JOT
JOURNAL OF
OBJECT TECHNOLOGY

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 AuthQuery() that represents whether a user is authorized by an FGAC-policy to execute an 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, where authorization constraints are specified using the Object Constraint Language (OCL). To illustrate our definition of the predicate 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 AuthQuery() "within" the database and, consequently, of enforcing FGAC-policies in databases, following a model-based 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.

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. 1, 2020.
http://dx.doi.org/10.5381/jot.2020.19.1.e1

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).

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. We aim to fill this critical gap in this paper by providing a model-based characterization of FGAC authorization for a large class of SQL queries. Concretely, we define a predicate AuthQuery() that represents whether a user is authorized to execute an SQL query on a database, according to the FGAC-policy specified in a SecureUML model. To illustrate this definition, we provide

examples of authorization decisions for different SQL queries, attemped 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 and Clavel 2019)— and, consequently, of enforcing, following a model-based approach, FGAC-policies in SQL databases.[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 a FGAC-policy, modeled using SecureUML, whether a user is authorized to execute an 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 this section, we define our notions of data object models. In modeling access control policies, we use data models to specify the data to be protected. 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 (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, $as = \langle asi, ase_l, c_l, ase_r, c_r\rangle$, 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.

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`.*

[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".

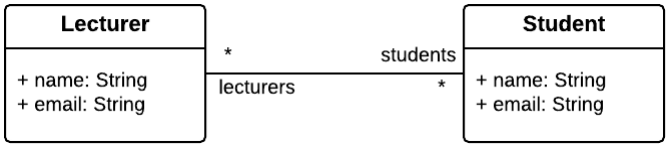

Figure 1 The `University` model

The end of the association `enrollment` at the class `Student` is `students`, while the end of the association at the class `Lecturer` is `lecturers`. 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 the students (denoted by `students`) and the lecturers (denoted by `lecturers`) of the courses the students have enrolled in.

Definition 2 (Object models) *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 threee 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 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 the 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 parametrized 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 $\mathcal{D}$. Now, 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 this section, we define our mappings from data models and object models to SQL. In characterizing access control authorization for SQL queries, we assume that SQL queries are executed on databases that implement the access control policies' underlying data models according to the mappings defined below.[3]

Definition 3 (The mapping of data models) *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* $\mathrm{SqlType}(t)$ = `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 $as \in AS$, $as = \langle asi, ase_1, c_1, ase_r, c_r\rangle \in AS$,*

```
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 (The mapping of object models) *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_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:*

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

Let $\langle asi, ase_1, c_1, ase_r, c_r\rangle$ be an assocation in AS, and let $\langle oi_1, c_1\rangle$ and $\langle oi_r, c_r\rangle$, be objects in OC. Then,

$$\mathrm{Eval}(\mathcal{O}, oi_1.ase_r) = \mathrm{Exec}(\overline{\mathcal{O}}, SELECT\ ase_r\ FROM\ asi\ WHERE\ ase_1 =$$

and

$$\mathrm{Eval}(\mathcal{O}, oi_r.ase_1) = \mathrm{Exec}(\overline{\mathcal{O}}, SELECT\ ase_1\ FROM\ asi\ WHERE\ ase_r =$$

[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 for each so-called SecureUML dialect. Next we define the actions that we consider in our SecureUML dialect:

Definition 5 (Read-actions) *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 (Instance read-actions) *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 (preceded by `$`)

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 (Security models) *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 $\mathrm{auth} : R \times \mathrm{Act(D)} \longrightarrow \mathrm{Exp}(\mathcal{D})$ 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 its students. Formally,*

$$\mathrm{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{enrollment})) = \$lecturers = \$caller.$$

  - *Any lecture can know his/her own email, as well as the emails of his/her students. Formally,*

$$\mathrm{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{email})) = (\$caller = \$self) \; or \; (\$caller.students -> includes(\$self))$$

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

- *Permissions:*
  - *Any lecturer can know its 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,*

$$\mathrm{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{email})) = (\$caller = \$self) \; or \; (\$caller.students -> includes(\$self)) \; or \; (\$caller.students -> exists(s|s.lecturers -> includes(\$self))).$$

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

- *Permissions:*
  - *Any lecturer can know its* colleagues*. Formally,*

$$\mathrm{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{enrollment})) = \$lecturers = \$caller \; or \; \$caller.students -> includes(\$students).$$

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 given read-action on a given scenario.

Definition 8 (The predicate Auth()**)** *Let $\mathcal{D}$ be a data model. Let $\mathcal{S} = \langle R, \text{auth} \rangle$ 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* $\text{read}(at, o) \in \text{Act}(\mathcal{O})$,

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, \text{read}(at, o)) \iff \text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(at))[\texttt{\$self} \mapsto o; \texttt{\$caller} \mapsto u]).$$

– *For any action* $\text{read}(as, o_l, o_r) \in \text{Act}(\mathcal{O})$,

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, \text{read}(as, o_l, o_r)) \iff \text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(as))[\$as_l \mapsto o_l; \$as_r \mapsto o_r; \texttt{\$caller} \mapsto u].$$

To illustrate the definition of the predicate Auth(), we show in Tables 1–4 the different values of this predicate, for the same actions, but with different users (*callers*), on different scenarios, and for different security models. The scenarios that we consider are `VGU#1` (Example 2) and `VGU#2` (Example 3). The security models that we consider are `SecVGU#A` (Example 4), `SecVGU#B` (Example 5), and `SecVGU#C` (Example 6).

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 AuthQuery() that checks, given a FGAC-policy modelled using our SecureUML dialect, whether a user is authorized to execute an SQL query on a database. To illustrate the definition of the predicate 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. As before, the scenarios that we consider are `VGU#1` (Example 2) and `VGU#2` (Example 3). The security models that we consider are `SecVGU#A` (Example 4), `SecVGU#B` (Example 5), and `SecVGU#C` (Example 6).

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*.
– `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′*.

```
SELECT email FROM lecturer WHERE lecturer_id = 'Huong'
```

Figure 2 Example. `Query#1`.

```
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`.

Attacker model Our definition of the predicate AuthQuery() aims to prevent a malicious attacker from obtaining unauthorized data by executing SQL queries in the database. We assume that the attacker (a) knows the database schema; b) understands SQL; and c) can execute arbitrary SQL queries in the database. On the other hand, we assume that the attacker does not know the security policy. Notice that the attacker's capacity (b) is critical. To illustrate this point, consider the queries `Query#`[1|2|3] in Figures 2–4. Suppose that, for a given scenario, all of them return the same result: namely, a non-empty email. The problem is that each query implicitly reveals additional information, which may be unauthorized. In particular, `Query#1` reveals that the email belongs to Huong. Then, `Query#2` reveals not only that the email belongs to Huong, but also that Thanh is enrolled in a course that Huong is teaching. Finally, `Query#3` reveals that the email belongs to Huong, and that Huong and Manuel are colleagues, i.e., that there is at least one student who is enrolled in a course that Huong is teaching, and also in a course that Manuel is teaching. At the end of this section, we will revisit this basic case study, to analyze if our definition of AuthQuery() prevents a malicious attacker to obtain confidential information by executing the queries `Query#`[1|2|3].

Preliminaries In the definition of our predicate AuthQuery(), we use the following auxiliary functions.

– PropsInSel(*selitems*): the set of *properties* (i.e., attributes and association-ends) that appear in a list of selected items.

```
SELECT DISTINCT email FROM lecturer
JOIN (SELECT e1.lecturers as lecturers
      FROM (SELECT * FROM enrollment WHERE lecturers = '
            ) AS e1
      JOIN (SELECT * FROM enrollment WHERE lecturers = '
            ) AS e2
      ON e1.students = e2.students
      ) AS TEMP
ON TEMP.lecturers = lecturer_id;
```

Figure 4 Example. `Query#3`.

		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.

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

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

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

Table 3 The predicate Auth(): Huong attempting to read lecturers' enrolled students.

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

Table 4 The predicate Auth(): Hieu attempting to read lecturers' enrolled students.

– PropsInWhe(exp): the set of *properties* (i.e., attributes and association-ends) that appear in a where-expression.
– PropsInOn(exp): the set of *properties* (i.e., attributes and association-ends) that appear in an on-expression.
– 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 AuthQuery() as follows:

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

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

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

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

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

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).*

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* = $\langle asi, ase_l, c_l, ase_r, c_r \rangle$. Then, AuthQuery($\mathcal{S}, \overline{\mathcal{O}}, u, r, q$) holds if and only if:

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

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

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 is authorized to execute this query. However, for any scenario with no students, all the lectures 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 is authorized to execute this query. However, for any scenario with no students, all the lectures 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 = stude
```

For any policy `SecVGU#`[A|B|C], *for any scenario* `VGU#`[1|2], *none of the lecturers is authorized to execute this query. Notice that this is so, even when, 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, AuthQuery($\mathcal{S}$, $\overline{\mathcal{O}}$, u, r, q) holds if and only if 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 any policy SecVGU#[A|B|C]*, for any scenario* VGU#[1|2]*, none of the lecturers is authorized to execute this query. Notice that this is so, even when this query is "equivalent" to the one in Example 7, which all lectures 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 $c_{[l|r]}$ JOIN *as* ON *exp* WHERE *exp*′, where *as* = $\langle asi, ase_l, c_l, ase_r, c_r \rangle$. Then, AuthQuery($\mathcal{S}, \overline{\mathcal{O}}, u, r, q$) holds if and only if:

- For every $o_{[l|r]} \in$ Exec($\overline{\mathcal{O}}$, SELECT $c_{[l|r]}$_id FROM $c_{[l|r]}$),
  - For every attribute $at = \langle ati, c_{[l|r]}, t \rangle$, such that $ati \in$ PropsInOn(*exp*), it holds that:

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, \texttt{read}(at, o_{[l|r]})).$$

- 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:

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

- For every $o_{[l|r]} \in$ Exec($\overline{\mathcal{O}}$, SELECT $c_{[l|r]}$_id FROM $c_{[l|r]}$ JOIN *as* ON *exp*),
  - For every attribute $at = \langle ati, c_{[l|r]}, t \rangle$, such that $ati \in$ PropsInWhe(*exp*′), it holds that:

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, \texttt{read}(at, o_{[l|r]})).$$

- For every $o_{[l|r]} \in$ Exec($\overline{\mathcal{O}}$, SELECT $c_{[l|r]}$_*id* FROM $c_{[l|r]}$ JOIN *as* ON *exp* WHERE *exp*′),
  - For every attribute $ati = \langle ati, c_{[l|r]}, t \rangle$, with $ati \in$ PropsInSel(*selitems*) it holds that:

$$\text{Auth}(\mathcal{S}, \mathcal{O}, u, r, \texttt{read}(at, o_{[l|r]})).$$

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 is 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 is 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 is authorized to execute this query.*

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

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

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

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

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

- For every $o \in$ 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 $ati \in$ PropsInSel(*selitems*) it holds that:

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

Example 22 *Consider the following SQL query:*

```
SELECT email FROM lecturer
  JOIN (SELECT lecturers FROM enrollment WHERE lecturer_id = 'Trang') AS TEMP
  ON lecturer_id = TEMP.lecturers.
```

For any policy `SecVGU#[A|B|C]`*, for any scenario* `VGU#[1|2]`*, Huong is authorized to execute this query. For policy* `SecVGU#[C]`*, for any scenario* `VGU#[1|2]`*, Manuel is also authorized to execute this query. For policy* `SecVGU#C` *and scenario* `VGU#2`*, Hieu is also 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 lectures is 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, AuthQuery$(\mathcal{S}, \overline{\mathcal{O}}, u, r, q)$ holds if and only if AuthQuery$(\mathcal{S}, \overline{\mathcal{O}}, u, r,$ *subselect*$)$ holds and

- if $ase_l \in$ PropsInOn(exp), with CompWithInOn$(exp, ase_l) = col$, but $ase_r \notin$ PropsInOn(exp), then:
  - For every $(o_l, o_r) \in$ Exec$(\overline{\mathcal{O}},$ `SELECT` *col*, c_r`_id` `FROM` *subselect*, $c_r)$, it holds that:

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

- Analogously, if $ase_r \in$ PropsInOn(exp), but $ase_l \notin$ PropsInOn(exp).
- if $ase_l \in$ PropsInOn(exp) and $ase_r \in$ 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:

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

- Similarly, if $ase_l \notin$ PropsInOn(exp) and $ase_r \notin$ PropsInOn(exp).

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]`*, Huong is authorized to execute this query. For policy* `SecVGU#[C]`*, for any scenario* `VGU#[1|2]`*, Manuel is also authorized to execute this query. For policy* `SecVGU#C` *and scenario* `VGU#2`*, Hieu is also 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 lectures is 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 = lecturer.
```

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

Case q = `SELECT` *selItems* `FROM` *subselect*$_1$ `JOIN` *subselect*$_2$ `ON` *exp* `WHERE` *exp*$'$. Then, AuthQuery$(\mathcal{S}, \overline{\mathcal{O}}, u, r, q)$ holds if and only if AuthQuery$(\mathcal{S}, \overline{\mathcal{O}}, u, r,$ *subselect*$_1)$ and AuthQuery$(\mathcal{S}, \overline{\mathcal{O}}, u, r,$ *subselect*$_2)$ holds.

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 Figure 5 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, our main goal was to propose a model-based characterization of fine-grained access control 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

		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 5 The function AuthQuery(): lectures attempting to execute case study's queries.

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) addresses the problem in a different way. It proposes an 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 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 decision are to be inferred from them— is still missing.

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) has been comprehensively defined (Sandhu et al. 1996), extended (Ahn and Sandhu 2000), standardized (Ferraiolo et al. 2001), and 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

Lecturer_id	Email_Choice
Hieu	
Huong	huong@vgu.edu.vn
Manuel	

Table 6 LecturerEmail_Huong View

policies. To illustrate this challenge, consider the problem of implementing the policy SecVGU#A with respect to emails, in the context of scenarios VGU#[1|2]. Following this view approach, one needs first to manually create, for each lecturer, a view representing the emails that he/she is authorized to read. Then, one needs to manually rewrite each query that attempts to access emails in such a way that the appropriate view is accessed instead. As an example, Table 6 shows the View LecturerEmail_Huong containing the emails that Huong can access. Then, when Huong attempts to execute the query:

```
SELECT email FROM lecturer WHERE lecturer_id = 'Huong',
```

it will be rewritten as:

```
SELECT email FROM LecturerEmail_Huong.
```

On the other hand, there are also 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 nd), IBM supports FGAC in DB2 through rows permission and column masking (DB2-IBM 2014), and PostgreSQL supports FGAC through row-level security (PostgreSQL nd). 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. To illustrate this challenge, consider the problem of implementing in Oracle VPD the policy `SecVGU#A` with respect to emails. Following VPD approach, one needs to manually rewrite each query by adding an appropriate where-clause such that, when executing the query, the results are filtered out according to the policy. As an example, the query

```
SELECT email FROM lecturer WHERE lecturer_id = 'Huong'
```

will be rewritten as:

```
SELECT email FROM lecturer WHERE lecturer_id = 'Huong'
  AND lecturer_id = SYS_CONTEXT(''USER_ID''),
```

where `USER_ID` is obtained from the application context. Not surprisingly, (VPD-Oracle nd) recommends against using this approach for complex queries, since the results may be unexpected, in particular for queries involving outer joins. Notice that the approach proposed in (Mehta et al. 2017), briefly discussed above, may produce similar "unexpected" results.

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. The first algorithm replaces the select-items that are protected by the policy by appropriate case-statements. As an example, consider the policy `SecVGU#A` with respect to emails. One needs first to manually create, for each lecturer, a table representing (in a specific way) the emails that he/she is authorized to read. For example, Table 7 shows the table representing the emails that Huong can access. Then, when Huong attempts to execute the query

```
SELECT email FROM lecturer WHERE lecturer_id = 'Huong'
```

it will be automatically rewritten as:

```
SELECT CASE WHEN
  EXISTS (SELECT email_Choice FROM lecturer_Huong
  WHERE lecturer.lecturer_id = lecturer_Huong.lecturer_id
  AND email_Choice = 1)
THEN email ELSE NULL END AS email
FROM lecturer
WHERE lecturer_id = 'Huong';
```

The second algorithm replaces the from-tables that are protected by the policy by appropriate left-joins. For example, when Huong attempts to execute

```
SELECT email FROM lecturer WHERE lecturer_id = 'Huong'
```

the query will be automatically rewritten as:

lecturer_id	ID_Choice	email_Choice
Hieu	1	0
Huong	1	1
Manuel	1	0

Table 7 `Lecuter_Huong` Table

```
SELECT email FROM
  (SELECT lecturer_id FROM lecturer) AS TEMP1
LEFT JOIN
(SELECT lecturer_id, email FROM lecturer WHERE
   EXISTS (SELECT email_Choice FROM lecturer_Huong
   WHERE lecturer.lecturer_id = lecturer_Huong.lecturer
  AND email_Choice = 1)
) AS TEMP2
ON TEMP1.lecturer_id = TEMP2.lecturer_id
WHERE TEMP1.lecturer_id = 'Huong';
```

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. Moreover, both algorithms "inject" null-values (representing "unauthorized access") in the result sets, which may raise unexpected results. In any event, 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 an 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, following the same principles underlying our current definition. We also plan to extend our model-based approach to address fine-grained access control for other SQL statements, like inserts, updates, and deletes.

Nevertheless, having a formal characterization of fine-grained access control (FGAC) authorization for SQL queries is, after all, a *prerequisite*. The challenge now is to *enforce* the corresponding authorization decisions when executing SQL

queries. [4] We have argued 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), where authorization constraints are specified using the Object Constraint Language (OCL) (OCL 2014). Interestingly, the availability of mappings from OCL to SQL (Nguyen and Clavel 2019) opens up the possibility of implementing AuthQuery() in the database and, consequently, of enforcing FGAC-policies following a model-based approach: namely, by automatically rewriting a non-secure SQL query into a secure query, with respect to a given SecureUML model, following the definition of the predicate AuthQuery().

References

Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2002). Hippocratic databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 143–154. VLDB Endowment.

Ahn, G.-J. and Sandhu, R. (2000). Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226.

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

Basin, D. A., Clavel, M., and Egea, M. (2011). A decade of model-driven security. In Breu, R., Crampton, J., and Lobo, J., editors, *16th ACM Symposium on Access Control Models and Technologies, SACMAT 2011, Innsbruck, Austria, June 15-17, 2011, Proceedings*, pages 1–10. ACM.

Basin, D. A., Doser, J., and Lodderstedt, T. (2006). Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91.

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

DB2-IBM (2014). Row and column access control support in IBM DB2 for i. Technical report, International Business Machines Corporation.

Demuth, B., Hußmann, H., and Loecher, S. (2001). OCL as a specification language for business rules in database applications. In Gogolla, M. and Kobryn, C., editors, *UML*, volume 2185 of *LNCS*, pages 104–117. Springer.

Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R. (2001). Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274.

[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.".

Kabra, G., Ramamurthy, R., and 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*, SIGMOD '06, pages 133–144, New York, NY, USA. Association for Computing Machinery.

LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., and DeWitt, D. (2004). Limiting disclosure in Hippocratic databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, volume 30 of *VLDB '04*, pages 108–119. VLDB Endowment.

Lodderstedt, T., Basin, D. A., and Doser, J. (2002). SecureUML: A UML-based modeling language for model-driven security. In Jézéquel, J., Hußmann, H., and Cook, S., editors, *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer.

Mehta, A., Elnikety, E., Harvey, K., Garg, D., and Druschel, P. (2017). Qapla: Policy compliance for database-backed systems. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC '17, pages 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. and Clavel, M. (2019). OCL2PSQL: An OCL-to-SQL code-generator for model-driven engineering. In Dang, T. K., Küng, J., Takizawa, M., and Bui, S. H., editors, *Future Data and Security Engineering - 6th International Conference, FDSE 2019, Proceedings*, volume 11814 of *Lecture Notes in Computer Science*, pages 185–203. Springer.

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

PostgreSQL (n.d.). PostgreSQL 12.2. 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. Technical report, OASIS. http://docs.oasis-open.org/xacml/3.0/.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *Computer*, 29(2):38–47.

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

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

About the authors

Hoàng Nguyễn Phước Bảo is post-baccalaureate researcher at the Vietnamese-German Univeristy (Vietnam). You can contact him at ngpbhoang1406@gmail.com.

Manuel Clavel is professor in Software Engineering and Programming Languages at the Vietnamese-German Univeristy (Vietnam). You can contact him at manuel.clavel@vgu.edu.vn.