Efficient Enforcement of Action-Aware Purpose-Based Access Control within Relational Database Management Systems

Article *in* IEEE Transactions on Knowledge and Data Engineering · March 2015

DOI: 10.1109/TKDE.2015.2411595

CITATIONS
12

READS
252

2 authors:

Pietro Colombo
Università degli Studi dell'Insubria
48 PUBLICATIONS **724** CITATIONS
SEE PROFILE

Elena Ferrari
Università degli Studi dell'Insubria
325 PUBLICATIONS **8,149** CITATIONS
SEE PROFILE

Efficient Enforcement of Action-aware Purpose-based Access Control within Relational Database Management Systems

Pietro Colombo Elena Ferrari
Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria, Varese, Italy
Email: {pietro.colombo,elena.ferrari}@uninsubria.it

Abstract—Among the variety of access control models proposed for database management systems (DBMSs) a key role is covered by the purpose-based access control model, which, while enforcing access control, also achieves basic privacy preservation. We believe that DBMSs could greatly take benefit from the integration of an enhanced purpose based model supporting highly customized and efficient access control. Therefore, in this paper we propose a purpose-based model that supports action-aware policy specification and a related efficient enforcement framework to be integrated into relational DBMSs. The experimental evaluation we have performed shows the feasibility and efficiency of the proposed framework.

1 Introduction

With the advent of recent innovations in fields related to mobile computing and digital networks, the volume of personal and sensitive data which are stored and processed is rapidly growing. In this scenario, a key role is played by database management systems (DBMSs), which store data and provide tools to access and analyze them. Although data protection via access access control is becoming a key requirement for DBMSs, at present, most of the commercial DBMSs natively integrate quite basic form of access control.

Several access control models have been proposed in the literature [1], such as the discretionary, mandatory, role-based, and purpose-based model, operating at granularity that ranges from the level of tables to the one of cells. Among these models, the purpose-based model covers a key role, as it helps bridging the gap between security and privacy oriented data protection mechanisms. Indeed, from a privacy oriented perspective, purposes represent the essence of privacy policies, whereas, from a security perspective, a new complementary dimension of access control. As a matter of fact, probably inspired by the seminal work by Agrawal et al. [2], in the recent years several purpose based access control models (e.g., [3], [4], [5], [6], [7]) have been proposed (see Section 7 for a detailed discussion). Some of these proposals enhanced the core purpose-based model with additional features to increase the efficacy of the control. For instance, in [3] and [5] purpose based access control is combined with role based mechanisms. We believe that the support for more expressive policies could potentially lead to define highly customized forms of access control, and relational DBMSs could greatly take benefit from the integration of models with these capabilities. However, in order to make achievable the integration, efficient enforcement techniques are needed.

Based on these considerations, in this paper we propose an action-aware purpose-based access control model for relational DBMSs, namely, a purpose-based model which enforces fine grained access control on the basis of: 1) the purposes of the access, 2) the actions executed by SQL queries on the accessed data, and 3) the categories of the accessed data. For instance, given a table *Employees(name, role, salary)*, let us consider the queries q_a: *select name, salary from Employees* and q_b: *select count(name), avg(salary) from Employees*. These two queries disclose different information related to the stored data. Indeed, the actual content of the fields *name* and *salary* of *Employees*' tuples is only shown by the result set of q_a. As such, q_a shows an higher threatening level than q_b. Proper policies should therefore be defined to regulate the execution of queries based on the actions (e.g., combinations, aggregations, filtering) that are executed on data. Moreover, data stored into different table columns can belong to data categories characterized by different sensitivity levels. For instance, considering once again table *Employees*, column *name* identifies an employee, *role* provides a public information related to the employee's function in the company, whereas *salary* is a private piece of information related to the employee contract. The threatening level of queries depends on the categories of the accessed data. For instance, let us consider again query q_a and an additional query q_c: *select name, role from Employees*. Since q_a relates private information to individuals (i.e., *salary* to *name*), it is more threatening than q_c, which relates public information to individuals (i.e., *role* to *name*). As such, we believe that policies should regulate the access based also on data categories. However, we are not aware of access control models for DBMSs with all the above mentioned capabilities. For instance, the model proposed by Byun and Li [3], which can be considered the reference purpose-based model for relational DBMSs, regulates the access based on

purpose compliance. The access is granted if the purposes for which the accessed data have been collected comply with the purposes for which the queries access the data. Besides supporting any purpose based policy expressible with [3], the model proposed in this paper supports action aware policies, allowing far higher levels of access control customization.

The proposed access control model has been implemented by a framework which allows integrating policy specification and enforcement capabilities into relational DBMSs. The framework is defined to minimize the memory consumption for policy specification and the time enforcement overhead. Experimental evaluations show the efficiency of the proposed solution.

The rest of the paper is organized as follows. Section 2 provides an overview of the framework. Section 3 presents the running example used throughout the paper. Section 4 introduces the conceptual elements characterizing our access control model. Section 5 discusses selected aspects of the framework, whereas Section 6 presents experimental results. Section 7 surveys related work. Section 8 concludes the paper. Finally, the paper includes 2 appendixes: Appendix A provides details related to the enforcement mechanisms, whereas Appendix B presents formal correctness proofs.

2 Overview

The framework implementing the action aware purpose-based access control model proposed in this paper includes several modules, which are graphically described in Figure 1. In this paper, we mainly focus on functionalities of the Access Control Management module, the Policy Management module, and the Enforcement Monitor.

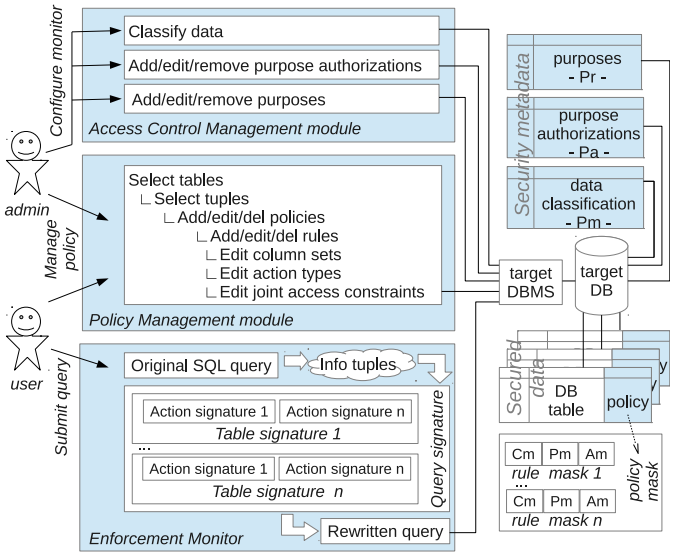


Figure 1. Framework overview

The Access Control Management module is used for: 1) defining the set of purposes involved in policy specification and enforcement; 2) specifying purpose-based authorizations; 3) classifying data stored into the target database into data categories used for access control purposes. These activities allow security administrators to configure access control for the target DBMS. All specified security related meta-data are stored into tables of the protected database.

The Policy Management module is used for fine-grained access control policy management. It can be used to serve users/administrators policy specification requests (e.g., add a policy), as well as to automatically handle updates to the specified policies as a consequence of modifications to the set of purposes or to the scheme of database tables.

Finally, the Enforcement Monitor enforces access control by means of SQL query rewriting, issuing the rewritten queries to the secured DBMS.

Details related to conceptual elements involved in the definition of these modules, and to the implemented specification and enforcement mechanisms, are presented in Sections 4 and 5.

3 Running Example

In order to ease the presentation of the needed concepts, in this section, we introduce a running example that we will use in the rest of the paper. We consider a scenario where a relational database stores information of patients hospitalized in a nursing home. Patients wear "smart watches" embedding sensors that sense position, movements, temperature and heart beats. The watches send the observed data to the DBMS that manages the *patients* database (db), which handles the sensed data along with contact information and nutritional information related to the patients.

patients includes the tables *users(user_id, watch_id, nutritional_profile_id)*, *sensed_data(watch_id, timestamp, temperature, position, beats)* and *nutritional_profile(profile_id, food_intolerances, food_preferences, diet_type)*.

sensed_data stores all data issued by smart watches and the time at which those data have been sensed, *nutritional_profile* stores information related to patients nutrition. Finally, table *users* keeps track of the registered patients.

4 The Domain Model

The application domain, which is considered in this work, is characterized by the following basic elements:

- Data, which are stored into a relational DBMS and classified into data categories.
- Queries, which access and process data.
- Action-aware purpose-based data policies (for the sake of brevity, referred to as policies in the remaining of the paper), which regulate the execution of queries.

In the rest of this section, after introducing the considered *data categories*, we formalize the concepts of *policy* and *query signature*, i.e., a model that specifies the tables and columns accessed by a query, the action types and the purpose for which the query accesses data.

4.1 Data categories

Data categories are mentioned by privacy legislations, such as the European Data Protection Directive,[1] which refers

1. Directive 95/46/EC of the European Parliament and of the Council of 24/10/1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data.

to sensitive, personal and identifiable data, as well as by privacy preserving data publishing work. For the purposes of this work, we consider the categories *identifier*, *quasi identifier*, *sensitive*, and *generic*. *Identifiers* data are personal data allowing a data subject to be directly identified.[2] A set of data is said to be *quasi identifier* if the joint access to all the elements in the set together with external data can allow the identification of the individual to which these data refer to [8]. Data are called *sensitive* when collect information that, if linked to an individual, reveals sensitive aspects of his/her private life. This class includes data such as medical, educational, financial, religious creed, political preferences, and employment information.[2] Finally, the *generic* category groups data that do not belong to any of the other categories.

Security administrators are charged with data categorization (see Section 2). The results of the data categorization for our running example is shown in Figure 2.

Table	Column	Category
users	user_id	identifier
users	watch_id	quasi identifier
users	nutritional_profile_id	quasi identifier
sensed_data	watch_id	quasi identifier
sensed_data	timestamp	generic
sensed_data	temperature	sensitive
sensed_data	position	sensitive
sensed_data	beats	sensitive
nutritional_profile	profile_id	quasi identifier
nutritional_profile	food_intolerances	sensitive
nutritional_profile	food_preferences	sensitive
nutritional_profile	diet_type	sensitive

Figure 2. Patients Data Categories

To the best of our knowledge, the proposed categories are the only ones referred to by privacy regulations. We believe that they could be sufficient to allow policy customization in a variety of application domains. However, the proposed list is not necessarily complete and administrators can add other categories with small extensions to the mechanisms introduced in Section 5. If useless for the purposes of a given application scenario, data categorization can be skipped. In this case the framework implicitly classifies all data of the target database as *generic*.

4.2 Policies

Policies specify the purposes for which data can be processed, the type of actions that can be executed on data, and the categories of the data that can be jointly accessed with those to which the policies are assigned. Policies are fine grained in that they can refer to single data items contained in each tuple.

Since policies are complex elements, we first introduce their basic constituent elements.

Let us start considering the concept of *action type*. Different forms of accesses are performed by SQL queries, which should be regulated by policies. We characterize such accesses according to four different dimensions.

The first dimension concerns the *indirection* of the access. Granting the *indirect* access, a query can access the data for which the policy has been specified only for filtering, grouping and ordering those data that will compose the result set of the query. In contrast, granting the *direct* access, a query is allowed to access the data on which the policy applies, and use them to derive the value of those data that compose the query result set. Intuitively, policies regulating the direct access should be more restrictive than those regulating indirect access.

Example 1 Suppose that Bob, a patient of the nursing room, specifies a policy that allows the *indirect* access to attribute *diet_type* of *nptp*, the *nutritional_profile* tuple that refers to his nutritional information. A query such as q1: *select food_intolerances from nutritional_profile where diet_type like 'vegan'*, complies with the policy. In contrast, queries performing a direct access, such as q2: *select * from nutritional_profile* are not authorized to access the *diet_type* field of *nptp*.

The second dimension concerns the *multiplicity* of the data sources accessed to derive the result set. This dimension allows refining policies that grant direct access. The multiplicity specifies whether a query accessing the field to which the policy is attached can show[3] the content of such a field, or it has to combine such value with those of data stored into other table columns. In the former case, the multiplicity is set to *single*, as the corresponding value derived by the query has been extracted from a single data field. In the latter case, it is set to *multiple*, as the derived value has been defined combining data belonging to multiple data fields.

Example 2 Suppose that Bob specifies a policy for his *sensed_data* tuples, which grants the direct access to the field *temperature* only when it is combined with other data sources, i.e., when queries execute a direct access from multiple sources. This policy allows queries such as *select temperature-avg(temperature), timestamp from users join sensed_data on users.watch_id = sensed_data.watch_id where user_id like 'Bob'* to access *temperature*, as this query derives the variation from the average temperature at a given observation time.

The third dimension concerns the *aggregation*. This dimension further specializes the policies in case of direct access to single or multiple sources, specifying whether a query can show the content of the constrained data fields or it has to combine them, through aggregation, with the homonymous field of other tuples. When this dimension is set to *aggregation*, the access to the referred field F of a tuple tp is only allowed if the accessing query aggregates the value of F with those of the F field of other tuples. If the component is set to *no aggregation*, the accessing queries can show the value of the corresponding field.

Example 3 Suppose that Bob specifies a policy for all his *sensed_data* tuples which allows the direct access with aggregation to column *temperature*. This policy allows the execution of queries such as: *select avg(temperature) from sensed_data s join users u on s.watch_id=u.watch_id where u.user_id like 'Bob'*. Indeed, this query aggregates the values of field *temperature* of multiple tuples.

2. Data Protection Code, Legislative Decree no. 196 of 30 June 2003 of the Italian legislation.

3. By "showing" we mean reporting the content of the field, or deriving values resulting from the application of scalar functions.

The forth (and last) dimension allows one to specify with which data category among *identifier*, *quasi identifier*, *sensitive*, and *generic*, the constrained data fields can be jointly accessed.

For example, if a data owner allows the joint access to generic and sensitive data, based on Table 2, attributes *timestamp*, and *temperature* can be accessed by the same query. Joint access to data belonging to different categories can reveal private/sensitive information related to individuals. This is the case, for instance, of joint access to identifier and sensitive data. The selectivity of policies should be defined accordingly.

The grouping of the aforementioned four dimensions contribute to the definition of an *action type*.

Definition 1 (Action type)**.** *An action type Ac is a tuple ⟨Ia, Ms, Ag, Ja⟩, where Ia:{d, i} specifies the indirection of the access (d=direct, i=indirect), Ms:{s, m} refers to the multiplicity of the data sources (s=single, m=multiple), Ag:{a, n} indicates the data aggregation criteria (a=aggregation, n=no aggregation), and Ja:⟨ i:{a, n}, q:{a, n}, s:{a, n}, g:{a, n}⟩ specifies the joint access constraint.*[4]

The joint access constraint to be included in an action type specifies the categories of data that can be jointly accessed with the table attributes for which the policy is specified. As such, the specification requires to list the categories for which the joint access is allowed and those for which it is not. More precisely, let us suppose to define a joint access constraint for a set of attributes of a table T referred to as A_T . If the joint access constraint specifies *i* as an allowed category (i.e., *Ja.i=a*), the attributes in A_T can be jointly accessed with any other table attributes (also of different tables) that have been classified as identifiers.

As previously mentioned, policies specify the *purposes* for which data collected in the system can be accessed by queries. The collection of the purposes defined for an application scenario forms the scenario's *purpose set*, referred to as *Ps*. The purpose set of our running example includes: *treatment* (p1), *payment* (p2), *healthcare-operations* (p3), *law-enforcement* (p4), *reporting* (p5), *research* (p6), *marketing* (p7), and *sale* (p8).

The composition of the aforementioned components specifies a policy rule. A rule specifies the purposes for which specific types of actions can be executed on given columns of a table. A policy is then defined by grouping different rules.

Definition 2 (Data policy)**.** *A policy PP is a tuple ⟨Rs, Tb, tp⟩, where Rs is a set of policy rules, Tb is the table to which PP is applied, whereas tp either refers to a specific tuple of Tb or it is set to ⊥, meaning that PP is applied to all Tb tuples. A rule R (collected in Rs) is in turn a tuple ⟨Cl, Pu, At⟩, where Cl is the set of columns of Tb to which R is applied, At is an action type, whereas Pu is the set of purposes for which actions belonging to At are authorized.*

4. Ja components have been labelled with the initial of the data category to which they refer to, (e.g., *i* refers to the access to data belonging to the *identifier* class). Such elements are set to *a* to specify that an access is allowed, whereas they are set to *n* to specify that it is not allowed

Example 4 Suppose that Bob specifies a policy that regulates the access to his *sensed_data*. The specified policy includes several rules, among which, *r1*= ⟨{*temperature, position, beats*}, {*p1,p2,p3,p4,p5,p6*}, ⟨*i,m,n,*⟨*n,n,a,n*⟩⟩⟩, specifies the purposes for which the attributes *temperature*, *beats*, and *position* can be accessed with indirect access; whereas *r2*= ⟨{*temperature, beats*}, {*p1,p3,p4,p6*}, ⟨*d,s,n,*⟨*n,n,a,n*⟩⟩⟩ specifies the allowed access purposes in case of direct access from single source with aggregation. Both *r1* and *r2* only allow the joint access with data in the *sensitive* category.

4.3 Query signatures

A *query signature* is a model describing the types of actions that are performed by a query on the accessed data. Query signatures are introduced to allow verifying the compliance of the actions performed by queries with the policies specified for the accessed data. Similar to policies, before formalizing the concept of *query signature*, we introduce the basic elements involved in its specification, referred to as *action* and *table signatures*.

An *action signature* is a model of an action performed by a query on given columns of a table. Multiple action signatures can be defined for the same query.

Definition 3 (Action signature)**.** *Let Q be a query and T one of the tables accessed by Q. An action signature As for Q is a tuple ⟨Cs, Ac⟩, where Cs is a subset of A_T attributes,*[5] *and Ac specifies the action type (see Def. 1) that is performed by Q on the attributes in Cs.*

It is important to note that, within policies, action types are constraints that regulate the actions executable by queries, whereas within action signatures, they model the actions executed by the considered query. Given this semantics, the joint access component *Ac.Ja* specifies the categories of data jointly accessed by Q. The joint access performed by Q is defined by considering all the data categories to which the accessed attributes belong to, as the following example clarifies.

Example 5 Let us consider the query: *select avg(temperature) from sensed_data s join users u on s.watch_id=u.watch_id where u.user_id like 'Bob'*. This query accesses the columns *temperature* and *watch_id* of *sensed_data* and *user_id* and *watch_id* of *users*. It performs a direct access from a single source with aggregation on column *temperature* and an indirect access to the remaining columns. Based on Table 2, *temperature* has been classified as *sensitive*, *watch_id* (of both the tables) as *quasi identifier*, and *user_id* as *identifier*. The joint access component of the action type that models the access to *temperature* is derived as the union of the data categories to which the other accessed attributes belong to. The derivation results into the set {*quasi identifier, identifier*}, as such the derived joint access component is $\langle a, a, n, n\rangle$.

The set composed of all the action signatures that describe the actions performed by a query on a table is denoted as *table signature*. More precisely, given a table T accessed by Q, the *table signature* of T is a tuple that collects the *action signatures* that refer to accesses to attributes of T. The query

5. Given a table T, we denote with A_T the attribute set of T.

signature of a query Q is defined by grouping the table signatures associated with the tables accessed by Q, together with the access purpose of Q, and the query signatures of Q sub-queries.

Definition 4 (Table signature)**.** *Let Q be a query and let $T_1..T_k$ be the tables accessed by Q. The table signature Ts_i defined for table $T_i \in \{T_1..T_k\}$, is a pair $\langle T_i, Acs\rangle$, where Acs is the set of action signatures of Q that refer to the access to T_i columns.*

The query signature Qs of Q is a tuple $\langle Ap, Tss, Qss\rangle$, where: 1) $Ap \in Ps$ specifies the access purpose of Q, 2) Tss is the set of table signatures derived for Q, each of which groups the action signatures derived for a different accessed table, and 3) Qss is a set of query signatures corresponding to Q sub-queries.

Example 6 Supposing that the query introduced in Example 5 is executed with access purpose *research*, the corresponding query signature is: ⟨*research*, {⟨*sensed_data*,{⟨{*temperature*},⟨*d*,*s*,*a*,⟨*a*,*a*,*n*,*n*⟩⟩⟩, ⟨{*watch_id*},⟨*i*,*s*,*n*,⟨*a*,*a*,*a*,*n*⟩⟩⟩}⟩,⟨*users*,{⟨{*user_id*,*watch_id*}, ⟨*i*,*s*,*n*,⟨*a*,*a*,*a*,*n*⟩⟩⟩}⟩}, ∅⟩ .

4.4 Policy compliance

Checking compliance of queries with a policy requires to compare components of policies with those of query signatures. Since policies and query signatures are complex elements, the analysis is shifted down to finest grained elements. We start considering the action types specified by policy rules and query action signatures. Checking compliance requires to verify that: 1) the components characterizing the type of operation performed on data are the same, 2) the components of the joint access constraint tuple specified for the action signature that are set to *a* are also set to *a* in the analogous component in the considered rule. More formally:

Definition 5 (Action type compliance)**.** *Let Ac be the action type specified for a policy rule R and let Ac′ be the action type specified for an action signature. Ac′ is said to be compliant with Ac iff: $Ac'.Ia = Ac.Ia \wedge Ac'.Ms = Ac.Ms \wedge Ac'.Ag = Ac.Ag \wedge (Ac'.Ja.i = a \rightarrow Ac.Ja.i = a) \wedge (Ac'.Ja.s = a \rightarrow Ac.Ja.s = a) \wedge (Ac'.Ja.q = a \rightarrow Ac.Ja.q = a) \wedge (Ac'.Ja.g = a \rightarrow Ac.Ja.g = a)$.*

Example 7 Let $Ac = \langle d, s, a, \langle a, a, a, n\rangle\rangle$ be the action type of a policy rule that regulates the access to the *temperature* field of *sensed_data*'s selected tuples. Let us evaluate the compliance of *Ac* with the action type *Ac′*=$\langle d, s, a, \langle a, a, n, n\rangle\rangle$ specified by the action signature that models the access to attribute *temperature*, described in Example 6. The indirection, multiplicity and aggregation dimensions of *Ac* and *Ac′* specify the same values. Moreover, *Ac* grants the joint access to data of categories *identifier*, *quasi identifier*, and *sensitive*, whereas *Ac′* performs a joint access to data of categories *identifier* and *quasi identifier*. Therefore, *Ac′* complies with *Ac*.

Besides action types, action signatures and policy rules also include the components purpose set and attribute set. Compliance analysis requires to compare the components specified for the action signature with the corresponding ones specified for the policy rule. The analysis of purpose and attribute components is quite straightforward, as it requires that the elements aggregated in the components specified for the action signatures are also included in the components specified for the policy rule.

Compliance of a query with respect to a policy is defined through the composition of the previously proposed analysis. More precisely, let Q be a query and Qs be its signature. Q is said to be compliant with a policy PP specified for a tuple of a table T accessed by Q if there exists a rule R in PP which is defined in such a way that: 1) the set of columns of T accessed by Q is a subset of those referred to by R.Cl,[6] 2) every action type specified for the actions performed by Q on T complies with the action type R.At, and 3) the access purpose of Q is included in the list of access purposes in R.Pu. Formally:

Definition 6 (Policy compliance)**.** *Let Q be a query that accesses data in table T_i, let Qs be the query signature of Q, and PP_j be a policy either specified for a given tuple t_k of T_i (i.e., $PP_j.T = T_i \wedge PP_j.t = t_k$), or for the whole table T_i (it is therefore applied to every tuple t of T_i). Let Ts_i be the table signature of Qs which refers to T_i.*

Qs is said to be compliant with PP_j iff: $\forall As \in Ts_i.Acs \rightarrow \exists R \in PP_j.Rs \wedge As.Cs \subseteq R.Cl \wedge$ (∀at ∈ As.Ac →(at complies with R.At)[7]) $\wedge Qs.Ap \in R.Pu$.[8]

5 The framework

In this section we discuss selected aspects of our framework, concerning configuration, policy specification and enforcement.

5.1 Framework configuration

Some preliminary configuration activities are required to support policy specification (see Figure 1). We assume a starting situation in which the data are stored into an already defined relational DB, referred to as the *target DB*. However, the configuration activities described in this section can also be applied in situations where the target DB is generated from scratch.

First, the DB administrator has to define the set of purposes that will be used for policy specification. This is achieved by introducing table *Pr(Id, Ds)* into the target database, where *Id* represents the purpose identifier, whereas *Ds* the purpose description. Considering our running example, table *Pr* includes the tuples: ⟨*p1, treatment*⟩, ⟨*p2, payment*⟩, ⟨*p3, healthcare-operations*⟩, ⟨*p4, law-enforcement*⟩, ⟨*p5,reporting*⟩, ⟨*p6, research*⟩, ⟨*p7, marketing*⟩ and ⟨*p8, sale*⟩.

Afterwords, the administrator has to classify the attributes of every table in the target DB based on the data categories introduced in Section 4.1. The categorization is operated at schema level, as such all the data items in the same table column belong to the same category. The classification is achieved introducing table *Pm(At, Tb, Ct)* into the target DB, which keeps track of the data category of every table column. More precisely, within table *Pm* the

6. Here and in what follows we use the dot notation to refer to selected components within a tuple.

7. Based on Def. 5.

8. If $PP_j.t = \perp$, PP is applied to every tuple of T.

columns *At* and *Tb* respectively keep track of the attributes to be categorized and the tables to which these attributes belong to, whereas *Ct* specifies the data category of *At*. The content of table *Pm* for the *patients* table is shown in Table 2.[9]

Then, the administrators have to define the purpose authorizations for the registered users. This is achieved by introducing table *Pa(Ui, Pi)* into the target DB, where *Ui* specifies the identifier of a user *u* to whom the authorization is granted, whereas *Pi* is the identifier of a purpose authorized for *u*.

Finally, the administrator has to alter the schema of the target DB tables to keep track of the enforced data policies. This is achieved by introducing a column *policy* into each table of the target DB. *policy* is defined as a binary attribute of variable length.[10]

5.2 Query signature generation

Query signatures must be derived from the SQL source code of the analyzed queries. For space limitations, in this section we only introduce the main steps of the derivation at a high level of abstraction. A more thorough description of the query signature derivation process is provided in Appendix A.

In order to ease the presentation of the derivation approach by abstracting the process from SQL code details, we introduce an abstract representation of SQL queries denoted as *query model*.

Definition 7 (Query model)**.** *A query model Qm of an SQL query Q is a tuple ⟨S,F,W,G,H⟩, where S models the select item(s) of Q, F models the data source(s) accessed by Q, W models the where clause, G represents the group by and H the having clause of Q, if any. More precisely:*

- *S models the items in the select statement of Q which are specified as a set of column expressions within which multiple attributes of the data sources specified in component F are referred to. Such attributes can also be used as input parameter of scalar or aggregate functions.*
- *F is a set of table expressions which specify the data sources accessed by Q. These data sources can be tables or subqueries. A table expression can even specify the join of multiple data sources.*
- *W is an expression specifying the where condition of Q. W can include sub-queries where attributes of the data sources specified in F are referred to.*
- *G specifies the group by clause of Q. G consists of a set of attributes of the data sources specified in F.*
- *H is an expression specifying the having clause of Q. H specifies aggregate functions that operate on attributes of the data sources referred to in F.*

Query models can be easily derived analyzing the source code of queries with an SQL parser,[11] however, for space limitations, in this paper we do not illustrate the implementation of the parsing activity.

The expressions defined within the clauses of the query refer to attributes of the accessed data sources. The clauses and the expressions that contain the attribute references provide information on the type of access performed on the referred attributes. Such information is used to derive the query signature. We model the access information associated with each referred attribute by means of a data structure, called *info tuple*.

Definition 8 (Info tuple)**.** *Let Q be a query and let V be an attribute referred to within an expression specified for a clause of Q. The info tuple specified for V is a tuple ⟨Id: Text, Ds:Text, Qi:Text, Ia:{d, i}, Ms:{s, m}, Ag:{a, n}, Ct:{i, s, q, g}, Ja:⟨i:{a,n}, s:{a,n}, q:{a,n}, g:{a,n}⟩⟩, where: Id is the name of the attribute V, Ds is the data source to which V belongs to, Qi specifies the identifier of the (sub)query within which the attribute has been referred to,[12] Ia specifies the indirection of the access to V, Ms specifies if V depends on a single or multiple data sources, Ag specifies if V is referred to as input parameter of an aggregate function, Ct specifies the data category of V, Ja specifies which other data categories are jointly accessed with V, whereas Pu specifies the access purpose of Q.*

The derivation of the query signature for a given SQL query Q is achieved by means of a stepwise process composed of three phases: 1) derivation of the query model *Qm* of Q and initial generation of *info tuples* for each attribute referred to within the expressions of each clause of *Qm*; 2) completion of the *info tuples* initializing the data category component *Ct* and joint access component *Ja*; and 3) analysis of the information specified within the *info tuples* and composition of action signatures, table signatures and query signatures.

Example 8 Let us consider the derivation of the query signature for the query: *select user_id, avg(beats) from users join sensed_data on users.watch_id= sensed_data.watch_id group by user_id having avg(beats)>90*, specifying *healthcare-operations* (p3) as access purpose.

In phase 1, the query is parsed deriving the query model *Qm=⟨{"user_id", "avg(beats)"},"users join sensed_data on users.watch_id= sensed_data.watch_id",⊥, {"user_id"}, "avg(beats)>90"⟩*. The analysis of the expressions in each clause of *Qm* generates the *info tuples* shown in the upper part of Figure 3. During phase 2 the info tuples are updated initializing the *Ct* and *Ja* components. The *Ct* component of each *info tuple* is initialized to the data category of the referred table column, which is specified in table *Pm*, whereas the *Ja* component is set deriving the union of the data categories of the jointly accessed table columns. Finally, in phase 3, the information specified within the *info tuples* is analyzed and composed to form the action signatures, table signatures, and query signature of the considered query. The derived query signature is composed of two table signatures that respectively refer to table *users* and *sensed_data*. The table signature of *users* includes three action signatures, one specifying the direct access to the columns *user_id* and two the indirect access to *user_id* and

9. Columns *At*, *Tb*, and *Ct* of *Pm* correspond to columns Attribute, Table, and Category of Table 2.

10. The choice of using a binary type is due to the adopted policy encoding strategy (see Section 5.3).

11. We have implemented the generation process using SQL General Parser (http://www.sqlparser.com).

12. If not specified, the identifier is derived as the hash of the query string.

Query: *select user_id, avg(beats) from users join sensed_data on users.watch_id= sensed_data.watch_id group by user_id having avg(beats)>90*
Access purpose: *healthcare-operations* (p3)
Phase 1 & 2 : Derivation of Info Tuples (values between round parenthesis have been set in Phase 2)

Id	Ds	Qi	Ia	Ms	Ag	Ct	Ja⟨i,q,s,g⟩	Pu
user_id	users	c94f2b5c	d	s	n	⊥(i)	⊥(⟨n,a,a,n⟩)	p3
beats	sensed_data	c94f2b5c	d	s	a	⊥(s)	⊥(⟨a,a,n,n⟩)	p3
watch_id	users	c94f2b5c	i	⊥	⊥	⊥(q)	⊥(⟨a,a,a,n⟩)	p3
watch_id	sensed_data	c94f2b5c	i	⊥	⊥	⊥(q)	⊥(⟨a,a,a,n⟩)	p3
user_id	users	c94f2b5c	i	⊥	⊥	⊥(i)	⊥(⟨n,a,a,n⟩)	p3
beats	sensed_data	c94f2b5c	i	⊥	⊥	⊥(s)	⊥(⟨a,a,n,n⟩)	p3

Phase 3 : Derivation of action signatures, table signatures, and query signature

Qs	Ap	Tss						Qss
		Ts	Cs	Ia	Ms	Ag	Ja	
c94f2b5c	healthcare-operations	users	{user_id}	d	s	n	n, a, a, n	∅
			{watch_id}	i	⊥	⊥	a, a, a, n	
			{user_id}	i	⊥	⊥	n, a, a, n	
		sensed_data	{beats}	d	s	a	a, a, n, n	
			{watch_id}	i	⊥	⊥	a, a, a, n	
			{beats}	i	⊥	⊥	a, a, n, n	

Figure 3. Query signature derivation example

watch_id, whereas the table signature of *sensed_data* groups three action signatures, one specifying the direct access to *beats*, and the other two specifying the indirect access to *watch_id* and *beats*. The lower part of Figure 3 shows the derived query signature.

5.3 Encoding strategies

Users can either specify policies for data tuples already stored into the target DB tables, or insert new records (which already include the policies).

Since our framework supports fine grained policies, we pay particular attention to their encoding to reduce the overhead in terms of memory usage and execution time of the policy compliance process. To increase efficiency, we use a uniform encoding for policies and queries. The encoding uses constructs, denoted as *masks*, which model the elements forming policies and query signatures (see Sections 4.2 and 4.3) as binary strings.

A *policy mask* is a collection of *rule masks* each encoding a single policy rule. Each rule mask is composed of sub-masks that model purposes, attributes, and action types.

Let us start considering the encoding of Pu, the purpose component of a rule R (see Def. 2). Each purpose element is specified within the purpose set table Pr of the target DB. Let Oc be an ordering criterion specified for the elements in Pr. We denote with p_i the *i-th* purpose element of Pr, ordered according to Oc.

Definition 9 (Purpose mask)**.** *Let PP be a policy, R a rule included in PP, and Pu the set of purposes specified by R. The purpose mask Pm that encodes Pu is a binary string defined as* $\sum_{p_i \in Ps} v$*, where* $v =' 1'$*, if* p_i *is among the authorized purposes in Pu,* $v =' 0'$*, otherwise.*[13]

Example 9 Let us consider the encoding of the purpose component *{p1,p3,p4,p6}* specified within the policy rule *r2* in Example 4. Let us assume that the alphabetic order of purpose identifiers is the ordering criterion to visit *Pr*. Based on Def. 9, the purpose mask derived for *r2* is the binary string *'10110100'*.

Let us now consider the encoding of attributes constrained by a policy rule R, specified by component Cl.

Definition 10 (Column mask)**.** *Let PP be a policy, R a rule included in PP, and T the table that includes the data tuple(s) for which PP has been specified. Let* a_i *be the i-th attribute of T wrt an ordering criterion* Vo*. The column mask Cm that encodes the Cl component of R is a binary string defined as* $\sum_{a_i \in A_T} v$*, where v='1', if* a_i *is among the attributes to which R is applied, v='0', otherwise.*

Example 10 Let us consider the policy rule *r2* in Example 4. The column set Cl of *r2* includes the elements *temperature* and *beats*, which respectively represent the 3^{rd} and 5^{th} attribute of *sensed_data*. As such, based on Def. 10, the column mask derived for *r2* is *'00101'*.

Finally, let us consider the encoding of the action type At specified by a policy rule R (cfr. Def. 1).

Definition 11 (Action type mask)**.** *Let PP be a policy, R a rule of PP, and T the table storing the tuple(s) for which PP has been specified. The action type mask Am that encodes the action type At of R is a string with the format "i d s m a n i q s g", where each token is set either to '0' or to '1'.*

More precisely, according to Def. 1, *i* and *d* specify the indirection of the access (*i*=indirect and *d*=direct), *s* and *m* the multiplicity (*s*=single and *m*=multiple), *a* and *n* the aggregation dimension (*a*=aggregation and *n*=no aggregation). The remaining fields model the joint access constraint, where each token refers to a data category (*i*=identifier, *q*=quasi identifier, *s*=sensitive, *g*=generic).

Example 11 Let us consider the action type specified within the policy rule *r2* in Example 4, which specifies the direct access from single source with no aggregation and the joint access to sensitive data. The corresponding action type mask is the binary string '0110010010'.

A rule mask is defined by the composition of an attribute, purpose, and action type mask.

13. With $\sum$ we denote the string concatenation operation.

Definition 12 (Rule mask)**.** *Let PP be a policy, R a rule of PP, and T the table storing the tuple(s) for which PP has been specified. The rule mask Rm of R is defined through the concatenation of the attribute, purpose and action type masks respectively derived from Cl, Pu, and At components of R. More precisely Rm=Cm+Pm+Am, where "+" is the string concatenation operator.*

Example 12 The rule mask *Rm* that encodes the policy rule *r2* in Example 4 is defined through the concatenation of the column, purpose, and action type masks derived in Examples 9, 10, 11, respectively. As such, *Rm* is set to '00101101101000110010010'.

A rule mask Rm is a binary string of variable length whose size depends on the cardinality of the purpose set Ps and the number of attributes that compose the table T (i.e., $|A_T|$) to which it refers to. Rm has length $|Ps| + |A_T| + k$, where k is the fixed size of the action type mask (see Def. 11).

A policy mask is defined by concatenating all the rule masks defined for the policy rules.

Definition 13 (Policy mask)**.** *Let PP be a policy, Rs the set of rule composing PP, and T the table that includes the tuple(s) for which PP has been specified. The policy mask PPm of PP is defined through the concatenation of the rule masks defined for each rule $R \in Rs$. More precisely, PPm=$\sum_{R \in Rs} Rm_R$, where Rm_R denotes the rule mask Rm of rule R.*

Policy compliance verification for a query *Q* is based on the compliance of the actions executed by *Q* with the policies specified for the data accessed by *Q*. This is done by checking the access purpose and the action signatures of its query signature *Qs* against the rules that compose the policies of the accessed data. To ease this checking, an action signature *As* is encoded as a binary string concatenating the binary masks *Cm* and *Am*, respectively derived for the components *Cs* and *Ac* of the action signature *As*, and *Pm*, i.e., the mask of the purpose component *Ap* of the query signature *Qs* that includes *As*. The encoding of *Cm*, *Am*, and *Pm* follows the same criteria defined for the corresponding attribute, action type and purpose components of policy rules.

Definition 14 (Action signature mask)**.** *Let Qs be the query signature derived for a query Q, As be an action signature of Qs, and Ap the access purpose of Q. The action signature mask that encodes As is a binary string Asm=Cm+Pm+Am, where Cm is the column mask of As, Pm the purpose mask derived for Ap wrt the purpose set in table Pr, whereas Am is the action type mask corresponding to component Ac of As.*

Example 13 Let us consider the action signature *As*=⟨*{temperature}, d, s, a,* ⟨*a, a, n, n*⟩⟩ belonging to the query signature *Qs* derived for the query in Example 6. Based on Def. 14, the action signature mask of *As* is '00100000010000110101100'.

5.4 Compliance analysis

Compliance analysis is based on action signature masks and policy masks introduced in the previous section. Based on Def. 6, checking policy compliance requires to verify whether each action signature of every table signature collected in a query signature complies with at least one of the rules composing the target policy. Compliance analysis is implemented starting from the basic elements of the query signature and the policy. We start considering action signatures and policy rules. Based on Def. 5, an action signature complies with a policy rule if: 1) the columns set of the signature is a subset of the one defined by the rule, 2) the action types of the action signature and the policy rule are compliant (see Def. 5), and 3) the access purpose specified by the action signature is among the purposes defined by the policy rule.

Taking benefit from the binary encoding presented in Section 5.3, this check is implemented by means of a "bitwise and" on the corresponding masks. If the result of this operation is a binary string equal to the action signature mask, the compliance is satisfied.

Definition 15 (Action signature rule compliance)**.** *An action signature As complies with a rule R iff the "bitwise and" between the action signature mask Asm corresponding to As and the rule mask Rm corresponding to R returns a binary string equal to Asm. More precisely, the compliance is satisfied iff $Asm \,\&\, Rm = Rm$.*

Based on Def. 6, compliance of *As* with a policy *PP* is verified iff there exists at least one policy rule among those specified by *PP* which satisfies the compliance. Policy compliance is implemented by extracting from *Pm* (i.e, the policy mask of *PP*) the rule mask Rm_i of each rule in *PP*, and then checking its compliance. The rule mask extraction is done by partitioning the policy mask based on the length of the rule mask, which in turn is derived from the cardinality of the column set, purpose set, and action type component (the latter is a constant). Let *split* be a function that receives as input a string *s* and an integer parameter *n*, and partition *s* into substrings of length *n*.

Definition 16 (Action signature policy compliance)**.** *An action signature As complies with a policy PP iff given 1) Asm, the action signature mask of As, 2) Pm, the policy mask of PP, and 3) Rml, the length of the rule masks in Pm $\exists rm \in split(Pm, Rml) \wedge Rm \& Asm = Asm$*

Listing 1 shows the pseudocode of function *compliesWith*, which checks the compliance of an action signature *As* with a policy *PP*, based on Def. 16.

Listing 1. Compliance check: function compliesWith

```
function compliesWith
(BitVarying asm, BitVarying pm):Boolean{
 Integer rmn= size(pm) div size(asm)
 if(size(pm) mod size(asm) ≠ 0) return false
 Boolean acomp=false
 for (i in 0..rmn){
  BitVarying rm= substring(pm, i*rml,(i+1)*rml-1)
  acomp=acomp ∨ (asm & rm=asm)
  if (acomp) return true
 } return acomp }
```

Finally, a query signature *Qs* is said to be compliant with a policy *PP* specified for data stored into a table *T*, if all the action signatures of *Qs* which specify an access to *T* columns comply with *PP*. More formally:

Definition 17 (Query signature policy compliance)**.** *Let Qs be a query signature, let PP be a policy, and let Pm be the*

policy mask of PP. Let Rml be the length of the rule masks composing Pm. The query signature Qs is said to be compliant with the policy PP iff $\forall Ts \in Qs.Tss(Ts.T = P.T) \rightarrow \forall Asm \in split(Tsm, Rml) \rightarrow \exists Rm \in split(Pm, Rml) \wedge Rm\&Asm = Asm$.

5.5 Policy enforcement

A tuple *tp* of a table *T* can only be accessed by a query *Q* if all action signatures of the query signatures of *Q* and *Q* sub queries which refer to columns of *T* comply with the policy specified for *tp*. This constraint is enforced rewriting the SQL code of *Q*. More precisely, the expressions specified within the *where* clause of *Q* and *Q* sub-queries are extended by conjuncting an expression, which through the invocation of *compliesWith*, checks the compliance of the (sub-)query with all the policies specified for the accessed data tuples. Functions *rewriteQuery* and *rwSubQueries*, whose pseudocode is shown in Listing 2, implement the rewriting.

Listing 2. Query rewriting functions

```
function rewriteQuery
(QueryModel qm, QuerySignature qs):Text{
 rwSubQueries(qm.S) rwSubQueries(qm.F)
 rwSubQueries(new Set(){qm.W})
 rwSubQueries(new Set(){qm.H})
 for(ts in qs.Tss){
  for(as in ts.Acs){
   BitVarying asm=getASmask(as,qs.Ap)
   Text extCd=
    "compliesWith(b'"+asm+"',"+ts.T+".policy)"
   qm.W= qm.W!=⊥? qm.W+"and"+extCd:extCd
 }} return toSqlCode(qm) }
function rwSubQueries (Set(Text) expS,
 QueryModel qm, QuerySignature qs){
 for(Text sqSrc in getSubQueries(expS)){
  QueryModel sqm=getQueryModel(sqSrc)
  QuerySignature sqs=select * from qs.qss
   where id=sqm.id
  Text rwsq=rewriteQuery(sqm,sqs)
  replace(expS,sqSrc,rwsq) }}
```

Let *qm* and *qs* be the query model and the query signature of a query Q, respectively. Function *rewriteQuery* rewrites the *where* clause of Q based on the information specified within the query model *qm* and the query signature *qs*. The original *where* clause is conjuncted with an expression within which *compliesWith* is invoked to check the compliance of all the action signatures in *qs* with the policies specified for the accessed tuples within the respective column *policy*. Since some of the clauses of Q can include expressions specifying sub-queries, the rewriting is propagated to the sub queries, through the invocation of function *rwSubQueries* (see Listing 2). *rwSubQueries* analyzes the expressions specified within the components *S*, *F*, *W* and *H* of *qm*, each of which could include a sub query. As such, *rwSubQueries* extracts the source code of each sub-query and after deriving the query model *sqm* and selecting the corresponding query signature *sqs* recursively invokes *rewriteQuery* on such elements. Finally, the source code of each analyzed sub-query is substituted by the rewritten code derived through the execution of *rewriteQuery*.

Example 14 Let us consider the query of Example 8. *rewriteQuery* is invoked specifying as actual parameters the query model *qm* and the query signature *qs*, both derived in Example 8. The *Tss* component of *qs* includes 2 table signatures respectively referring to tables *users* and *sensed_data*, each collecting two action signatures (see Figure 3). As specified in Listing 2, *rewriteQuery* first derives the action signature mask of each action signature in *qs* by concatenating the column, purpose, and action type masks of the corresponding action signature components. The derived masks are composed to form the expression *extCd* (see Listing 2), which is used to initialize component *W* of *qm*. Once this expression has been derived, function *toSqlCode* derives the SQL code of the rewritten version of the query. The generated code is shown in Listing 3.

Listing 3. Example of rewritten query

```
select user_id, avg(beats) from users join sensed_data
 on users.watch_id = sensed_data.watch_id where
 compliesWith(b'10000001000000110010110O',
   users.policy) and
 compliesWith(b'01000001000001000001110O',
   users.policy) and
 compliesWith(b'10000001000001000000110O',
   users.policy) and
 compliesWith(b'00001001000000110101100O',
   sensed_data.policy) and
 compliesWith(b'10000001000001000001110O',
   sensed_data.policy) and
 compliesWith(b'00001001000001000001100O',
   sensed_data.policy)
group by user_id having avg(beats)>90
```

5.6 Complexity Analysis

The *complexity* of a rewritten query *q* generated with the proposed enforcement framework is measured as the number of policies that are checked during *q* execution. The complexity upper bound can be estimated by means of a static query analysis. More precisely, the dimensions affecting the complexity upper bound are: 1) the number of accessed tables, 2) the number of tuples in each table, and 3) the number of action signatures derived for the query considered which are checked for compliance with the policies associated with the accessed data.

Let us consider a target database *db* containing *N* tables T_i ($1 \leq i \leq N$). Let *q* be a rewritten query which accesses tables of *db*, and let us assume that table T_i includes n_i tuples, and a policy has been specified for each of these tuples. Let us denote with *primitive* a (sub)query that does not include other sub-queries, whereas *structured* is a (sub)query containing other sub-queries.

Let us start considering the case when *q* is primitive. *q* accesses k tables ($1 \leq k \leq N$), whereas j_i ($1 \leq j_i \leq 5$) action signatures are defined for each accessed table T_i (see the derivation process described in Section 5.2). The maximum number of policy compliance checks required to evaluate the compliance of one action signature specified for a table T_i with the policies specified for T_i's tuples is n_i (i.e., the number of tuples stored in T_i). The total number of checks for the action signatures referring to table T_i is derived multiplying n_i, the complexity of a single action signature, by j_i, the number of action signatures that refer to T_i's columns. The overall complexity of *q* is derived summing the policy checks required for each accessed table. Therefore, since *q* accesses k tables, the complexity upper bound is $\Sigma_{i=1}^{k} n_i * j_i$. It is worth noting that the actual complexity can be far from this bound. Indeed, due to filtering

expressions specified within the clauses *where*, *having* and *join* conditions, the effective number of accessed tuples per table can be far lower than n_i. Moreover, in case the *a-th* action signature of table T_i (with $a < j_i$) does not comply with the policy specified for the tuple under analysis, the whole *where* condition is evaluated *false* and the remaining $(j_i - a) + \Sigma_{b=i}^{k} j_b$ action signatures are not checked.

The complexity upper bound of *structured* queries is derived by composition of the complexity of the included sub-queries. More precisely, let us denote with q_s^{cp}, where $cp : \{S, F, W, H\}$, the *s-th* sub-query specified within component *cp* of *q*, and let us denote with $|cp|$ the number of sub-queries which are specified within *cp*. The complexity upper bound of *q* is derived by means of a recursive function *cub* defined as follows.[14]

$$cub(q) = \begin{cases} \Sigma_{i=1}^{k} n_i * j_i, \text{if } q \text{ is } primitive \\ \Sigma_{i=1}^{k} n_i * j_i + \Sigma_{cp \in \{S,F,W,H\}} \Sigma_{s=1}^{|cp|} cub(q_s^{|cp|}), \text{else} \end{cases} \quad (1)$$

Although no constraint is imposed to the use of the proposed access control framework, we can assume it will be used in common application scenarios for DBMSs. By assuming a typical scenario we can have a quantitative indication of the problem size, with upper bound values of the variables in Equation (1). More precisely, we assume that: 1) the target DB includes tens of tables ($N < 100$) each characterized by tens of attributes and each storing thousands to millions tuples ($n_i < 10^7$); and 2) queries specify up to 2 layers of subqueries, and no more than 1 sub-query is included in each query clause ($|cp| \leq 1$).

5.7 Correctness of query rewriting

Query rewriting correctness is stated by Theorems 1 and 2.

Let *Q* be a query that accesses a set of tables $Ts = T_1..T_n$, *Qs* be the query signature of *Q*, *Q'* be the rewritten version of *Q*, and *Rs* and *Rs'* be the result sets of *Q* and *Q'*, respectively. Let us denote with AcS_T the set of all the action signatures of *Qs* which model actions executed by *Q* on given columns of a table *T* (where $T \in Ts$). Given a tuple *t* of *Rs*, we denote with S_{t_T} the *supplier tuples set* of *t* in *T* (where $T \in Ts$), i.e., the set of tuples of *T* from which *Q* derives *t*. Let us denote with s_{t_T} a supplier tuple belonging to the set S_{t_T}.

Theorem 1 (Security)**.** *For each tuple $t' \in Rs'$, the policy specified for the supplier tuple $s_{t'_T}$ of each accessed table T ($T \in Ts$) complies with each action signature of AcS_T. More precisely, $\forall t' \in Rs' \forall T \in Ts \forall s_{t'_T} \in S_{t'_T} \forall As \in AcS_T \rightarrow$ compliesWith(Asm, $s_{t'_T}$[policy]), where Asm is the action signature mask of As and $s_{t'_T}$[policy] is the projection of the supplier tuple $s_{t'_T}$ on column policy .*

Let us assume an order relation *o* for the tuples belonging to S_{t_T}. We denote with $s^i_{t_T}$ the i-th tuple of S_{t_T} wrt *o*.

Theorem 2 (Completeness)**.** *Given a tuple $t \in Rs$ if for each $T \in Ts$ the policy specified for the i-th tuple $s^i_{t_T}$ of the supplier tuple set S_{t_T} of t complies with each action signature in AcS_T, then there exists a tuple $t' \in Rs'$ such that for each $T \in Ts$ the set of supplier tuples of t' in T includes $s^i_{t_T}$. More precisely, $\forall t \in Rs \forall T \in Ts \forall i (1 \leq i \leq |S_{t_T}|) \forall As \in AcS_T$ (compliesWith(Asm, $s^i_{t_T}$[policy]) $\rightarrow \exists t' \in Rs' \wedge \forall T \in Ts \rightarrow s^i_{t_T} \in S_{t'_T}$*

Formal proofs are reported in Appendix B.

6 Performance analysis

In order to assess the performance of the proposed enforcement mechanism, we compare the execution time of rewritten and original queries by varying: 1) the characteristics of the policies specified for the accessed data, and 2) the size of the accessed dataset.

The analysis is based on the running example. Patients data are generated in such a way that: 1) each patient is described by one tuple in *users*, one in *nutritional_profile*, and multiple tuples in *sensed_data*, each representing a sensed sample; and 2) a policy is specified for each tuple of *patients*, whereas all tuples of *sensed_data* referring to the same smart watch are covered by the same policy as represent samples sensed for the same patient.

In the remaining of this section, we discuss characteristics of policies, SQL queries and datasets which have been used for performance analysis.

6.1 Properties of policies

Let *T* be a table containing *n* tuples tp_i, where $1 \leq i \leq n$, each covered by a policy. Let q_{or} be a query accessing *T* and q_{rw} be the rewritten query derived from q_{or}. We define the *selectivity* of the policies specified for the tuples in *T* wrt q_{rw}, as the percentage of tuples which, due to non-compliance with the specified policies, have not been used to derive q_{rw}'s result set.

Definition 18 (Policy selectivity wrt a query)**.** *Let Q be a query and let q_{rw} be the corresponding rewritten query. Let T be a table accessed by Q. The selectivity s of the policies specified for the tuples of T evaluated wrt the execution of q_{rw}, is defined as $s = 1 - |Atp|/|Ttp|$, where Ttp is the set of all T tuples that q_{rw} tries to access, and $Atp \subseteq Ttp$, is a set that includes all Atp tuples that based on policy compliance are used to compute the result set of q_{rw}.*

If $s = 0$, the result set of the rewritten query is equal to the result set of the original query. In this case q_{rw} complies with all the policies and no tuple is discarded. In contrast, when $s = 1$, no access is allowed.

For testing purposes, we assess the enforcement overhead wrt policy selectivity.

Given a table *T* including *n* tuples, we want to generate policies that provide a given selectivity level *s* wrt *no filtering queries*, namely SQL queries that do not include *where*, *having* and *join on* expressions, and thus access all tuples of *T*. The choice of no filtering queries allows the straightforward definition of a given selectivity level wrt *n*. More precisely, in order to generate policies for *T* with selectivity *s* wrt *no-filtering* queries, we should generate $s*n$ policies that do not comply with any query action signature, and $(1-s)*n$ policies that comply with all query action signatures. To achieve this, policies are generated as composed of *pass-all* and *pass-none* rules. A *pass-all* rule is a policy rule represented by a

14. All variables referred to in the definition of *cub* refer to the sub-query on which the function has been invoked.

q1	`select distinct watch_id from sensed_data`
q2	`select count(watch_id) from sensed_data`
q3	`select count(watch_id) from sensed_data where not watch_id like 'watch100'`
q4	`select food_intolerances, count(user_id) from users join nutritional_profiles on users.nutritional_profile_id=nutritional_profiles.profile_id where not food_intolerances like 'no_intolerance' group by food_intolerances`
q5	`select user_id, temperature from users join sensed_data on users.watch_id=sensed_data.watch_id where sensed_data.temperature>37 and timestamp>0`
q6	`select user_id, avg(temperature), avg(beats) from users join sensed_data on users.watch_id=sensed_data.watch_id where timestamp >0 and nutritional_profile_id in (select profile_id from nutritional_profiles where not food_intolerances like 'no_intolerance') group by user_id`
q7	`select user_id, avg(beats), food_preferences from users join sensed_data on users.watch_id=sensed_data.watch_id join nutritional_profiles on users.nutritional_profile_id=nutritional_profiles.profile_id where diet_type like 'low_sugar' group by user_id, food_preferences`
q8	`select user_id, avg(s1.b) from users join (select watch_id as w, beats as b from sensed_data where beats>100) s1 on users.watch_id=s1.w group by user_id`

Figure 4. Ad-hoc generated queries

rule mask with all binary digits set to '1'. Such a rule is compliant with any action signature. In contrast, a *pass-none* rule is characterized by a rule mask whose binary digits are all set to '0'. In this case, no action signature comply with the rule. Let us refer to the policies only composed of *pass-all* and *pass-none* rules as *scattered*.

Thus, to generate policies with selectivity s (such that $0 \leq s \leq 1$) wrt *no-filtering* queries, we randomly generate $s * n$ scattered policies only consisting of *pass-none* rules and $(1 - s) * n$ scattered policies each including one *pass-all* rule. The use of scattered policies has no implications on performance. It can be proved that no matter the rules in PP are pass-all, pass-none or common rules the compliance analysis of an action signature *As* wrt a policy *PP* takes the same time as the execution time does not depend on the values specified by the rule masks.

6.2 SQL Queries selection

We aim at testing the performance by considering a set of queries which covers a variety of the expressions specifiable within SQL selection statements. Therefore, we consider queries that: 1) select data from both single and multiple tables, 2) aggregate data from multiple tuples grouping them on different columns, 3) execute filtering operations on single and aggregate values, 4) include sub-queries, 5) perform multiple joins.

For the experiments we use two query benchmarks. The first one includes the queries *q1-q8* in Figure 4, which have been defined to cover the above mentioned selection cases. The second benchmark includes twenty automatically and randomly generated queries, referred to as *r1-r20* in what follows. This additional set has been added to show that the framework behavior is consistent with any type of query, and thus queries *q1-q8* have not been picked to show the best performance cases. The adopted generation approach for random queries, after analyzing the *patients* scheme, randomly selects the tables and the attributes which shall be accessed, and randomly derives the expressions that will characterize the projection, join, where, group by and having statements, based on the type of the selected attributes, and the values these elements can assume (e.g., the value of *temperature* is within a given range). Figure 5 summarizes characteristics of the generated queries.

Query	Description
r1, r12, r20	select from a single data source and aggregate data
r2, r7, r17	join multiple data sources, aggregate data and filter the grouped data
r3, r4, r14, r16	join multiple data sources
r5, r8, r11, r13, r15, r18	join multiple data sources and aggregate data
r6, r9, r10, r19	select from a single data source

Figure 5. Random queries

6.3 Experiments

Our experiments have been run on an Intel Core-i5 PC with 4GB of RAM, using PostgreSQL ver 9.1.9. Function *compliesWith* has been implemented as a PostgreSQL user-defined C function. The experiments analyze the enforcement overhead by varying 1) the selectivity of access control policies, 2) the size of the accessed dataset.

Experiment 1: We consider a scenario composed of 1,000 patients with 1,000 samples sensed for each patient. Therefore, tables *users* and *nutritional_profiles* store 1,000 tuples,

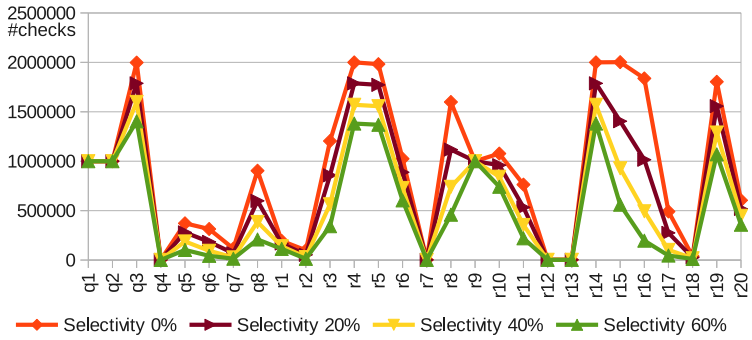


Figure 6. Policy compliance checks per query

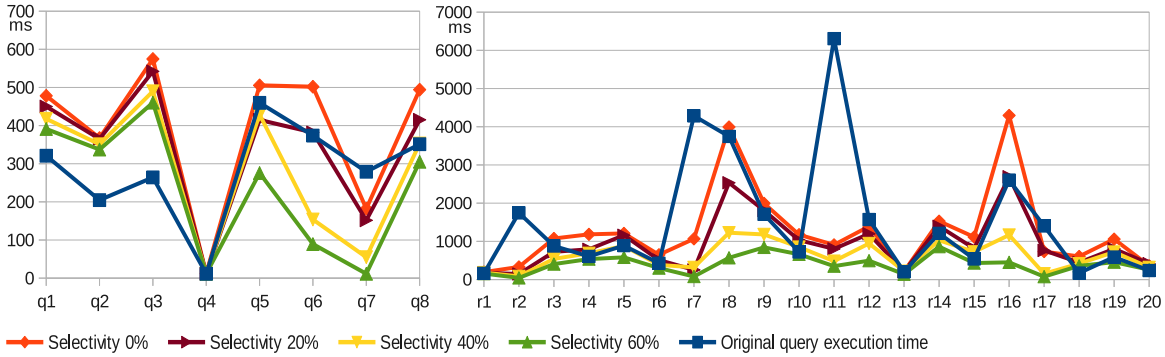


Figure 7. Query execution time vs policy selectivity

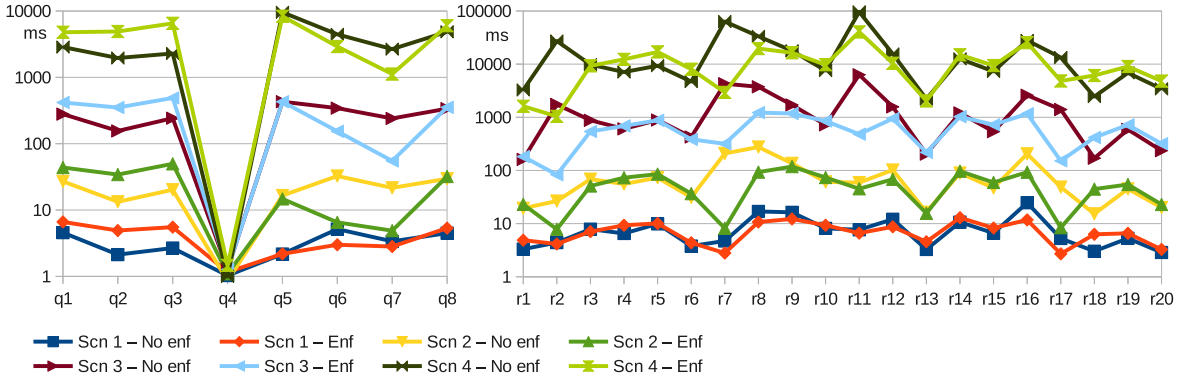


Figure 8. Query execution time vs dataset size

whereas *sensed_data* collects 1.000.000 tuples. While keeping the same data in these tables, we have run the original and rewritten versions of queries *q1-q8* and *r1-r20* by varying the selectivity of the policies specified for *users*, *sensed_data*, and *nutritional_profiles*. Policies have been generated in such a way that the same selectivity considered wrt *no-filtering* queries characterize the policies specified for each table. We have decided to specify scattered policies with selectivity values 0, 0.2, 0.4 and 0.6, and 1 to 3 rules each, [15] as we assume these values represent common use scenarios. Policy rules have a length of 24 bits (5 bits are used for the column set, 10 bits for the action type, 8 bits for the purposes, and 1 bit is added to get a rule length that allows the execution of string manipulation operations for rule masks extractions).

15. Policies have been defined in such a way that the number of rules that compose each policy varies uniformly in the range 1..3 and the position of the compliant rule (if any) varies with the same criterion.

Although in Section 5.6 we have shown that the complexity upper bound can be estimated using static query analysis, the complexity can also be accurately measured by keeping track and counting the number of policy compliance checks performed during query execution. The diagram in Figure 6 shows the complexity of the queries measured as the number of times function *compliesWith* is invoked to check the compliance of a query action signature with a policy. As shown by Figure 6, independently from the considered query, the complexity decreases with the growing of policy selectivity. Filtering and grouping expressions drastically cut the complexity of queries and this effect is multiplied in the presence of join operations

(e.g., *r5*, *r8*). Therefore, queries with apparently a very easy structure, such as *q1* and *q2*, have an higher complexity than queries that jointly access multiple tables. Another aspect to be considered is how policy selectivity affects the measured complexity. The selectivity has no considerable effect on queries that do not specify filtering conditions, such as *q1* and *q2*. As visible with query *q3*, when filtering conditions are specified, the selectivity magnifies the effect of the filtering. Similarly, as visible with *q4-q8*, *r3-r4* when multiple tables are jointly accessed, the selectivity further amplifies the effect of join conditions. As such, the number of tuples that are effectively accessed in the case of joint access decreases with the growth of the selectivity.

The main goal of this first set of experiments is to evaluate the execution time overhead of the rewritten queries. The results got with queries *q1-q8* and *r1-r20* are shown in Figure 7, where the execution time of the original queries is compared with those of the rewritten queries by varying the selectivity of the policies.

As shown by Figure 7, the greatest overhead is related to policies with selectivity 0. Indeed, in this case, the rewritten queries access the same number of tuples accessed by the original query, and a policy check is executed for each accessed tuple. With the increase of policy selectivity, the execution time of the same query progressively decreases. This trend is verified with both ad-hoc and randomly generated queries. This behavior is an expected consequence of the results shown by Figure 6), based on which, with the boost of polciy selectivity the number of policies to check decreases. A complementary cause of this effect is the cardinality of the result set. For instance, the rewritten versions of queries *q1* and *q2* have the same complexity (see Figure 6), but a different execution time. The result set of the original version of *q1*, includes 10^6 tuples, whereas the result set of *q2* includes only one tuple. As a result, the execution time of the original version of *q2* takes around 100ms less than *q1* (see Figure 7). This trend is also observed for the rewritten queries. For *q1*, the number of tuples in the result set of the rewritten queries varies from 10^6 to $4 * 10^5$, which justifies the different execution time in Figure 7. For *q2*, the number of tuples that are aggregated by the rewritten queries varies from 10^6 to $4 * 10^5$. This justifies shorter execution time for higher selectivity values. However, since the result set only includes one tuple, the execution time range of the rewritten queries is narrower.

The considerations on the effect of the selectivity in the presence of filtering and join expressions on the measured complexity are confirmed also wrt the execution time overhead. The need to access less tuples and to return smaller results set impacts the execution time. For instance, with selectivity 0.2, 0.4 and 0.6, the execution time of the rewritten versions of *q5 q6* and *q7* is lower than the one of the original query. However, as shown in Figure 7, the execution time overhead never affects system usability. In contrast, in the presence of filtering expressions, the policy selectivity magnifies the filtering effect, decreasing the accesses, and, as a consequence, the execution time. Based on these observations, we believe that the proposed encoding and enforcement criteria features very good performance.

Experiment 2: In this experiment we assess the enforcement overhead when varying the size of the dataset considered. We consider four distinct scenarios, where we vary the number of samples which are sensed for each patient. More precisely, in scenario 1 (*Scn 1*) *sensed_data* includes 10^4 tuples, whereas in *Scn 2*, *Scn 3*, and *Scn 4*, *sensed_data* contains 10^5, 10^6 and 10^7 rows, respectively. In all scenarios, tables *users* and *nutritional_profiles* store 1.000 tuples each. For this experiment, policies selectivity is set to 0.4, whereas each policy contains from 1 to 3 rules.Policy specification uses the same approach used in Experiment 1.

The diagram in Figure 7 compares the execution time of the original and rewritten version of queries *q1-q8* and *r1-r20* in all considered scenarios. The execution time trend of the original queries is similar in all scenarios. The same situation is also verified with the rewritten queries. Variations among scenarios are due to the different size of the involved datasets which affect the filtering effect of queries and thus the quantity of data that are actually processed. The differences between the absolute value of the execution time of rewritten and original queries are weakened in *Scn 1*, as, due to the small size of the dataset, the volume of data that are filtered out and not processed by the rewritten queries, is quite limited. These differences are more emphasized with the growth of the dataset size (the diagram in Figure 7 uses a logarithmic scale). However, the trend is similar in all scenarios. The experiment results show a very good scalability of the approach, with comparable performance measured with datasets of different size.

7 Related work

The research area on purpose based access control includes only few proposals targeting relational DBMSs. Agrawal et al. [2] in their seminal work discussed high level development strategies of DBMS components charged to monitor DBMS activities based on privacy policies. Our work is aligned with [2] wrt the key role of purposes, and some enforcement strategies. Byun and Li [3] propose a purpose and role based access control model for relational DBMSs. The work also proposes a query rewriting approach to enforce purpose based access control policies. Kabir and Wang [4] propose a conditional purpose based access control model (CPBAC) which extends [3] with conditional purposes. In [9], Kabir et al. propose the Role-involved Purpose-based Access Control (RPAC), an extended version of CPBAC that integrates concepts from RBAC. Ni et al. [10] propose a family of models called Conditional Privacy-aware Role Based Access Control (P-RBAC), which extend RBAC integrating concepts like purposes and obligations allowing the specification of privacy policies. Peng et al. [11] propose a purpose based access control model that extends RBAC. The characterizing feature of [11] is the dynamic association of access purposes to user queries based on system and user attributes. Our work differs from [3], [4], [9], [10], [11] for the access control model, as none of them is action aware.

In [5], we proposed a framework for the automatic generation of enforcement monitors for purpose and role based privacy policies and their integration into DBMSs. In [6] we have extended the framework to support policies also including obligations. However, the access control models in [5] and [6] do not support action aware policies.

Although not directly related, other work in the area of privacy-aware access control propose logic-based approaches to the specification and enforcement of privacy policies which also consider purposes. The logic framework proposed by Datta et al. [12] also allows to check audit logs for compliance with privacy policies. DeYoung et al. use the framework in [12] to formalize some US privacy laws [13]. Jafari et al. [7] propose a model that formalizes the concept of purpose and its relation to system actions, and a model checking algorithm that checks the system compliance with the specified policies. Other formal frameworks (e.g., [14]) support privacy requirement specification and provide mechanisms to check system correctness with respect to these requirements. However, different from our framework, these formalisms are privacy oriented and do not enforce policies at SQL query execution time.

Finally, some work in the literature (e.g., [15] and [16]) propose language based policy specification and enforcement frameworks. However these approaches target applications under development, separating the programming of functional aspects from privacy concerns. In contrast, our framework aims at complementing existing relational DBMSs with data protection capabilities.

8 Conclusions

This paper presented a framework to integrate *action aware* purpose based access control into DBMSs. Our framework allows regulating the access to data performed by SQL queries based on: the access purposes of the query to be executed, the types of actions that the query should execute on data, and the categories of the data jointly accessed during the execution. The framework supports policy specification and enforcement. It has been defined to minimize policy enforcement overhead. The enforcement is achieved through query rewriting. As future work we plan to: 1) build a toolkit supporting the integration of the proposed framework into different DBMSs, 2) extensively evaluate performance and dependability applying the framework to realistic case studies, 3) extend the framework integrating support for role based access control, 4) integrate mechanisms to regulate the specification of data categories and policies and to manage policy, data and category updates.

References

[1] E. Ferrari, *Access Control in Data Management Systems*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[3] J. Byun and N. Li, "Purpose based access control for privacy protection in relational database systems," *The VLDB Journal*, vol. 17, no. 4, 2008.

[4] M. E. Kabir and H. Wang, "Conditional purpose based access control model for privacy protection," in *Proceedings of the Twentieth Australasian Conference on Australasian Database-Volume 92*. Australian Computer Society, Inc., 2009, pp. 135–142.

[5] P. Colombo and E. Ferrari, "Enforcement of purpose based access control within relational database management systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 11, 2014.

[6] ——, "Enforcing obligations within relational database management systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 11, no. 4, 2014.

[7] M. Jafari, P. W. Fong, R. Safavi-Naini, K. Barker, and N. P. Sheppard, "Towards defining semantic foundations for purpose-based privacy policies," in *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '11. New York, NY, USA: ACM, 2011, pp. 213–224.

[8] L. Sweeney, "k-anonimity: a model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.

[9] M. Kabir, H. Wang, and E. Bertino, "A role-involved conditional purpose-based access control model," in *E-Government, E-Services and Global Processes*, ser. IFIP Advances in Information and Communication Technology, M. Janssen, W. Lamersdorf, J. Pries-Heje, and M. Rosemann, Eds. Springer Berlin Heidelberg, 2010, vol. 334, pp. 167–180.

[10] Q. Ni, E. Bertino, J. Lobo, C. Brodie, C.-M. Karat, J. Karat, and A. Trombeta, "Privacy-aware role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 3, p. 24, 2010.

[11] H. Peng, J. Gu, and X. Ye, "Dynamic purpose-based access control," in *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, Dec 2008, pp. 695–700.

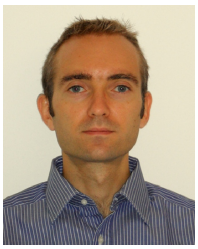
[12] A. Datta, J. Blocki, N. Christin, H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Sinha, "Understanding and protecting privacy: Formal semantics and principled audit mechanisms," in *Proceedings of the 7th International Conference on Information Systems Security*, ser. ICISS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–27.

[13] H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Datta, "Experiences in the logical specification of the hipaa and glba privacy laws," in *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society*, ser. WPES '10. New York, NY, USA: ACM, 2010, pp. 73–82.

[14] F. Massacci, J. Mylopoulos, and N. Zannone, "From hippocratic databases to secure tropos: a computer-aided re-engineering approach," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 02, 2007.

[15] K. Hayati and M. Abadi, "Language-based enforcement of privacy policies," in *Proceedings of the 4th International Conference on Privacy Enhancing Technologies*, ser. PET'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 302–313.

[16] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 85–96, 2012.



Pietro Colombo received the PhD in Computer Science in 2009 from the University of Insubria (Italy). Currently, he works as research associate within the STRICT SociaLab investigating the definition of privacy-aware data management systems. His research interests are mainly related to data privacy and model driven engineering.

Elena Ferrari is a full professor of Computer Science at the University of Insubria, Italy and scientific director of the K&SM Research Center. Her research activities are related to access control, privacy and trust. In 2009, she received the IEEE Computer Society's Technical Achievement Award for "outstanding and innovative contributions to secure data management". She is an IEEE fellow and an ACM Distinguished Scientist.