`, for any scenario `VGU#`[1|2], none of the lecturers are authorized to execute this query. Also, for any policy `SecVGU#`[`B`|`C`] and scenario `VGU#1`, none of the lecturers are authorized to execute this query. However, for any policy `SecVGU#`[`B`|`C`] and scenario `VGU#2`, Huong is authorized to execute this query (but only her).

		SecVGU#A		SecVGU#B		SecVGU#C	
caller	*action*	VGU#1	VGU#2	VGU#1	VGU#2	VGU#1	VGU#2
Manuel	`read`(enroll, Manuel, *any student*)	✓	✓	✓	✓	✓	✓
	`read`(enroll, Huong, Chau)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Huong, An)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Huong, Thanh)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Huong, Hoang)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Huong, Nam)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Hieu, Chau)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Hieu, An)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Hieu, Thanh)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Hieu, Hoang)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Hieu, Nam)	✗	✗	✗	✗	✗	✗
Huong	`read`(enroll, Manuel, Chau)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Manuel, An)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Manuel, Thanh)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Manuel, Hoang)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Manuel, Nam)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Huong, *any student*)	✓	✓	✓	✓	✓	✓
	`read`(enroll, Hieu, Chau)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Hieu, An)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Hieu, Thanh)	✗	✗	✗	✗	✓	✓
	`read`(enroll, Hieu, Hoang)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Hieu, Nam)	✗	✗	✗	✗	✗	✗
Hieu	`read`(enroll, Manuel, Chau)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Manuel, An)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Manuel, Thanh)	✗	✗	✗	✗	✗	✓
	`read`(enroll, Manuel, Hoang)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Manuel, Nam)	✗	✗	✗	✗	✗	✓
	`read`(enroll, Huong, Chau)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Huong, An)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Huong, Thanh)	✗	✗	✗	✗	✗	✓
	`read`(enroll, Huong, Hoang)	✗	✗	✗	✗	✗	✗
	`read`(enroll, Huong, Nam)	✗	✗	✗	✗	✗	✓
	`read`(enroll, Hieu, *any student*)	✓	✓	✓	✓	✓	✓

Table 2 The predicate Auth(): lecturers attempting to read lecturers' enrolled students.

Example 10. Consider the following SQL query:

```
SELECT email
FROM Lecturer
WHERE Lecturer_id = 'Huong';
```

For policy SecVGU#A, for scenario VGU#[1|2], only Huong is authorized to execute this query. For any policy SecVGU#[B|C], for scenario VGU#1, both Huong and Manuel are authorized to execute this query. But, for any policy SecVGU#[B|C], for scenario VGU#2, all lecturers are authorized to execute this query.

Case q = SELECT *selitems* FROM *as* WHERE *exp*, where $as = \langle asi, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$. Then, $\mathrm{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, q)$ holds if and only if:

- For every $(o_{\mathrm{l}}, o_{\mathrm{r}}) \in \mathrm{Exec}(\overline{\mathcal{O}}$, SELECT c_{l}_id, c_{r}_id FROM c_{l}, c_{r} WHERE *exp*), it holds that:

$$\mathrm{Auth}(\mathcal{S}, \mathcal{O}, u, r, read(as, o_{\mathrm{l}}, o_{\mathrm{r}})).$$

Example 11. Consider the following SQL query:

```
SELECT lecturers
FROM Enrollment;
```

For any policy SecVGU#[A|B|C], for any scenario VGU#[1|2], none of the lecturers are authorized to execute this query. However, for any scenario with no students, all the lecturers will be authorized to execute this query.

Example 12. Consider the following SQL query:

```
SELECT 1
FROM Enrollment;
```

For any policy SecVGU#[A|B|C], for any scenario VGU#[1|2], none of the lecturers are authorized to execute this query. However, for any scenario with no students, all the lecturers will be authorized to execute this query.

Example 13. Consider the following SQL query:

```
SELECT students
FROM Enrollment
WHERE lecturers = 'Huong';
```

For any policy SecVGU#[A|B|C], for any scenario VGU#[1|2], only Huong is authorized to execute this query.

Example 14. Consider the following SQL query:

```
SELECT lecturers
FROM Enrollment
WHERE lecturers = students;
```

For any policy SecVGU#[A|B|C], for any scenario VGU#[1|2], all of the lecturers are authorized to execute this query. Notice that this is so, in scenarios VGU#[1|2], the set of lecturers who are their own students is empty.

Example 15. Consider the following SQL query:

```
SELECT students
FROM Enrollment
WHERE lecturers = 'Hieu';
```

For any policy SecVGU#[A|B|C], for any scenario VGU#[1|2], only Hieu is authorized to execute this query. Notice that this is so, even when, in scenario VGU#1, the set of students enrolled with Hieu is empty.

Case q = SELECT *selitems* FROM *subselect* WHERE *exp*. Then, $\mathrm{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, q)$ holds if and only if $\mathrm{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, subselect)$ holds.

Example 16. Consider the following SQL query:

```
SELECT TEMP.Lecturer_id
FROM (SELECT Lecturer_id, email
  FROM Lecturer) AS TEMP;
```

For policy SecVGU#A, for any scenario VGU#[1|2], none of the lecturers are authorized to execute this query. Also, for any policy SecVGU#[B|C] and scenario VGU#1, none of the lecturers are authorized to execute this query. However, for any policy SecVGU#[B|C] and scenario VGU#2, Huong is authorized to execute this query (but only her). Notice that this is so, even when this query is "equivalent" to the one in Example 7, which all lecturers are authorized to execute in all circumstances.

Example 17. Consider the following SQL query:

```
SELECT TEMP.email
FROM (SELECT email
  FROM Lecturer
  WHERE Lecturer_id = 'Huong') AS TEMP;
```

For policy SecVGU#A, for any scenario VGU#[1|2], only Huong is authorized to execute this query. For any policy SecVGU#[B|C], for scenario VGU#1, both Huong and Manuel are authorized to execute this query. Then, for any policy SecVGU#[B|C], for scenario VGU#2, all lecturers are authorized to execute this query.

Example 18. Consider the following SQL query:

```
SELECT TEMP.email
FROM (SELECT email
  FROM Lecturer) AS TEMP
WHERE TEMP.Lecturer_id = 'Huong';
```

For policy SecVGU#A, for any scenario VGU#[1|2], none of the lecturers are authorized to execute this query. Also, for any policy SecVGU#[B|C] and scenario VGU#1, none of the lecturers are authorized to execute this query. However, for any policy SecVGU#[B|C] and scenario VGU#2, Huong is authorized to execute this query (but only her). Notice that this is so, even when this query is "equivalent" to the one in Example 17, which other lecturers beside Huong are authorized to execute for some policies SecVGU#[A|B|C], and some scenarios VGU#[1|2].

Case q = `SELECT` *selitems* `FROM` $c_{[\mathrm{l}|\mathrm{r}]}$ `JOIN` *as* `ON` *exp* `WHERE` exp', where $as = \langle asi,\ ase_{\mathrm{l}},\ c_{\mathrm{l}},\ ase_{\mathrm{r}},\ c_{\mathrm{r}}\rangle$. Then, $\mathrm{AuthQuery}(\mathcal{S},\overline{\mathcal{O}},u,r,q)$ holds if and only if:

- For every $o_{[\mathrm{l}|\mathrm{r}]} \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` $c_{[\mathrm{l}|\mathrm{r}]}$`_id FROM` $c_{[\mathrm{l}|\mathrm{r}]}$),
  - For every attribute $at = \langle ati,\ c_{[\mathrm{l}|\mathrm{r}]},\ t\rangle$, such that $ati \in \mathrm{PropsInOn}(exp)$, it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(at,o_{[\mathrm{l}|\mathrm{r}]})).$$
- For every $(o_{\mathrm{l}}, o_{\mathrm{r}}) \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` c_{l}`_id`, c_{r}`_id FROM` $c_{\mathrm{l}}, c_{\mathrm{r}}$), it holds that:
$$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(as,o_{\mathrm{l}},o_{\mathrm{r}})).$$
- For every $o_{[\mathrm{l}|\mathrm{r}]} \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` $c_{[\mathrm{l}|\mathrm{r}]}$`_id FROM` $c_{[\mathrm{l}|\mathrm{r}]}$ `JOIN` *as* `ON` *exp*),
  - For every attribute $at = \langle ati,\ c_{[\mathrm{l}|\mathrm{r}]},\ t\rangle$, such that $ati \in \mathrm{PropsInWhe}(exp')$, it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(at,o_{[\mathrm{l}|\mathrm{r}]})).$$
- For every $o_{[\mathrm{l}|\mathrm{r}]} \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` $c_{[\mathrm{l}|\mathrm{r}]}$`_`*id* `FROM` $c_{[\mathrm{l}|\mathrm{r}]}$ `JOIN` *as* `ON` *exp* `WHERE` exp'),
  - For every attribute $ati = \langle ati,\ c_{[\mathrm{l}|\mathrm{r}]},\ t\rangle$, with $ati \in \mathrm{PropsInSel}(selitems)$ it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(at,o_{[\mathrm{l}|\mathrm{r}]})).$$

Example 19. Consider the following SQL query:

```
SELECT email
FROM Lecturer
JOIN Enrollment
ON Lecturer_id = lecturers;
```

For any policy `SecVGU#`[`A`|`B`|`C`], for any scenario `VGU#`[`1`|`2`], none of the lecturers are authorized to execute this query.

Example 20. Consider the following SQL query:

```
SELECT email
FROM Lecturer
JOIN Enrollment
ON Lecturer_id = 'Huong';
```

For any policy `SecVGU#`[`A`|`B`|`C`], for any scenario `VGU#`[`1`|`2`], none of the lecturers are authorized to execute this query.

Example 21. Consider the following SQL query:

```
SELECT email
FROM Lecturer
JOIN Enrollment
ON Lecturer_id = lecturers
WHERE lecturers = 'Huong';
```

For any policy `SecVGU#`[`A`|`B`|`C`], for any scenario `VGU#`[`1`|`2`], none of the lecturers are authorized to execute this query.

Case q = `SELECT` *selItems* `FROM` *c* `JOIN` *subselect* `ON` *exp* `WHERE` exp'. Then, $\mathrm{AuthQuery}(\mathcal{S},\overline{\mathcal{O}},u,r,q)$ holds if and only if $\mathrm{AuthQuery}(\mathcal{S},\overline{\mathcal{O}},u,r,subselect)$ holds and

- For every $o \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` *c*`_id FROM` *c*),
  - For every attribute $at = \langle ati,\ c,\ t\rangle$, such that $ati \in \mathrm{PropsInOn}(exp)$, it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(at,o)).$$
- For every $o \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` *c*`_id FROM` *c* `JOIN` *subselect* `ON` *exp*),
  - For every attribute $at = \langle ati,\ c,\ t\rangle$, such that $ati \in \mathrm{PropsInWhe}(exp')$, it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(at,o)).$$
- For every $o \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` *c*`_id FROM` *c* `JOIN` *subselect* `ON` *exp* `WHERE` exp'),
  - For every attribute $at = \langle ati,\ c,\ t\rangle$, such that $ati \in \mathrm{PropsInSel}(selitems)$ it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(at,o)).$$

Example 22. Consider the following SQL query:

```
SELECT email
FROM Lecturer
JOIN (SELECT lecturers
  FROM Enrollment
  WHERE lecturers = 'Huong') AS TEMP
ON Lecturer_id = TEMP.lecturers;
```

For any policy `SecVGU#`[`A`|`B`|`C`], for any scenario `VGU#`[`1`|`2`], only Huong is authorized to execute this query. Notice that this is so, even when this query is "equivalent" to the one in Example 21, which none of the lecturers are authorized to execute for any policy `SecVGU#`[`A`|`B`|`C`], and any scenarios `VGU#`[`1`|`2`].

Case q = `SELECT` *selItems* `FROM` *as* `JOIN` *subselect* `ON` *exp* `WHERE` exp'. Then, $\mathrm{AuthQuery}(\mathcal{S},\overline{\mathcal{O}},u,r,q)$ holds if and only if $\mathrm{AuthQuery}(\mathcal{S},\ \overline{\mathcal{O}},\ u,\ r,\ subselect)$ holds and

- if $ase_{\mathrm{l}} \in \mathrm{PropsInOn}(exp)$, with $\mathrm{CompWithInOn}(exp, ase_{\mathrm{l}}) = col$, but $ase_{\mathrm{r}} \notin \mathrm{PropsInOn}(exp)$, then:
  - For every $(o_{\mathrm{l}},\ o_{\mathrm{r}}) \in \mathrm{Exec}(\overline{\mathcal{O}}$, `SELECT` *col*, c_{r}`_id FROM` *subselect*, c_{r}), it holds that:
  $$\mathrm{Auth}(\mathcal{S},\mathcal{O},u,r,read(as,o_{\mathrm{l}},o_{\mathrm{r}})).$$
- Analogously for the case of $ase_{\mathrm{r}} \in \mathrm{PropsInOn}(exp)$, but $ase_{\mathrm{l}} \notin \mathrm{PropsInOn}(exp)$.

– if $ase_l \in \mathrm{PropsInOn}(exp)$ and $ase_r \in \mathrm{PropsInOn}(exp)$, then:

- For every $(o_l, o_r) \in$ Exec($\overline{\mathcal{O}}$, `SELECT` c_l`_id`, c_r`_id` `FROM` c_l, c_r), it holds that:

$$\mathrm{Auth}(\mathcal{S}, \mathcal{O}, u, r, read(as, o_l, o_r)).$$

Example 23. Consider the following SQL query:

```
SELECT TEMP.email
FROM Enrollment
JOIN (SELECT Lecturer_id, email
  FROM Lecturer
  WHERE Lecturer_id = 'Huong') AS TEMP
ON TEMP.Lecturer_id = lecturers;
```

For any policy `SecVGU#`[`A`|`B`|`C`], for any scenario `VGU#`[`1`|`2`], only Huong is authorized to execute this query. Notice that this is so, even when this query is "equivalent" to the one in Example 21, which none of the lecturers are authorized to execute for any policy `SecVGU#`[`A`|`B`|`C`], and any scenario `VGU#`[`1`|`2`].

Example 24. Consider the following SQL query:

```
SELECT TEMP.email
FROM Enrollment
JOIN (SELECT Lecturer_id, email
  FROM Lecturer
  WHERE Lecturer_id = 'Trang') AS TEMP
ON TEMP.Lecturer_id = lecturers;
```

For any policy `SecVGU#`[`A`|`B`|`C`], for any scenario `VGU#`[`1`|`2`], all lecturers are authorized to execute this query.

Case q = `SELECT` *selItems* `FROM` $subselect_1$ `JOIN` $subselect_2$ `ON` *exp* `WHERE` exp'. Then, $\mathrm{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, q)$ holds if and only if $\mathrm{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, subselect_1)$ and $\mathrm{AuthQuery}(\mathcal{S}, \overline{\mathcal{O}}, u, r, subselect_2)$ holds.

Case study We end this section by analyzing how our definition of AuthQuery() can prevent a malicious attacker from obtaining confidential information when executing the queries `Query#`[1|2|3], introduced in Figures 2–4. Concretely, in Table 3 we show the values of AuthQuery() for different combinations of the queries `Query#`[1|2|3], the users Huong, Manuel, and Hieu, the scenarios `VGU#`[1|2], and the security models `SecVGU#`[A|B|C]. Notice in particular that:

– Manuel is not authorized to execute `Query#2` in the scenarios `VGU#`[1|2], according to the security model `SecVGU#C`. This is to be expected, since in these scenarios Manuel and Huong are not *colleagues* with respect to Thanh, and, therefore, `SecVGU#C` does not authorize Manuel to see that Thanh is a student enrolled in Huong's courses.

– Similarly, Hieu is not authorized to execute `Query#2` in the scenario `VGU#1`, according to the security model `SecVGU#C`. This is to be expected, since in this scenario Hieu and Huong are not *colleagues* with respect to Thanh, and, therefore, `SecVGU#C` does not authorize Hieu to see that Thanh is a student enrolled in Huong's courses. However, in the scenario `VGU#2`, Hieu and Huong are *colleagues* with respect to Thanh, and, therefore, `SecVGU#C` does authorize Hieu to see that Thanh is a student enrolled in Huong's courses. Hence, as expected, Hieu is authorized to execute `Query#2` in this scenario according to `SecVGU#C`.

– Huong is not authorized to execute `Query#3` in the scenarios `VGU#`[1|2], according to the security models `SecVGU#C`. This is to be expected, since in these scenarios Huong and Manuel are not *colleagues* with respect to all the students enrolled in Manuel's courses, and, therefore, `SecVGU#C` does not authorize Huong to see them.

6. Related work

In this paper we have proposed a model-based characterization of fine-grained access control (FGAC) authorization for SQL queries. To the best of our knowledge, this seems to be the first attempt to propose such a characterization. In the past, (Cranor et al. 2002; Ashley et al. 2003) proposed the idea of specifying (privacy) policies using specific formalisms (e.g., P3P, EPAL, or XACML (Rissanen 2013)), and then translating these policies into security checks to be stored as meta-data in the databases. However, a formal definition of how these checks are generated from the policies, and how they interact with the execution of the queries is, to the best of our knowledge, still missing. More recently, (Mehta et al. 2017) addressed the problem in a different way. They propose a SQL-like language for writing the policies, and then an algorithm for automatically rewriting the queries by, essentially, adding the policies as where-clauses. Unfortunately, for this approach to work, the policies need to be (re-)written with the queries in mind. In fact, the policy language provides special constructors depending on whether the rules apply to queries that access only one column, more than one column, a user defined function, or an aggregate function. As a consequence, these policies can be hardly considered a *model*, and a formal discussion of their actual meaning —namely, what resources they protect, and what authorization decisions are to be inferred from them— is still missing.

Not related with authorization of SQL query execution, which is the focus of our paper, FGAC policies have been certainly studied before in the context of model-driven engineering. In particular, (Martínez et al. 2018) defined a language for specifying FGAC policies for models and metamodels, which also uses OCL for declaring "authorization constraints". (Bergmann et al. 2016; Debreceni et al. 2019) proposed a way to enforce FGAC policies, using bidirectional transformations, in order to control the access (read, modify) to the models in a collaborative modeling environment. Although focused on identifying policy violations on databases, and not to prevent them when executing queries, (Hamann et al. 2015) shares with our approach the use

		SecVGU#A		SecVGU#B		SecVGU#C	
caller	*query*	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		✗	✗	✗	✗	✗	✗

Table 3 The function AuthQuery(): lecturers attempting to execute case study's queries.

of modeling languages, and, in particular, of the OCL language, to express FGAC policies.

Notice that we left outside the scope of this paper the question of how we may propose to *enforce* FGAC policies when executing SQL queries. A quick review of the state of things with regards to FGAC access control in RDBMS, will shed light upon the current challenges, as well as set the stage for further discussing our future work. As it is well-known, *role-based* access control (RBAC) (Sandhu et al. 1996; Ahn & Sandhu 2000; Ferraiolo et al. 2001) is currently supported as a key security feature of database management systems (RDBMS). Nevertheless, RBAC is clearly insufficient for specifying FGAC. An approach often suggested for implementing FGAC in RDBMS that does not *natively* support FGAC (e.g., MySQL or MariaDB (Montee 2015)) consists of using *views*, in combination with the native RBAC support. This approach, however, is time-consuming, error-prone, and scales poorly. Moreover, the resulting implementations are hard to maintain, should any changes occur either in the database or in the FGAC policies. On the other hand, there are RDBMS that support FGAC —albeit at different degrees, and not in all versions— using an interesting variety of, more or less, "ad hoc" and proprietary mechanisms. In particular, Oracle supports FGAC through their Virtual Private Databases (VPD) (VPD-Oracle n.d.), IBM supports FGAC in DB2 through rows permission and column masking (DB2-IBM 2014), and PostgreSQL supports FGAC through row-level security (PostgreSQL n.d.). However, in these cases, the FGAC policies need to be manually implemented using each RDBMS's specific mechanism. Clearly, this task is time-consuming, error-prone, and scales poorly. Moreover, the resulting implementations are hard to maintain, should any changes occur either in the database or in the FGAC policies. The so-called Hippocratic Databases proposed in (Agrawal et al. 2002) to ensure privacy in IBM databases shows as well the limitations of the current solutions for implementing FGAC in RDBMS. Similar to the view approach, the idea is to create (meta-data) tables in the database for storing the policies. Then, (LeFevre et al. 2004) proposes two different algorithms to automatically rewrite a query in such a way that, when executing the query, the results are filtered out according to the policy. As in the case of the view approach, manually creating the (meta-data) tables representing the policies is time-consuming, error-prone, and scales poorly. To the best of our knowledge, Hippocratic Databases have not yet been realized.

7. Conclusions and future work

In this paper we have proposed a model-based characterization of fine-grained access control (FGAC) authorization for SQL queries. More specifically, we have defined a predicate AuthQuery() that represents whether a user is authorized by an FGAC-policy to execute a SQL query on a database. To illustrate our definition, we have provided examples of authorization decisions for different SQL queries, attempted by different users, in different scenarios, and with respect to different FGAC-policies. Currently, our definition does not cover the full SQL query language. In particular, we have left out outer joins, group-by clauses, and aggregation functions. We plan to extend our definition to cover these and other elements of the SQL query language. We also plan to extend our model-based approach to address fine-grained access control for other SQL statements, like inserts, updates, and deletes.

Having a formal characterization of FGAC-authorization for SQL queries is, however, only a *prerequisite*. The challenge now is to *enforce* the corresponding authorization decisions when executing SQL queries. [4] We have seen that the solutions provided by the major RDBMS are still far from ideal: in fact, they are time-consuming, error-prone, and scale poorly. In our proposal, FGAC-policies are modeled using SecureUML (Lodderstedt et al. 2002), and authorization constraints are specified using the Object Constraint Language (OCL) (OCL 2014). Interestingly, the availability of mappings from OCL to SQL (Nguyen & Clavel 2019) opens up the possibility of implementing AuthQuery() within the database and, consequently, of being able to enforce FGAC-policies following a model-driven approach.

References

Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. (2002). Hippocratic databases. In *Proceedings of the 28th international conference on very large data bases* (pp. 143–154). VLDB Endowment.

Ahn, G.-J., & Sandhu, R. (2000, November). Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, *3*(4), 207–226. Retrieved from https://doi.org/10.1145/382912.382913 doi: 10.1145/382912.382913

Ashley, P., Hada, S., Karjoth, G., Powers, C., & Schunter, M. (2003, 01). *Enterprise privacy authorization language (EPAL).* (https://www.w3.org/2003/p3p-ws/pp/ibm3.html)

Basin, D. A., Clavel, M., & Egea, M. (2011). A decade of model-driven security. In R. Breu, J. Crampton, & J. Lobo (Eds.), *16th ACM symposium on access control models and technologies, SACMAT 2011, innsbruck, austria, june 15-17, 2011, proceedings* (pp. 1–10). ACM. Retrieved from https://doi.org/10.1145/1998441.1998443 doi: 10.1145/1998441.1998443

Basin, D. A., Doser, J., & Lodderstedt, T. (2006). Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, *15*(1), 39–91. Retrieved from https://doi.org/10.1145/1125808.1125810 doi: 10.1145/1125808.1125810

Bergmann, G., Debreceni, C., Ráth, I., & Varró, D. (2016). Query-based access control for secure collaborative modeling using bidirectional transformations. In B. Baudry & B. Combemale (Eds.), *Proceedings of the ACM/IEEE 19th international conference on model driven engineering languages and systems, saint-malo, france, october 2-7, 2016* (pp. 351–361). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2976793

Cranor, L., Langheinrich, M., Marchiori, M., & Reagle, J. (2002, 01). *The platform for privacy preferences 1.0 (P3P1.0) specification.* (https://www.w3.org/TR/P3P/. Obsoleted 30 August 2018)

[4] Although the following opinion deserves a longer discussion, we certainly agree with (Kabra et al. 2006) about the importance of supporting FGAC at the database level: "Fine-grained access control [on databases] has traditionally been performed at the level of application programs. However, implementing security at the application level makes management of authorization quite difficult, in addition to presenting a large surface area for attackers —any breach of security at the application level exposes the entire database to damage, since every part of the application has complete access to the data belonging to every application user.".

Data Security Guide: Using Oracle Virtual Private Database to Control Data Access. (n.d.). (https://docs.oracle.com/database/121/DBSEG)

Debreceni, C., Bergmann, G., Ráth, I., & Varró, D. (2019). Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations. *Software and Systems Modeling*, *18*(3), 1737–1769. Retrieved from https://doi.org/10.1007/s10270-017-0631-8 doi: 10.1007/s10270-017-0631-8

Demuth, B., Hußmann, H., & Loecher, S. (2001). OCL as a specification language for business rules in database applications. In M. Gogolla & C. Kobryn (Eds.), *Uml* (Vol. 2185, p. 104-117). Springer.

Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., & Chandramouli, R. (2001, August). Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, *4*(3), 224–274. Retrieved from https://doi.org/10.1145/501978.501980 doi: 10.1145/501978.501980

Hamann, L., Sohr, K., & Gogolla, M. (2015). Monitoring database access constraints with an RBAC metamodel: A feasibility study. In F. Piessens, J. Caballero, & N. Bielova (Eds.), *Engineering secure software and systems - 7th international symposium, essos 2015, milan, italy, march 4-6, 2015. proceedings* (Vol. 8978, pp. 211–226). Springer. Retrieved from https://doi.org/10.1007/978-3-319-15618-7_16 doi: 10.1007/978-3-319-15618-7_16

ISO/IEC 9075-(1–10) Information technology – Database languages – SQL (Tech. Rep.). (2011). International Organization for Standardization. (http://www.iso.org/iso/)

Kabra, G., Ramamurthy, R., & Sudarshan, S. (2006). Redundancy and information leakage in fine-grained access control. In *Proceedings of the 2006 acm sigmod international conference on management of data* (pp. 133–144). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1142473.1142489 doi: 10.1145/1142473.1142489

LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., & DeWitt, D. (2004). Limiting disclosure in Hippocratic databases. In *Proceedings of the thirtieth international conference on very large data bases* (Vol. 30, pp. 108–119). VLDB Endowment.

Lodderstedt, T., Basin, D. A., & Doser, J. (2002). SecureUML: A UML-based modeling language for model-driven security. In J. Jézéquel, H. Hußmann, & S. Cook (Eds.), *UML 2002 - the unified modeling language, 5th international conference, dresden, germany, september 30 - october 4, 2002, proceedings* (Vol. 2460, pp. 426–441). Springer. Retrieved from https://doi.org/10.1007/3-540-45800-X_33 doi: 10.1007/3-540-45800-X_33

Martínez, S., Fouche, A., Gérard, S., & Cabot, J. (2018). Automatic generation of security compliant (virtual) model views. In J. Trujillo et al. (Eds.), *Conceptual modeling - 37th international conference, ER 2018, xi'an, china, october 22-25, 2018, proceedings* (Vol. 11157, pp. 109–117). Springer. Retrieved from https://doi.org/10.1007/978-3-030-00847-5_10 doi: 10.1007/978-3-030-00847-5_10

Mehta, A., Elnikety, E., Harvey, K., Garg, D., & Druschel,

P. (2017). Qapla: Policy compliance for database-backed systems. In *Proceedings of the 26th usenix conference on security symposium* (pp. 1463–1479). USA: USENIX Association.

Montee, G. (2015). *Row-level security in MariaDB 10: Protect your data.* (https://mariadb.com/resources/blog/)

Nguyen, H. P. B., & Clavel, M. (2019). OCL2PSQL: An OCL-to-SQL code-generator for model-driven engineering. In T. K. Dang, J. Küng, M. Takizawa, & S. H. Bui (Eds.), *Future data and security engineering - 6th international conference, FDSE 2019, proceedings* (Vol. 11814, pp. 185–203). Springer.

Object Constraint Language specification version 2.4 (Tech. Rep.). (2014, February). Object Management Group. (https://www.omg.org/spec/OCL/)

PostgreSQL 12.2. (n.d.). (Part II. SQL The Language. Chapter 5. Data Definition. 5.8. Row Security Policies. https://www.postgresql.org/docs/12/ddl.html)

Rissanen, E. (2013). *eXtensible access control markup language (XACML) version 3.0* (Tech. Rep.). OASIS. (http://docs.oasis-open.org/xacml/3.0/)

Row and column access control support in IBM DB2 for i (Tech. Rep.). (2014). International Business Machines Corporation. Retrieved from https://www.redbooks.ibm.com/redpapers/pdfs/redp5110.pdf/

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996, February). Role-based access control models. *Computer*, *29*(2), 38–47. Retrieved from https://doi.org/10.1109/2.485845 doi: 10.1109/2.485845