A Policy Management Tool for Fine-Grained Database Access Control
Ninghui Li (Purdue), Peng Liu (Penn State) and Ting Yu (NC State)

1. Introduction

Database access control plays a central role in ensuring authorized access to sensitive information. As cross-organization information sharing becomes a must for the collaboration among corporations and governmental agencies, table-level or column-level database access control is no longer adequate. View-based approaches, on the other hand, require separate views for potentially large number of entities, which is not only costly, but also associated with consistency and integrity problems. Recently, row-level access control was proposed as an effective solution. Many commercial DBMSs have supported row-level access control. And they are widely employed in critical commercial and governmental information systems.

However, row-level access control in commercial DBMSs is implementation-oriented. They only offer mechanisms to enforce row-level access control policies. But how to manage such complex policies is largely ignored. It thus requires policies to be manually coded into applications by programmers, which is a highly error-prone process. Policies may be implemented incorrectly, either due to programmers’ negligence or their misunderstanding of policy requirements. Further, since policies are manually coded, once errors are introduced, they are very hard to detect. Nowadays, a large number of critical information infrastructures (e.g., financial, health-care and telephony) are managed by DBMSs. Incorrect security policy implementation may severely jeopardize homeland security and national interests.

We propose to develop policy management tools to address the above problem. The tool is realized through the following major technical aspects. First is the design of a formal model for row-level access control, so that access control policies can be formally specified and analyzed. Potential security vulnerabilities in policies can be systematically identified and fixed in the formal level. Second, after policies are formally verified, the tool will automatically generate policy enforcement code for database applications, eliminating implementation errors that might be introduced by programmers. Automatic generation of policy enforcement codes also substantially minimizes management overhead when access control policies are revised or updated. Third, the tool will extensively optimize the generated code to ensure no significant negative impact on database performance.

To the best of our knowledge, there are no policy management tools available for fine-grained database access control. Most policy management approaches today focus on policy modeling and specification. There is little effort on ensuring the secure implementation of database security policies.

2. Technical Approach

2.1 Row-level Access Control

Many major commercial DBMSs (e.g., Oracle 8i and later releases, IBM DB2, Microsoft SQL server and Sybase Adaptive Sever Enterprise) support row-level access control. Though named differently, their basic approaches are all based on query-rewriting. In this section, we use Oracle’s Virtual Private Database (VPD) as an example to show how row-level access control is implemented and what vulnerabilities it has.
In Oracle VPD, before executing a user’s query, the DBMS attaches a predicate to the query’s WHERE clause, which further controls which can be accessed by the query. For each relation, a policy procedure has to be defined. This procedure is essentially a program, which, given a user’s context attributes (e.g., login name, role, department, etc.), returns the predicate corresponding to the user’s privilege. In other words, the policy procedure implements an application’s security policy. For example, suppose a policy states that an employee can access his own records, while a manager is allowed to access records of those who are working under him. The policy procedure may look like the following:

```
Create function my_security_function () return varchar2
as begin
    if (sys_context("userenv", "role")= "mgr") then
        return 'emp_mgr = sys_context("userenv", "session_user")';
    elsif (sys_context("userenv", "role")= "emp") then
        return 'emp_name = sys_context("userenv", "session_user")';
    else return '1=0';
    end if;
end;
```

Policy procedures are written by programmers using PL/SQL. Since PL/SQL is a Turing-Complete language, the approach above is very flexible, and can implement arbitrarily complex policies. Due to its flexibility, however, the same policy may be implemented quite differently by different programmers. And if there are errors introduced due to programmers’ negligence or misunderstanding of the policy, it is very hard to catch those errors. Indeed, the code shown above is not correct for most applications, since it overlooked the fact that a manager is usually a senior role to an employee role. A manager should also be able to access his own record, which is not correctly enforced by the above policy procedure.

We argue that depending on programmers to implement security policies cannot provide a high assurance of fine-grained access control. Direct analysis of policy procedures written in PL/SQL also tends to be complex and expensive, especially when there are a large number of policy procedures, which is typical in critical information systems.

2.2 A Policy Management Tool for Row-level Access Control

The goal of the tool is to provide high assurance of the correct specification and implementation of database fine-grained access control, and eliminate potential errors introduced by programmers to the maximum extent. The key idea is to formally model row-level access control policies in relational databases. After policies are verified to be correct, a policy management tool will automatically translate them into implementation codes (e.g., policy procedures in VPD), which can be further imported to databases without the involvement of programmers.

The following figure shows the architecture of the tool.
We next briefly describe each components.

**Design a formal policy model for row-level access control:** We propose the concept of *relation graphs* to represent tables and their logical relationships. Each node in the graph represents the schema of a relation \( R \). An edge from \( R \) to \( R' \), of the form \((R.a_k, R')\), indicates that attribute \( a_k \) of \( R \) is a foreign key to \( R' \). There is also an implicit edge from a relation’s primary key to its other attributes. Intuitively, a path from one attribute \( R_1.a \) to another attribute \( R_2.b \) represents a possible information retrieval method of a particular entity.

A user context \( UC \) contains a user’s attributes relevant to authorization decisions. There are two types of user attributes. *Session attributes* are those assigned by the system when a user starts a session, e.g., user_id and role. *Derived attributes* are those that can be retrieved from the database based on a user’s session attributes. Consider a relation EMP(user_id, rank, mgr_id). A user’s rank and mgr_id are derived attributes since they can be obtained by a select query based on the user’s login ID. Following the edges between relations, a derived attribute can even be obtained from a table not containing the user’s session attributes. Our model also contains a user role hierarchy, which is commonly used in commercial DBMSs.

Each table is attached with constraints in the form (role, condition). For example, if a policy says that an employee can see the records of those working under the same manager, then the constraint will be (employee, mgr_id=\( UC.mgr_id \)).Though \( UC.mgr_id \) is a derived attribute, when specifying constraints, we do not need to distinguish it from session attributes. In the code generation phase, the tool will automatically create queries to retrieve derived attributes.
according to the relation graph. Such an abstraction facilitates the formal definition and analysis of access control policies.

**Development of a policy management toolkit:** Our toolkit first serves as the front end for the design and analysis of security policies. It provides a user-friendly representation of the database and its access control constraints. It also provides intuitive indication when a potential vulnerability is identified. Second, the toolkit is responsible for generating DBMS-specific policy enforcement codes from formal policies, and importing them into DBMS. In general, a session attribute is directly translated to a login user’s environment parameters, which are provided by most DBMSs. A derived attribute, on the other hand, has to be translated to a sub-query. For example, in Oracle VPD, the code for the constraint “mgr_id = UC.mgr_id” should be “mgr_id in select mgr_id from EMP where user_id=sys_context(“userenv”, “session_user”)”. Such translation can be done in a straightforward way by following the relation graph. The toolkit also considers role hierarchies so that the generated code reflects all the privileges of a user. For example, if a user is logged in as a manager, which is a senior role to the employee role, then the toolkit combines the constraints for both roles to get an overall constraint for the user.

**Performance optimization:** A brute force approach to generating policy enforcement code can guarantee its correctness. But, it is usually not as efficient as that written by programmers. Great efforts need to be put on code optimization. For example, some derived attributes of a user are static during a single session. So instead of retrieving such attributes through a sub-query every time the user issues a query, they can be stored in the user’s environment parameters, and only need to be retrieved once at the beginning of a session. How to identify such redundancy is also related to the formal specification and analysis of access control policies.

### 3. Conclusion

Though row-level access control is widely adopted, there is no tool available for the management of complex fine-grained access control policies. The proposed tool not only provides high assurance of correct policy enforcement, but also dramatically reduces the overhead of policy implementation, so that more efforts can be focused on policy specification and verification, which is the most crucial part for database security.