Enforcing Agile Access Control Policies in Relational Databases using Views

Nicolas Papernot, Patrick McDaniel, and Robert J. Walls
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{ngp5056,mcdaniel,rjwalls}@cse.psu.edu

***Abstract*—Access control is used in databases to prevent unauthorized retrieval and tampering of stored data, as defined by policies. Various policy models provide different protections and guarantees against illegal accesses, but none is able to offer a universal fit for all access control needs. Therefore, the static nature of access control mechanisms deployed in commercial databases limit the security guarantees provided. They require time-consuming and error-prone efforts to adapt access control policies to evolving security contexts. In contrast, we propose a fully automated and agile approach to access control enforcement in relational databases. We present tractable algorithms that enforce any policy expressible using the high-level syntax of the Authorization Specification Language. This includes complex policies involving information flow control or user history dependencies. Our method does not require any modification to the database schema or user queries, thus allowing for a transparent implementation in existing systems. We demonstrate our findings by formulating two classic access control models: the Bell-LaPadula model and the Chinese Wall policy.**

I. Introduction

A. Databases and Access Control

Relational databases are widely used to provide structured storage of information. They often store valuable and sensitive data like personal records, intelligence information, or credit card numbers. Databases are thus the target of many attacks: 7 of the top 10 security threats published by OWASP involve data protection in databases [2]. To reduce the attack surface, database management systems deploy data protection mechanisms. Once users are authenticated, *access control mechanisms* analyze queries over the database to preserve confidentiality and integrity properties. *Confidentiality* prevents unauthorized access to data. *Integrity* ensures data is not improperly modified. These two properties are typically expressed as a set of policy rules defining when accesses are secure i.e. whether or not a user is authorized to execute a specific operation on the database.

The current workflow for implementing access control in a commercial database like Oracle MySQL or Microsoft SQL Server is as follows: (1) an access control policy is defined by a competent individual in the organization, (2) a database administrator is responsible for correctly applying the set of rules forming the policy to create a matching set of privileges for users, and (3) the access control mechanisms provided by the database management systems enforce these user privileges. This workflow is both time-consuming for the administrator and error-prone. This is due to the gap between policy models and access control mechanisms. Furthermore, traditional mechanisms cannot enforce policies involving knowledge about a user's past accesses. We address these issues by making the following contributions:

- We propose an automated method for translating high-level database policies into access control mechanisms.
- We design algorithms supporting various policies like static access control, information flow, or mutual exclusion. Thus, we provide the foundation for agile enforcement of access control policies.
- We develop tractable algorithms supporting `SELECT`, `INSERT`, `DELETE`, and `UPDATE` SQL queries.
- Our method does not require any modifications of the database schema or user queries.

This paper describes our initial efforts and is part of ongoing research efforts.

B. Multilevel Security for Relational Databases

Multilevel security (MLS) is an example of policy frequently used with access control mechanisms. MLS illustrates the dynamics and complexity of access-control. It is important to military applications because it implements the need-to-know principle: a user has access to information only if required. MLS is based on a model introduced in 1973 by Bell and LaPadula [3] wherein objects and users are labeled using a two dimensional lattice made of a sensitivity level and a set of compartments. The ordered set of *sensitivity levels* is: Unclassified (U), Confidential (C), Secret (S), and Top Secret (TS). *Compartments* include Nuclear, Europe, Cyberdefense, etc. Figure 1 is an example MLS-labeled table.

id	name	mission	destination
-	*U*,{`Naval`}	*TS*,{`Naval`}	*S*,{`Naval`}
1	Seawolf	spy	Russia
2	Roosevelt	patrol	Gulf of Aden
3	Normandy	patrol	Gulf of Oman

Fig. 1. MLS table `ships`: first row corresponds to column names and second row to security levels. "-" symbol indicate unlabelled columns (not monitored by access control mechanisms).

Users are given *clearances* for some labels which limits their access to controlled data. Access control is enforced

978-1-5090-0073-9/15/$31.00 ©2015 IEEE

though two properties: the simple security property and the *-property. To read object o labeled (s_o, c_o), the simple security property requires that user u possess clearance (s_u, c_u) such that $(s_u \geq s_o)$ & $(c_o \subset c_u)$ where s denotes the sensitivity component and c the compartments component of a label. The property is also known as no read up. Similarly, the *-property requires that $(s_u \leq s_o)$ & $(c_o \subset c_u)$ for user u to write object u. The property is also known as no write down.

q_1	`SELECT id,name FROM ships`
q_2	`SELECT * FROM ships`
q_3	`INSERT INTO ships (name) VALUES (enterprise)`
q_4	`INSERT INTO ships (id,mission, destination) VALUES (5,spy, China sea)`
q_5	`UPDATE ships SET destination=Yemen WHERE mission=spy`

Fig. 2. Example queries on table `ships`. We show why a user u with clearance label $(S, \{\texttt{Naval}\})$ can execute q_1, and q_4 but not q_2, q_3, and q_5.

Let us now analyze how the Bell-LaPadula model is enforced on queries stated in Figure 2. We suppose user u is given clearance label $(S, \{\texttt{Naval}\})$. Query q_1 reads columns `id` and `name`. The `id` column is unlabeled, so mechanisms do not control it. The `name` column's sensitivity label is smaller than the user's clearance (`U`) and user has access to the `Naval` compartment. Thus, the simple security property is satisfied: query q_1 can be executed. However, q_2 violates the simple security property because the `mission` column is labeled `TS`, which is above u's sensitivity clearance. Similarly, q_3 violates the *-property while on the contrary q_4 satisfies it. Finally, u cannot execute q_5 since he can write the `destination` column (according to the *-property) but cannot read the `mission` column (according to the simple security property).

Such access control properties can be enforced in current database management systems by specifying static user privileges on tables and columns. But if the policy is modified, updating privileges to take into account changes made can be both complex and error-prone. Furthermore, this static approach to access control cannot enforce policies depending on previous user requests. For instance, if mutual exclusion is required between two tables in order to prevent conflicts of interest, access control must consider user history while analyzing requests. This highlights the need for an approach where user access rights can dynamically evolve over time.

C. Our approach

Instead of assuming that administrators express access control policies using low-level mechanisms provided by the database management system, such as user privileges on tables, we propose the following: administrator write policies in a high-level language whose syntax is close to English. This is both faster and less error-prone. The remainder of the process is automated and *guarantees* enforcement of the policy. To achieve this goal, our approach is based on the *view* mechanism. Views are tables defined in terms of queries over other tables. Thus, views can be used to limit access to data stored in the database by providing a restricted version of tables.

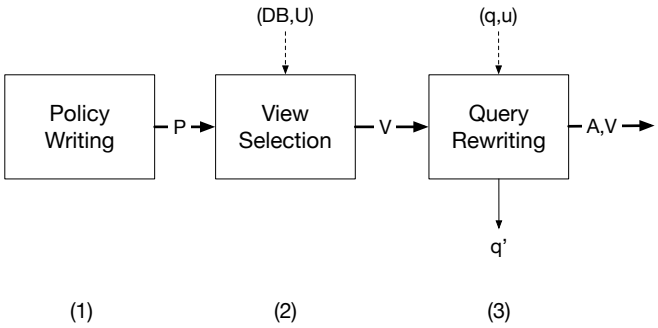

Fig. 3. Overview: (1) the administrator writes policy set P, (2) the view selection algorithm generates a satisfying view set V, and finally (3) the query rewriting algorithm rewrites each query with views in V and keeps a record of past accesses in set A.

Our approach (cf. Figure 3) leverages three successive phases: *policy writing*, *view selection*, and *query rewriting*.

1) In the *policy writing* phase described in section II, a database administrator states access control policy P using a formalism which we introduce later.
2) Before users start querying the database, views of each table are generated by the *view selection* algorithm presented in section III. The resulting view set V corresponds to authorizations defined by the access control policy P written in the previous phase.
3) The *query rewriting* phase begins as users query the database. As described in section IV, each time a user u issues a query q, we use the user's views on the database to rewrite it. The user is provided with a query q' defined over views in V. Query q' is guaranteed to be compliant with policy P. Throughout the process, view set V is updated to ensure compliance with policy P.

We state formally the view selection and query rewriting problems as:

a) View Selection: Given a database schema DB, a policy set P, and a set of user U, generate a set V of views that satisfies policy P.

b) Query Rewriting: Given a database schema DB, a policy set P, a set of users U, a satisfying view set V, a set of past user accesses A, for a query q defined over DB, rewrite query q' over views in V and update V to maintain policy compliance.

II. Policy Writing

A. Modeling Access Control Policies

In addition to Bell and LaPadula's confidentiality model [3], Biba designed the dual model for integrity using integrity labels [5]. Introduced later, Role Based Access Control simplifies permission management [19].

Various languages provide a specific formalism tailored to security policies [18], [21], [16]. For instance, the Ponder

Specification Language, provides a formalism to express static access control rules [7]. We use the Authorization Specification Language (ASL) [11] as its formalism allows us to consider policies involving user history.

We seek to demonstrate the expressiveness of our methodology in terms of the complex policy models it can enforce. For instance, ASL concisely expresses the SeaView [8], Bell-LaPadula, or Chinese Wall policies [6].

B. Authorization Specification Language

The Authorization Specification Language (ASL) was proposed by Jajodia et al. [11]. ASL Policies are sets of predicates and rules. A *predicate* is written using the following arguments: subjects s (users), objects o (tables or columns), and actions a (SQL commands `SELECT`, `INSERT`, `DELETE`, and `UPDATE`). Here is a list of possible predicates:

- $\texttt{active}(s, r)$: specifies that role r is active for user s
- $\texttt{in}(s, s')$: expresses direct membership of s in s'. Direct membership is explicitly stated.
- $\texttt{dirin}(s, s')$: expresses indirect membership of s in s'. Indirect membership is inherited.
- $\texttt{typeof}(o, t)$: specifies type t of object o

Each *rule* has the format: $head \leftarrow body$. The syntax used to write *head* is `type(arguments)` where `arguments` are identical to those previously described for predicates. The *body* of a rule is a set of literals. A *literal* is a predicate, a rule, or the negation of a predicate or rule. Rules expressed in ASL are true if and only if the conjunction of their body literals are true. The possible rule types are:

- *done*: $\texttt{done}(o, s, R, a, t)$ represent past accesses made by subjects. Their body is empty. They form set A.
- *authorization* : $\texttt{cando}(o, s, +a)$ allows accesses to objects. Conditions on subjects and objects can be specified using `in`, `dirin`, or `typeof` literals.
- *derivation*: $\texttt{dercando}(o, s, +a)$ expresses propagation of authorizations. The body can be made of `cando`, `dercando`, `done`, `in`, `dirin`, or `typeof` literals.
- *access control*: $\texttt{grant}(o, s, r, +a)$ describe broader rules than specific authorization rules of type `cando` or `dercando`. They are made of `cando`, `dercando`, `done`, `do`, `in`, `dirin`, or `typeof` literals.
- *resolution*: $\texttt{do}(o, s, +a)$ solves potential conflicts between rules of type `cando` or `dercando`.
- *integrity*: `error()` rules are made of `grant`, `cando`, `dercando`, `done`, `do`, `in`, `dirin`, or `typeof` literals. When a new rule is inserted, integrity rules should evaluate to false.

The only rules added to policy set P during execution are past accesses `done`. We make the following assumptions on policy set P:

- Policies are defined at column granularity: objects can be any table $T \in DB$ or any column $T.c$ of a table T.
- Authorization and derivation rules are only used to write positive authorizations. Considering negative authorizations would require conflict resolution [14].

Require: L, DB, U
1: Let $P = \emptyset$
2: **for** each $T \in DB$ **do**
3: **for** each $u \in U$ **do**
4: **if** $(l_{T,1} \leq l_{u,1})$ & ... & $(l_{T,n} \leq l_{u,n})$ **then**
5: add rule $\texttt{grant}(T, u, R, SELECT) \leftarrow$
6: $\texttt{active}(u, l_u)$ & $\texttt{typeof}(T, l_T)$
7: **else if** $(l_{T,1} \geq l_{u,1})$ & ... & $(l_{T,n} \geq l_{u,n})$ **then**
8: add rule $\texttt{grant}(T, u, R, INSERT) \leftarrow$
9: $\texttt{active}(u, l_u)$ & $\texttt{typeof}(T, l_T)$
10: **else if** $(l_{T,1} = l_{u,1})$ & ... & $(l_{T,n} = l_{u,n})$ **then**
11: add rule $\texttt{grant}(T, u, R, UPDATE) \leftarrow$
12: $\texttt{active}(u, l_u)$ & $\texttt{typeof}(T, l_T)$
13: add rule $\texttt{grant}(T, u, R, DELETE) \leftarrow$
14: $\texttt{active}(u, l_u)$ & $\texttt{typeof}(T, l_T)$
15: **end if**
16: **end for**
17: **end for**
18: **return** P

Fig. 4. Generating a policy set for the n-dimensional Bell-LaPadula model.

- Policy rules have a finite number of literals.
- Rules whose literals are defined recursively (e.g., `dercando`) will have a finite number of recursive rules.

C. The Multilevel Security example

We here introduce a generalized n-dimensional lattice and provide an algorithm illustrating the construction of a policy set P corresponding to the Bell-LaPadula model for set of users U and tables DB. Set P can then be used as an input for our view selection and query rewriting algorithms. A label is a n-component vector $(x_1, ..., x_n)$. Each component x_i corresponds to a lattice dimension and is an element from an ordered set X_i. Thus, the set of labels $L = X_1 \times ... \times X_n$ is a partially ordered set and models any n-dimensional lattice.

We now seek to apply the Bell-LaPadula model to lattice L. The simple security property for our lattice L states that a user u labeled l_u can read a table T labeled l_T if:

$$\forall i \in 1..n, l_u[i] \geq l_T[i]$$

Similarly, the *-property states that a user u labeled l_u can write to a table T labeled l_T: $\forall i \in 1..n, l_u[i] \leq l_T[i]$. Using these two conditions, the algorithm in Figure 4 generates a set of ASL `grant` rules which correspond to the Bell-LaPadula model for lattice L.

The algorithm's termination follows from the finite number of users and objects. We now prove its correctness by contradiction. Let's suppose that according to policy set P generated by the algorithm, user u can `SELECT` table T, although the operation violates the simple security property. A user u is given the right to `SELECT` table T if and only if there exists a `grant` rule with parameters $(T, u, *, \texttt{SELECT})$ which evaluates to true. By construction, the `grant` rule evaluates to true if and only if $\texttt{typeof}(T, l_T)$ and $\texttt{active}(u, l_u)$ are simultaneously true. In other words, $l_T \in L$ is the object's

label and $l_u \in L$ is the user's label. In addition, the algorithm only generates such `grant` rules when $\forall i \in 1..n, l_u[i] \geq l_T[i]$. This means that the simple security property is verified, which contradicts our proof hypothesis: no violation occurred. By repeating similar analysis for other SQL commands, we conclude the proof of correctness for this algorithm.

D. The Chinese Wall policy example

The Chinese Wall is used to model mutual exclusion and is often used to model access control policies in legal and financial firms. Let us consider two competing companies, A and B, both represented by a law firm. In its database DB, the law firm stores tables holding information for A: namely $A1$ and $A2$. These tables are labelled using ASL rule $\texttt{typeof}(T, A)$. They also use database DB to store some tables regarding B: namely $B1$ and $B2$. They are labelled with $\texttt{typeof}(T, B)$. Once an law firm employee starts representing company A, he cannot represent company B. Consequently, he cannot access tables $B1$ and $B2$. The scenario is similar when an employee first represents B. This policy can be expressed using the following two ASL rules:

- $p_1 = \texttt{grant}(T, u, r, *) \leftarrow !done(T', u, *, *, *)$ `&` $\texttt{typeof}(T, A)$ `&` $\texttt{typeof}(T', B)$
- $p_2 = \texttt{grant}(T, u, r, *) \leftarrow !done(T', u, *, *, *)$ `&` $\texttt{typeof}(T, B)$ `&` $\texttt{typeof}(T', A)$

We have demonstrated the construction of policy set P. The view selection phase described hereafter uses this set to construct a view set V corresponding to the policy.

III. View Selection

Once policy set P is written using ASL, view selection can start and the remainder of our approach is automated. View selection completes before users start querying the database. Each user $u \in U$ is given a set V_u of views to which he has access. A set V_u contains 4 views of each database table $T \in DB$, each corresponding to one of the SQL command that can be used to query the database: $v_s(u, T) \in V_{u,s}$ for `SELECT`, $v_i(u, T) \in V_{u,i}$ for `INSERT`, $v_d(u, T) \in V_{u,d}$ for `DELETE`, and $v_u(u, T) \in V_{u,u}$ for `UPDATE`. A view in $V_{u,a}$ must only return data that the user has legitimate access to using SQL command a. Having a one-to-one correspondance between SQL operations and views for each table facilitates the query rewriting process. Moreover, views defined over one table using selection and projection have a key property: they are *updatable views* if we assume that all view columns have a default value [1]. If policy set P includes rules dependent of previous queries, some views in V will be limited progressively as the user queries the database. We name V the set of views generated once the view selection process has been iterated for each user. Figure 5 illustrates the overall structure of set V.

The view selection algorithm implements the following vs function: $V = vs(DB, P, U)$. For each pair $(u, T) \in U \times DB$, the algorithm in Figure 6 creates the 4 previously described views $v_s, v_i, v_d, v_u \in V_{u,s} \times V_{u,i} \times V_{u,d} \times V_{u,u}$ corresponding to the subset of table columns that user u can manipulate

Require: DB, U, P
1: $V = V_1 \cup ... \cup V_{|U|} = \emptyset$
2: **for** each $u \in U$ **do**
3: **for** each $a \in \{\texttt{SELECT}, \texttt{INSERT}, \texttt{DELETE}, \texttt{UPDATE}\}$ **do**
4: Create empty set $V_{u,a}$
5: **for** each table $T \in DB$ **do**
6: Compute $G = G(u, T, *, a)$
7: Compute $C = C(u, T, *, a)$
8: Compute $D = D(u, T, *, a)$
9: **if** $G \cup C \cup D = \emptyset$ **then**
10: $columns = \emptyset$
11: **for** each column $c \in T$ **do**
12: Compute $G_c = G(u, T, c, a)$
13: Compute $C_c = C(u, T, c, a)$
14: Compute $D_c = D(u, T, c, a)$
15: **if** $(G_c \cup C_c \cup D_c \neq \emptyset)$ **then**
16: $columns = columns \cup \{c\}$
17: **end if**
18: **end for**
19: $v_a(u, T)$ = `SELECT` $columns$ `FROM` T
20: **else if** $(G \cup C \cup D \neq \emptyset)$ **then**
21: create $v_a(u, T)$ = `SELECT * FROM T`
22: **end if**
23: $V_{u,a} = V_{u,a} \cup \{v\}$
24: **end for**
25: **end for**
26: $V_u = V_{u,s} \cup V_{u,i} \cup V_{u,d} \cup V_{u,u}$
27: **end for**
28: **return** V

Fig. 6. ASL View Selection algorithm

using respectively `SELECT`, `INSERT`, `DELETE`, and `UPDATE` queries. To evaluate the access rights of user u, `grant`, `cando`, and `dercando` rules are considered. They give user u full or partial access to table T using each of the 4 SQL command. The following sets of rules are defined:

$$G(u, T, c, a) = \{g = \texttt{grant}(t.c, s, R, a) \in P | u \in \texttt{users}(g)\}$$
$$C(u, T, c, a) = \{c = \texttt{cando}(t.c, s, a) \in P | u \in \texttt{users}(g)\}$$
$$D(u, T, c, a) = \{d = \texttt{dercando}(t.c, s, a) \in P | u \in \texttt{users}(g)\}$$

where $\texttt{users}(g)$ returns the set of users to which rule g applies. We detail subroutine `users` in future work. For instance, $G(u, T, c, a)$ is the set of `grant` rules which apply to user $u \in U$, table column $T.c \in DB$, and SQL command a. Rules apply to the whole table T if a $*$ character is used instead of a column c. The user is given access to the whole table when at least one of the sets $G(u, T, *, a), C(u, T, *, a), D(u, T, *, a)$ is non-empty. If the condition is unsatisfied, a similar one is tested for each column of table T. Finally, a view v of table T is generated accordingly.

Termination of the algorithm follows from the finite number of tables and users. The view selection algorithm is correct if for any user $u \in U$, the set $V_u \subset V$ of views he/she can access is compliant with policy set P. The view selection algorithm

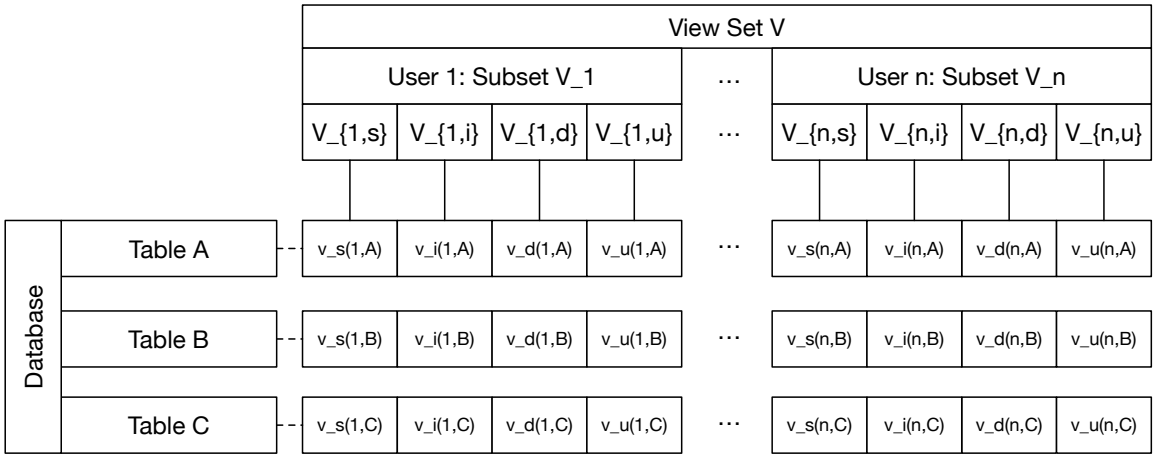

Fig. 5. The view set V structure is detailed for two users: 1 and n. Each user is given 4 sets: one corresponds to the access control for `SELECT` queries, one for `INSERT`, one for `DELETE`, and the last one for `UPDATE`. Thus, for each user, 4 views of each of the 3 tables A, B, and C are created.

is run before users query the database so there are no past accesses. Hence, the algorithm is correct if and only if each set V_u is compliant with `grant`, `cando`, and `dercando`, and `error` rules applying to user u. We will provide formal proofs in the full version of this paper.

IV. Query Rewriting

The query rewriting phase rewrites queries over tables into equivalent queries over views in V and compliant with policy P. However, this is not always possible as users may not have the required access rights to find strictly equivalent queries. The algorithm implements the following qr function with $q \in Q$ defined over tables and q' defined over V: $\{q', V', A'\} = qr(DB, P, A, V, u, q)$. Set V' is an updated version of V taking into account eventual modifications to access rights due to rules depending on past accesses. Set A' is an updated version of A including new accesses made in the rewritten query. The algorithm accepts all queries written using SQL commands: `SELECT`, `INSERT`, `DELETE`, and `UPDATE`.

The query rewriting algorithm must fulfill 2 objectives. First, it must rewrite a query q using views in V to ensure compliance with P or return an error if it is not possible. Second, it must restrict set V if required by policy once q has been executed. The algorithm in figure 7 parses query q and replaces each table column $T.c$ by a view column $v.c$ such that $v \in V_{u,a}$ and $T.c \subset v$, where a is the SQL command of q. Each time a column $T.c$ is rewritten and corresponding view $v.c$ is not empty, a rule of type `done` is added to A' to keep track of the access made by the user. The view set is then updated by looking for all views in $V_{u,a}$ built using a policy rule conflicting with set A'. This search is done using subroutine `find-done`, which we will present in future work. For each of these rules, the conflicting table columns are identified and removed from the view.

The algorithm terminates because there is a finite number of columns in a query and subroutine `find-done` terminates (proof in future work). We have previously proven in the view selection algorithm analysis that set V initially corresponds to policy set P. We can deduce from this fact that as long as the query is rewritten solely using views in V (which is the case here), it will be compliant with P. Future work will provide a formal proof of security based on this observation.

V. Related Work

A. Enforcing Policies using Views

Bender et al. enforce policies using view selection and query rewriting [4]. However, they suppose that views are manually generated by an administrator, whereas our view selection phase automates this task. Furthermore, they restrict queries to a subclass of conjunctive queries while our algorithm can be used with any SQL query.

Similarly, Motro assumes the user is given views and provides query rewriting [15]. Queries are rewritten using views of the user's views. This work bears the same limitations: the view set is assumed to be given as an input, and the method is only valid for static access control.

Rizvi et al. apply query rewriting to provide fine-grained access control for databases [17]. They assume an administrator defines *parameterized views*: views customized using variables taking different values for each user. They introduce inference rules to test whether queries can be equivalently rewritten using views. However, they do not prove the completeness of their rules for any class of queries. Moreover, parameterized views are still limited to static access control.

B. View Selection and Query Rewriting

Mami et al. define view selection and survey various solutions [13]. Algorithms discussed include deterministic [9] and randomized [22] approaches. All solutions seek to reduce some combination of query processing, maintenance, or storage costs. In contrast, our view selection algorithm ensures policy compliance.

Halevy et al. survey query rewriting algorithms and partition them according to their end goal: data integration or query

Require: DB, P, A, V, u, q
1: $q' = q$
2: $A' = \emptyset$
3: let a be the operation of q
4: **for** each table column $T.c$ mentioned in q **do**
5: **if** $\exists v \in V_{u,a}, T.c \subset v$ **then**
6: rewrite q' with $v.c$ instead of $T.c$
7: **if** $v.c \neq \emptyset$ **then**
8: $A' \leftarrow A' \cup \texttt{done}(T.c, u, R, a, t)$
9: **end if**
10: **else**
11: **return** "query not compliant"
12: **end if**
13: **end for**
14: **for** each $d \in A'$ **do**
15: $\{p_j\}_j = \texttt{find-done}(P \cup A \cup A', d)$
16: **for** each p_j **do**
17: **if** $p_j == \texttt{error}()$ **then**
18: **return** "query not compliant"
19: **else if** $p_j == \texttt{dercando}(T.c, s, +a)$ **then**
20: empty c from $v \in V_{u,a}$ such that $T.c \subset v$
21: **else if** $p_j == \texttt{do}(T.c, s, +a)$ **then**
22: empty c from $v \in V_{u,a}$ such that $T.c \subset v$
23: **else if** $p_j == \texttt{grant}(T.c, s, r, +a)$ **then**
24: empty c from $v \in V_{u,a}$ such that $T.c \subset v$
25: **end if**
26: **end for**
27: **end for**
28: **return** $\{q'$,V',$A' \cup A\}$

Fig. 7. ASL Query Rewriting algorithm

optimization [10]. *Data integration* uses views to provide a uniform query interface to different data sources [12]. *Query optimization* uses materialized views to improve query processing [20]. Our solution differs in the construction of the view set V. Further, the complexity of our algorithm does not reside in query rewriting but instead in view management.

VI. Conclusion

We presented tractable algorithms capable of enforcing any access control policy expressible using the ASL language. We obtain strong results valid for all SQL queries using the major commands: `SELECT`, `INSERT`, `DELETE`, and `UPDATE`. The approach does not require modification of the database schema or user queries. This work lays foundations for agile enforcement of policies in relational databases. A full version of the paper will provide full asymptotic analysis and security proofs. Next steps include implementing this technique in a real-world database to study its impact on performance and developing optimal ways to modify policy set P on the fly without running the full view selection process.

Acknowledgment

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Mysql 5.0 reference manual - updatable and insertable views. http://dev.mysql.com/doc/refman/5.0/en/view-updatability.html.
[2] Owasp top 10. https://www.owasp.org/index.php/Top_10_2013-Top_10.
[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.
[4] G. M. Bender, L. Kot, J. Gehrke, and C. Koch. Fine-grained disclosure control for app ecosystems. In *Proceedings of the 2013 international conference on Management of data*, pages 869–880. ACM, 2013.
[5] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.
[6] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214. IEEE, 1989.
[7] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001.
[8] D. E. Denning, T. F. Lunt, R. R. Schell, W. R. Shockley, and M. Heckman. The seaview security model. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 218–233. IEEE, 1988.
[9] H. Gupta. Selection of views to materialize in a data warehouse. In *Database Theory ICDT'97*, pages 98–112. Springer, 1997.
[10] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.
[11] S. Jajodia, P. Samarati, and V. Subrahmanian. A logical language for expressing authorizations. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 31–42. IEEE, 1997.
[12] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. 1996.
[13] I. Mami and Z. Bellahsene. A survey of view selection methods. *ACM SIGMOD Record*, 41(1):20–29, 2012.
[14] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. *ACM Transactions on Information and System Security (TISSEC)*, 9(3):259–291, 2006.
[15] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *Data Engineering, 1989. Proceedings. Fifth International Conference on*, pages 339–347. IEEE, 1989.
[16] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.
[17] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM, 2004.
[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.
[19] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
[20] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The gmap: A versatile tool for physical data independence. *The VLDB Journal*, 5(2):101–118, 1996.
[21] L. F. P. Valente, G. H. Cooper, R. A. Shaw, and K. G. Sherlock. Declarative language for specifying a security policy, Aug. 17 2004. US Patent 6,779,120.
[22] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 33(4):458–467, 2003.