Research of Formal Verification for SQL Operations in Top Level Specification of a Secure Database

Zhipeng Wang, Hong Zhu and Meiyi Xie

Computer School of Sci & Tec, Huazhong University of Sci & Tec,
Wuhan, Hubei, China, 430074
{hkdwzp, whzhuhong}@gmail.com, xiemeiyi@sina.com

Abstract

A high security level DBMS requires a formal specification and verification on the security model and top level specification design. The specification and verification towards SQL operations are important especially. In this paper, based on the security model and top level specification, we propose a novel approach to solve the specification and verification issues towards SQL operations. Firstly, we formally define the SQL operations in FTLS; then, we give the definitions of the simple SQL operations and propose a method to verify those simple SQL operations; finally, we transform the verification of the SQL operations in FTLS to the verification of the component simple SQL operations. The process of verification shows that our approach makes a comprehensive specification of SQL operations and simplifies the verification procedure.

Keywords: *Security Model; Formal Top Level Specification; SQL Operation; Formal Verification*

1. Introduction

When developing a high security level (rated as B2 and above in TCSEC [1], or EAL5 and above in CC [2]) system, formal specification and verification are needed. And as the criterions require, when developing such a high security level DBMS, we need to make formal specification and verification in both Security Model and Formal Top Level Specification (FTLS). A secure DBMS is an extension of the traditional DBMS, e.g. the subjects and objects are bounded with security levels; the definitions of entity integrity and referential integrity are extended, etc. The FTLS of the secure DBMS includes the formal specification and verification of SQL operations, which is important for verifying whether the implementation is consistent with security requirements. Nowadays, SQL statements are more and more complex, and the specification and verification of SQL operations are therefore more difficult. So it is of great significance to propose an approach for the specification and verification of SQL operations in FTLS.

Teresa F.Lunt et al. have done a series of research from security model, FTLS to verification policies in SeaView project [5, 6]. However, because of the database technology limitations, they only researched the simple SQL operations. For example, one can use a *select* statement for create a new table, i.e. a *create* statement can include *select* clauses. They didn’t consider this situation, but only considered the security level of subject and object, the data integrity, etc. in simple *create* statement. Nowadays, there is some research for specification and verification of SQL operations. Li Xu-shuai et al. give a definition of the formal semantics of the SQL query, and propose a way to prove the equivalence of two SQL queries [3]. Li Hai-long et al. establish a model to define the SQL query based on the knowledge of complier construction principles and logic algebra, and verify the SQL query

[4]. The current research has a main problem: there isn't sufficient research on specification and verification of complex SQL operations, which are very common in modern DBMS.

1.1. The Problems

A secure DBMS model is an extension of BLP security model. The FTLS is designed on top of the secure DBMS model. There are following problems in formal specification and verification of SQL operations in FTLS.

Firstly, SQL statements in modern DBMS are more and more complex, which increases the difficulty of verification. For example, a query may include multi-table join, nested subqueries, correlated subqueries, which can access the database and change the database states. We need to record all these accesses in the specification and verification. The complexity of the operations increases the complexity of specification and verification.

Secondly, because FTLS is designed on top of model, a trivial way for its verification is to make a mapping between the model and the FTLS. Unfortunately, the operation rules in FTLS are not corresponding to those in the model. There exists the situation that one rule in the model maps multiple SQL operations. So it is a problem to make a comprehensive and clear specification of the SQL operations in FTLS.

Thirdly, most proof tools and languages, such as Gallina, Z, Isabella, PVS, etc. are not competent for complex structures [7-11]. We take the nested structures of different types as an example. Struct A includes struct B and struct C, and C includes A. This structure is hard for proof tools mentioned above to express for verification. However, it is common in complex SQL statements. A *select* statement may include a *having* clause and a *where* clause, which may also include *select* statements. The limitations of the proof tools make the verification problems more difficult.

1.2. Our Contributions

Focusing on the problems above, our work is to make a comprehensive and clear specification of the SQL operations in FTLS, and make it easier for verification using proof tools. Our contributions are as follows:

- We propose an approach for the specification of SQL operations in FTLS, and give the rules to transform the SQL statements to this specification.
- On top of the specification, we give the definitions of the simple SQL operations, and propose a method to verify those simple SQL operations. Then we transform the verification of the SQL operations in FTLS to the verification of the component simple SQL operations.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the security model of the secure DBMS. In Section 3, we introduce the FTLS and formally specify the SQL operations in FTLS. We also give the rules to transform the SQL statements to those specifications of SQL operations. In Section 4, we verify the SQL operations in details, and discuss some problems in verification. In Section 5, we review the related work of formal specification and verification in SQL and different systems. Finally, in Section 6 we conclude the paper.

2. Security Model for Secure DBMS

Formal security model is the base of the formal specification and verification for a system. BLP (Bell-LaPadula) model [12] is an early designed security model for multi-level security system but still used widely today. According to the features of the secure DBMS, Zhu Hong etc. extended the BLP model for a database system in [13]. They added the integrity constraint of the database, modified some operation rules, etc. and verified the security of the model. It is the basic model of our work. We will briefly introduce this security model in security policy, basic elements, safety properties, state transition rules and security definitions.

2.1. Security Policy

The security policy of a system is a set of strict rules for system behavior according to the security requirements. It decides the safety properties. In secure DBMS, the system assigns each subject with a security level to indicate his/her capability to access information, and each object with a security level to indicate its confidentiality. The security policy in secure DBMS is: information can only flow from low security level objects to high security level subjects. If a subject wants to read from and write to an object, he/she must have the same security level with the object. Because of this security policy, we need to verify all the operation rules to make it sure that there is no information flow from high security level objects to low security level subjects.

2.2. Basic elements

The DBMS is abstracted as a state machine [13]. V represents the set of database states v.

$v = (B, M, F, S, O)$: this five-tuple describes a database state, in which:

S: represents the set of subjects. $s \in S$ is a subject, and it often indicates a session or a user in database.

O: represents the set of objects. $o \in O$ is an object, and it can be a database, a schema, a table, a tuple, a procedure, etc. The minimal granularity of the object for security level is tuple. The hierarchy of the objects is: database, schema, table, tuple make up a tree.

F: represents the set of security level functions. It includes two security level functions: f_c and f_o, which return the security level of a subject and an object respectively.

$X\_OP = \{r, w, e, a, c\}$: represents the set of access types. r represents read-only access, w represents read-write access, e represents execute access, a represents write-only access, and c represents control access.

M: represents the set of access rights. It is the subset of $(S \times O \times X\_OP)$. The element (s, o, x_op) in M means subject s has the x_op access right to the object o.

B: represents the records of accesses. It is the subset of $(S \times O \times X\_OP)$. The element (s, o, x_op) in B means subject s has done the x_op access to the object o.

2.3. Safety Properties

The BLP model defines three security properties: Discretionary-Security Property, Simple-Security Property and Star-Security Property. In addition, according to the secure DBMS features, there are three integrity constraints in the secure DBMS model:

Object-Compatibility Property: a state v satisfies Object-Compatibility Property, if and only if in v, if object o_2 is the father of object o_1, it must satisfy $f_o(o_1) \geq f_o(o_2)$.

It means the security level of an object must dominate the security level of its father. For example, the security levels of the tuples in a table must be equal or greater than the security level of this table. To a subject, if he/she can not access a table because of the low security level, he/she can neither access the tuples in this table.

Entity-Integrity Property: a state *v* satisfies Entity-Integrity Property, if and only if in *v*, for any tuples o_1 and o_2 in an arbitrary table, their primary keys can not be NULL, and either $f_o(o_1) \neq f_o(o_2)$, or their primary keys are different.

Here the security level is added into the traditional entity integrity. Subjects with different security levels can use the same primary key. This can avoid the covert channel caused by the primary key and ensure the safety of the information flow.

Reference-Integrity Property: a state *v* satisfies Reference-Integrity Property, if and only if in *v*, for any tuples o_1 and o_2, if o_1 is referenced by o_2, then either o_2's foreign key is NULL, or its value must be the same with o_1's primary key's value and $f_o(o_1)=f_o(o_2)$.

The Reference-Integrity Property only allows the reference between tuples with the same security level. It is similar to the meaning of Entity-Integrity Property.

2.4. State Transition Rules

There are 10 state transition rules in BLP model to guarantee the safety properties [12]. Based on the features of the secure DBMS, Zhu Hong etc. modified these rules and proposed 10 state transition rules corresponding to *select, update, insert, alter, delete, drop, create, execute, grant, revoke* in database. The details are represented in [13], which you can refer to if you are interested.

2.5. Security Definitions

The security definitions are as follows:

Definition 1. Safe State. A safe state is a state $v \in V$ that satisfies all safety properties.

Definition 2. Safe Operation. If the state v_i is the pre-state of an operation and v_{i+1} is the post-state of the operation, and they are both safe states, then the operation is a safe operation.

Definition 3. Safe System. A safe system is a system in which all the states are safe, which means the initial state of the system v_0 is safe, and if any state v_i that can be transited from v_0 is safe, after arbitrary operation, the post-state v_{i+1} is also safe.

Based on these definitions, Zhu Hong etc. specified and verified the extended DBMS model using COQ tool, and ensured the safety of the model [13].

3. Formal Top Level Specification for Secure DBMS

3.1. Introduction

The Formal Top Level Specification (FTLS) for secure DBMS is specified in formal languages (Gallina, Z, etc.), and consists of three parts: System State, Safety Properties and SQL Operations. The mapping between security model and FTLS is described in Table 1:

Table 1. Mapping between Security Model and FTLS

FTLS		Security Model
System State (*VF*)	Objects Set (*s_ObjectSet*)	Corresponding to *O*
	Data Dictionary (*s_DD*)	Corresponding to *S, M, F*
	User Data (*s_UD*)	No corresponding, for integrity constraint
	Access Set (*s_B*)	Corresponding to *B*
Safety Properties		The same as in security model
SQL Operations		Details of the state transition rules in security model

The security definitions in security model is also suitable for FTLS.

3.2. System State

The system state in FTLS includes the details of the system implementation, e.g. how to store access rights, how to present integrity constraint of the user data, etc. There are four elements in the system state: objects set, data dictionary, user data and access set. As shown in Table 1, *VF = (s_ObjectSet, s_DD, s_UD, s_B*), and is formalized like this (in COQ):

Record State : Set := st {

s_ObjectSet : object_set;

s_DD : DD;

s_UD : UserData;

s_B : (set accessB) }.

The objects set *s_ObjectSet* corresponds to *O* in the security model. *s_DD* is the description of the data dictionary, which stores the access rights (corresponds to *M* in the security model), security levels of subjects and objects (corresponds to *F* in the security model), etc. *s_UD* records the values of the user data, which is used for integrity constraint verification, because the minimal granularity of the object is tuple, and data items are not in the objects set. *s_B* corresponds to *B* in the security model.

3.3. Safety Properties

The safety properties in FTLS are the same as in the security model and are expressed as follows (in COQ):

Simple-Security Property (*SimpleSecurity (v : State)*), Star-Security Property (*StarSecurity (v : State)*), Discretionary-Security Property (*DiscretionarySecurity (v : State)*), Object-Compatibility Property (*ObjectCompatibility (v : State)*), Entity-Integrity Property (*EntityIntegrity (v : State)*), Reference-Integrity Property (*ReferenceIntergrity (v : State)*).

These properties are the invariance in verification. As the Definition 1 describes, a state is safe if and only if it satisfies all these six safety properties, i.e. (in COQ)

Definition SecureState (v : State) := SimpleSecurity v /\ StarSecurity v /\ DiscretionarySecurity v /\ ObjectCompatibility v /\ EntityIntegrity v /\ ReferenceIntergrity v.

According to Definition 3, if we want to verify that the system is safe, we should first verify that the initial state of the system v_0 is safe. In FTLS, the initial state $v_0 = (O_0, DD_0, \Phi, \Phi)$, in which the user data and the access set are Φ. Because the initial access set *s_B* is Φ, Simple-Security Property, Star-Security Property and Discretionary-Security Property are satisfied obviously. Because the initial user data *s_UD* is *Φ*, Entity-Integrity Property and Reference-Integrity Property are also satisfied. Because the only object in the initial objects set is the new created database, Object-Compatibility Property is then satisfied. So the initial state of the system v_0 is safe.

3.4. SQL Operations

The operation set in FTLS is *OP = {select_op, insert_op, update_op, delete_op, create_op, alter_op, drop_op, grant_op, revoke_op, execute_op}*, in which there are all the SQL operations. Compared to the transition rules in the security model, the SQL operations in FTLS are more detailed and closer to the DBMS implementation.

In the base of system state specification and initial state verification, we can verify the whole FTLS. The key step is to verify whether arbitrary SQL operation is safe, so it is very important to specify the SQL operations. In this paper, considering the standard SQL syntax and the implementation of DBMS, we take *select* and *update* operations as examples to specify the SQL operations in FTLS in the view of security. First, these two SQL operations are normal and representative for everyday use. Second, they are also typical from the view of verification. As *union*, *intersection* and *difference* are only the operations for the results and not so complex, we won't consider these three operations in this paper. As the SQL operations for views can be translated to the operations for the base tables, we won't consider the operations for views neither.

Definition 4. *s_tc*. A first-in-last-out list whose elements are 2-tuple of <*t, c*>, which mean <table, where_clause>. The list records all the tables and their where clauses for filtering tuples in select operation.

Definition 5. select operation. A select operation is specified by this 3-tuple of <*v, s, s_tc*>, in which:

v∈VF; *s* is a subject, whose information can be got from *s_DD*; *s_tc* is as specified in Definition 4**.**

Definition 6. update operation. An update operation is specified by this 6-tuple of <*v, s, o, c, s_tc, s_av*>, in which:

v and *s* are the same as in Definition 5; *o∈s_ObjectSet*, represents the target table in update operation; *c* represents the potential where clause in update operation, and if there is no where clause, *c* is denoted as *1* (which means the condition is true); *s_tc* records the tables and their where clauses for filtering tuples in the potential select clause, and if there is no select clause, *s_tc* is denoted as *Φ*; *s_av* is a list whose elements are 2-tuple of <*att, value*>, which mean <attribute, value>, the list records all the attributes and their new values in update operation.

In the above definitions of SQL operations, *v, s, o, c* are single element, and all others are linear lists. This specification is easy to deal with for proof tools. However, SQL statements are often nested and complex themselves. It is a problem to transform the complex SQL statements to the specification as in Definition 5 in FTLS. So we will give the details of the rules for transformation.

3.5. SQL Statements Transformation

Before transformation, we need to analyse the syntax of SQL statements to confirm the components of each SQL statement, then we give the rules to transform these components to the specifications above. We still use *select* and *update* statements as examples for transformation.

1. Syntax analysis

The SQL92 standard specifies the complete SQL syntax. However, from the view of security, some components in SQL statements are needless. For example the *order by* clause in *select* statement has nothing to do with the security and can be ignored in our analysis. The following shows the refined components of each SQL statements.
(1) select statement

A select statement consists of these components: {*select_list, from_clause, where_clause, having_clause*}, in which:

select_list represents the column objects for selection and can be column name or *select_clause*. *from_clause* can be one or more tables, or *select_clause*. *where_clause* can be NULL, or include simple expression (contains no *select_clause*), or include complex expression (contains *select_clause*); *having_clause* is the same as *where_clause*.

(2) update statement

An update statement consists of these components: {*object, set_clause, where_clause*}, in which:

object represents the table object for update. *set_clause* represents the target columns and new values for update (a set whose elements are 2-tup of <*attribute, value*>). *where_clause* is the same as in *select* statement above.

2. Transformation rules

We denote the current state for SQL operation as *v* and the subject for SQL operation as *s* in the specification, and use the following rules for transformation.

Rule 1. select statement transformation.

① If *select_list* is column names, let them go; if *select_list* is *select_clause*, use Rule 1.

② There are three cases for *from_clause*:

a. If *from_clause* is a single table name *t*, then we treat the *where_clause* of this statement as *c*, and add <*t, where_clause*> into the *s_tc* list. If there is no *where_clause* (*where_clause* is NULL), *c* is marked as *1*, which means the expression for filtering is true.

b. If *from_clause* involves more than one tables, $t_1,\ldots,t_i\ldots$, i.e. the statement includes multi-table join, then we treat the conditions in the *where_clause* relevant to each table as $c_1,\ldots,c_i\ldots$, and add $<t_1, c_1>,\ldots,<t_i, c_i>\ldots$ into the *s_tc* list.

c. If *from_clause* includes *select_clause*, then we use Rule 1 to handle the *select_clause*.

③ If *where_clause* is NULL or includes only simple expression, let it go; if *where_clause* includes complex expression, use Rule 1 to handle the *select_clause* therein.

④ If *having_clause* is NULL or includes only simple expression, let it go; if *having _clause* includes complex expression, use Rule 1 to handle the *select_clause* therein.

Rule 2. update statement transformation.

① Treat the target table *object* as *o* in the specification of update operation.

② Add the 2-tup of <*attribute, value*> in the *set_clause* to the *s_av* list.

③ Treat the *where_clause* as *c* in the specification of update operation. If there is no *where_clause* (*where_clause* is NULL), *c* is marked as *1*, which means the expression for filtering is true; If *where_clause* includes only simple expression, let it go; if *where_clause* includes complex expression, use Rule 1 to handle the *select_clause* therein to form the *s_tc*.

For example, an update statement "set the price to 100 for the goods whose type is wine" is as follows:

UPDATE sell

SET sell.price = 100

WHERE sell.id = (SELECT id FROM store WHERE store.type = 'wine');

According to Rule 2, we treat the table "*sell*" as *o* in the specification of update operation, take "{<*price, 100*>}" as *s_av*, and treat "*sell.id = (SELECT id FROM store WHERE store.type = 'wine')*" as *c*; for the *select_clause* in its *where_clause*, we use Rule 1 to transform it to {<*store, store.type = 'wine'*>} and take it as *s_tc*. At last, the update statement is transformed to the specification in Definition 6 as: <*v, s, sell, sell.id = (SELECT id FROM store WHERE store.type = 'wine'), {<store, store.type = 'wine'>}, {<price, 100>}*>.

4. Analysis and Verification for SQL Operations

In section 3.4, we give the definitions of SQL operations in the view of security. As mentioned above, this multi-tuple specification only including linear lists and single element is much easier to deal with for proof tools than the original nested and complex SQL statements. However, as the verification process is very complex, we will do some further simplification. First, we give the definitions of simple SQL operations, and propose a method to verify those simple SQL operations. Then we transform the verification of the SQL operations in FTLS to the verification of those component simple SQL operations. We also give the proof of the correctness of this approach. The same as above, we use *select* and *update* operations as examples for the analysis and verification.

4.1. Analysis and Verification for Select Operation

1. Analysis of simple select operation

Definition 7. simple select operation. A simple select operation is specified by this 4-tuple of <*v, s, t, c*>, in which:

v∈*VF*; *s* is a subject, whose information can be got from *s_DD* in *v*; *t*∈*s_ObjectSet*, whose type is table; *c* represents the where_clause for filtering tuples which includes only simple expression (contains no clause).

In secure DBMS, simple select operation rule is: If the object table *t* for select exists, and subject *s* has the *select_op* privilege on *t*, and the security level of *s* dominates the security level of *t*, then filter the tuples in *t* based on the security level and *c*, and for the qualified tuple *o*, add (*s, o, select_op*) to the *s_B* in the state; otherwise the operation fails and the state stays unchanged.

Figure 1 shows the specification of simple select operation rule.

```
select(v, s, t, c):
if     objType(t) = table ∧ HavePriv(v,(s,t,select_op)) = true /\
       $f_c(v,s) \geq f_o(v,t)$
then   ∀ ( o ∈includedby (t) /\ c(o) = true /\ $f_c(v,s) \geq f_o(v,o)$ ),
       v*= ( (s_ObjectSet v), (s_DD v), (s_UD v), select_addB(o) )
else   v*=v
```

Figure 1. Specification of Simple Select Operation Rule

v represents the pre-state of the operation, and *v** represents the post-state. After the operation succeeds, the first three parts of *v** are the same as in *v*. *select_addB(o)* means that for the qualified tuple *o*, add (*s,o,select_op*) to the *s_B* in the state. *objType(t) = table* means the type of *t* is table. *HavePriv(v,(s,t,select_op))* is the function to judge whether *s* has the *select_op* privilege on *t*. $f_c(v,s)$ and $f_o(v,t)$ are the functions to get the security levels of subject *s* and object *t* respectively. *o∈includedby(t)* means *o* is the object included by *t*. *c(o) = true* means *o* satisfies the filtering condition *c*. For readability, the specifications and verifications are all written improved on the original COQ codes.

2. Verification of simple select operation

According to Definition 2, the object for verifying an operation is that on the premise pre-state of the operation is safe, the post-state of the operation is also safe, i.e. *SecureState(v*) = true*. We will verify each safety property.

As an example for Simple-Security Property, Figure 2 shows the lemma of Simple-Security Property for simple select operation.

```
Lemma selectSP:
if     BasicProperties(v) = true /\ SimpleSecurity(v) = true
       select(v,s,t,c) = v*
then   SimpleSecurtiy(v*) = true
```

Figure 2. Lemma of Simple-Security Property for Simple Select Operation

s is the subject of the operation, *t* is the table object of the operation, and *c* is the condition for filtering tuples. *BasicProperties* includes some basic properties relevant to the database, such as a table and its tuples belong to the same database, a table and its tuples are father and sons, etc. These properties seem obvious, but the verification will not succeed without them, which shows the rigor of the proof tool. *select* is as specified in Figure 1.

This lemma shows: if the pre-state satisfies the Simple-Security Property, after the simple select operation, the post-state also satisfies the Simple-Security Property. The verification of this lemma is the verification of the Simple-Security Property for the simple select operation.

We list some of the common commands for verification in COQ: *unfold, intros, inversion, rewrite, elim, replace, apply, symmetry, destruct, generalize, auto,* etc. Other verifications of simple operations are similar to the verification of the simple select operation, which we will not show here any more.

3. Analysis and verification of select operation

Comparing definition 5 with definition 7, we can find that the difference between select operation and simple select operation is: the specification of select operation includes *s_tc*, which is a list whose elements are 2-tuple of <*t*, *c*>, while there is only one *t*, *c* in the specification of simple select operation.

Figure 3 shows the specification of select operation rule.

```
select(v, s, s_tc):
case  s_tc
     nil => v
     tc :: ltc =>
{
if      objType(fst tc) = table  ∧  HavePriv(v,(s,(fst tc),select_op)) = true  ∧  $f_c$(v,s) ≥
        $f_o$(v,(fst tc))
then    ∀ ( o ∈includedby (fst tc) ∧ (snd tc)(o) = true ∧ $f_c$(v,s) ≥ $f_o$(v,o) ),
        v*= ( (s_ObjectSet v), (s_DD v), (s_UD v), select_addB(o) )  ∪  select(v, s, ltc)
else    v*= ( (s_ObjectSet v), (s_DD v), (s_UD v), (s_B v) )  ∪  select(v, s, ltc)
}
```

Figure 3. Specification of Select Operation Rule

s_tc is the list specified in definition 4. "*::*" is the symbol for concatenation. *tc :: ltc* represents the list by concatenating the list *ltc* to the element *tc* (which means *tc* is the first element in the new list). *(fst tc)* represents the first element (table) in the 2-tuple *tc*, while *(snd tc)* represents the second element (condition) in the 2-tuple *tc*. Other symbols are the same as in the specification of simple select operation rule.

The object for verifying select operation is the same as for simple select operation, which is on the premise pre-state of the operation is safe, the post-state of the operation is also safe. The difference is: the post-state of the select operation is more complex to describe. As specified in Figure 3, we need to analyse each element in the list *s_tc*, and add each result for the analysis together to form the final post-state *v**.

As *s_tc* is a first-in-last-out list, the node taken out prior is the one added into the list later, which represents the deeper select clause in the select statement (it doesn't matter which table is prior in the same hierarchy for table join). So we can simplify the verification of the select operation. First we take out all the nodes <t_i, c_i> from the list *s_tc* sequentially, and form the simple select operations <*v*, *s*, t_i, c_i> with state *v* and subject *s*. Then we invoke the verification of simple select operation mentioned above to verify those simple select operations in order, which is easier, so that we can transform the verification of select operation to those of sequential simple select operations. In this way, the first verified clause is the deepest select clause, while the last verified is the main select clause.

For example, the transformation, analysis and verification of the select statement:

SELECT ts.sname, tsc.score, (SELECT AVG(score) FROM tsc WHERE cno = '100001') Average FROM ts, tsc WHERE ts.sno=tsc.sno AND tsc.cno = (SELECT cno FROM tc WHERE cname = 'Chinese');
is like this:

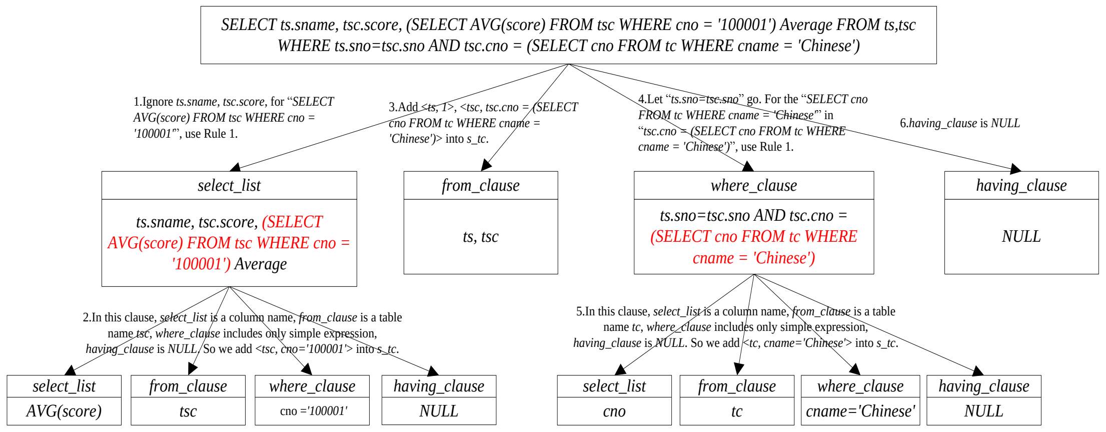

SELECT ts.sname, tsc.score, (SELECT AVG(score) FROM tsc WHERE cno = '100001') Average FROM ts,tsc WHERE ts.sno=tsc.sno AND tsc.cno = (SELECT cno FROM tc WHERE cname = 'Chinese')
1.Ignore ts.sname, tsc.score, for "SELECT AVG(score) FROM tsc WHERE cno = '100001'", use Rule 1.
3.Add <ts, 1>, <tsc, tsc.cno = (SELECT cno FROM tc WHERE cname = 'Chinese')> into s_tc.
4.Let "ts.sno=tsc.sno" go. For the "SELECT cno FROM tc WHERE cname = 'Chinese'" in "tsc.cno = (SELECT cno FROM tc WHERE cname = 'Chinese')", use Rule 1.
6.having_clause is NULL
select_list
ts.sname, tsc.score, (SELECT AVG(score) FROM tsc WHERE cno = '100001') Average
from_clause
ts, tsc
where_clause
ts.sno=tsc.sno AND tsc.cno = (SELECT cno FROM tc WHERE cname = 'Chinese')
having_clause
NULL
2.In this clause, select_list is a column name, from_clause is a table name tsc, where_clause includes only simple expression, having_clause is NULL. So we add <tsc, cno='100001'> into s_tc.
5.In this clause, select_list is a column name, from_clause is a table name tc, where_clause includes only simple expression, having_clause is NULL. So we add <tc, cname='Chinese'> into s_tc.
select_list
AVG(score)
from_clause
tsc
where_clause
cno ='100001'
having_clause
NULL
select_list
cno
from_clause
tc
where_clause
cname='Chinese'
having_clause
NULL

Figure 4. Example for Transformation, Analysis and Verification of Select Operation

As depicted in Figure 4, first we use Rule 1 in section 3.5 to transform the original statement to the specification as in definition 5 like this: *<v, s, {<tc, cname='Chinese'>, <tsc, tsc.cno=(SELECT cno FROM tc WHERE cname='Chinese')>, <ts, 1>, <tsc, cno='100001'>}>* . Then as described in this section, we take out the nodes from the list *s_tc* in the above specification sequentially, and form the 4 simple select operations: *<v, s, tc, cname='Chinese'>*, *<v, s, tsc, tsc.cno=(SELECT cno FROM tc WHERE cname='Chinese')>*, *<v, s, ts, 1>*, *<v, s, tsc, cno='100001'>*. At last we invoke the verification of simple select operation to verify these simple select operations in order.

4. The correctness of the verification of select operation

The following theorem shows the correctness of the verification of select operation mentioned above.

Theorem 1. The approach to transform the verification of select operation to those of sequential simple select operations is correct.

Proof. From the analysis of select operation, we can see that the change of the successful operation on the state is for *s_B*, i.e. to add the (*s*,*o*,*select_op*) to the *s_B* in the state for those qualified tuple *o*. And which *o* is qualified is decided by *t*, *c* and the security levels. From the specifications of select operation and simple select operation, we can see that the proof of Theorem 1 is the same as the correctness proof of the approach to transform the *s_tc* to each single $<t_i, c_i>$ in order for verification. According to Rule 1 in section 3.5, there are two relationships among the nodes $<t_i, c_i>$ in the list *s_tc*: 1) the nested select clause in a lower hierarchy and the main select clause in a higher hierarchy; 2) the different tables in the same hierarchy for table join. We will discuss these two cases respectively.

1) Because *s_tc* is a first-in-last-out list, the node taken out prior is the one added into the list later, which represents the deeper select clause in the select statement. So when we take out the nodes $<t_i, c_i>$ from the list *s_tc* sequentially, and form the simple select operations $<v, s, t_i, c_i>$ for verification, we are first verifying the select clause in a lower hierarchy and then the main select clause in a higher hierarchy, which is consistent with the execution of the select statement. So this is obviously correct.

2) For the table join, the proof is by contradiction. Let *Sec(O)* be the proposition that it is secure to add (*s*, o_i, *select_op*) into *s_B* for all o_i in the tuple set *O*. Let $<t_1, c_1>$, $<t_2, c_2>$ be the

nodes in the *s_tc* list where t_1 and t_2 are the only tables for join. Let O_1 be the tuple set in which all tuples are qualified for t_1, c_1 and the security level, and O_2 for t_2, c_2. Let O be the tuple set in which all the tuples are accessed at last in the original select statement. Then our proof for this case is simplified like this: $Sec(O_1) \wedge Sec(O_2) \rightarrow Sec(O)$. Because c_1, c_2 are the filtering conditions relevant to each single table t_1, t_2 respectively, we have $O \subset O_1 \cup O_2$. Suppose that $Sec(O)$ is false, then $\exists$ o $\in$ O, so that $\neg$ Sec(O). As $O \subset O_1 \cup O_2$, we have $o \in O_1$ or $o \in O_2$. Assume that $o \in O_1$, than $\neg Sec(O_1)$, which contradicts the condition $Sec(O_1)$. So the proposition is true. This proof can also be extended to joins for multiple tables.

In conclusion, the approach to transform the verification of select operation to those of sequential simple select operations is correct. □

4.2. Analysis and Verification for Update Operation

1. Analysis of simple update operation

Definition 8. simple update operation. A simple update operation is specified by this 5-tuple of <*v, s, t, c, s_av*>, in which:

v, s, t, c are the same as in Definition 7; *s_av* is a list whose element is 2-tuple of <*att, value*>, and represents the attributes and their new values in update operation.

In secure DBMS, simple update operation rule is: If the object table *t* for update exists, and subject *s* has the *update_op* privilege on *t*, and the security level of *s* dominates the security level of *t*, then filter the tuples in *t* based on the security level and *c*, and check the new values whether they satisfy the integrity constraint, then for the qualified tuple *o*, add (*s, o, update_op*) to the *s_B* in the state, and modify the values in user data *s_UD*; otherwise the operation fails and the state stays unchanged.

Figure 5 shows the specification of simple update operation rule.

```
update(v, s, t, c, s_av):
if    objType(t) = table $\wedge$ HavePriv(v,(s,t,update_op)) = true $\wedge$ $f_c(v,s) \geq f_o(v,t)$
then  $\forall$ ( o $\in$includedby (t) $\wedge$ c(o) = true $\wedge$ $f_c(v,s) = f_o(v,o)$
          $\wedge$ ( if  $\exists$  (att,val) $\in$s_av, att $\in$FindPkey(v,t)
             then   val $\neq$ null $\wedge$ ( $\forall$ $o_i \in$ includedby (t), $o_i \neq o$, $f_o(v, o_i) \neq f_o(v,o)$ $\vee$ FindValue((s_UD v),$o_i$,att) $\neq$ val )
             )
          $\wedge$ ( if  $\exists$  (att,val) $\in$s_av, att $\in$FindFkey(v,t)
             then   val=null $\vee$ ( $\exists$ $o_j$, objType($o_j$)=tuple,
                    fa($o_j$)=FindRefTable(v,t,att), $att_j$=FindRefAtt(v,t,att),
                    val=FindValue((s_UD v),$o_j$,$att_j$) $\wedge$ $f_o(v, o_j)=f_o(v,o)$ )
             )
        )
      v*= ( (s_ObjectSet v), (s_DD v), update_changeUD(s_av), update_addB(o) )
else  v* = v
```

Figure 5. Specification of Simple Update Operation Rule

After the operation succeeds, the first two parts of *v** are the same as in *v*. *update_changeUD(s_av)* means that update the user date *s_UD* in the state according to *s_av*. *update_addB(o)* means that for the qualified tuple *o*, add (*s,o,update_op*) to the *s_B* in the state. *FindPkey(v,t)* is the function to find the primary key of table *t*. *FindValue((s_UD* *v),o_i,att)* is the function to find the value of the attribute *att* in the tuple o_i. *FindFkey(v,t)* is the function to find the foreign key of table *t*. *FindRefTable(v,t,att)* is the function to find which table is referenced by the attribute *att* of the table *t*. *FindRefAtt(v,t,att)* is the function to find which attribute is referenced by the attribute *att* of the table *t*.

2. Analysis and verification of update operation

Comparing definition 6 with definition 8, we can find that the difference between update operation and simple update operation is: the specification of update operation includes *s_tc*, which is a list whose elements are 2-tuple of <*t*, *c*>, representing the select clauses for filtering the tuples; while there is only one table *t* and condition *c* including only simple expressions in the specification of simple update operation.

The specification of update operation rule is similar to the specification of simple update operation rule. In the specification of simple update operation rule, *c(o)=true* means tuple *o* is qualified for the filtering condition *c*, which is a simple expression; while in the specification of update operation rule, *c* is a complex expression that may include select clauses. So we need to invoke the specification of select operation rule mentioned above to specify the update operation rule.

The object for verification of update operation is the same as for simple update operation, which is on the premise pre-state of the operation is safe, the post-state of the operation is also safe. The difference is: except for the changes caused by the object tuples of the update, the post-state of the update operation is also effected by the select clause, i.e. the post-state of the successful update operation is *v*= ((s_ObjectSet v), (s_DD v), update_changeUD(s_av), (update_addB(o))) ∪select(v, s, s_tc).*

So we can verify the update operation in two steps: first, we invoke the verification of select operation to verify the *s_tc* list, which represents the select clauses in the update operation; then, for the *v, s, o, c, s_av*, which forms the main clause of the original update operation, we just verify it as for the verification of simple update operation.

3. The correctness of the verification of update operation

Theorem 2. The approach to transform the verification of update operation to the verification of select operation and verification of simple update operation is correct.

Proof. The security policy of select operation is "no read up", i.e. no subject is allowed to read from any object with a higher security level. This should also be satisfied for the select clause in the update operation. So it is necessary to invoke the verification of the select operation to handle the *s_tc* list in update operation. One can refer to the proof for Theorem 1 for this situation.

When the verification of the select clauses passes, the *c* in the main update clause is then confirmed. The security policy of update operation is that a subject can only modify the object tuple with the same security level. From the specification of the simple update operation, we can see that the filtering for object tuples includes condition *c* and the equation for security levels. Then, for those qualified tuples *o*, we add (*s*, *o*, *update_op*) to the *s_B* in the state and modify the values in user data *s_UD*. So we need to verify this post-state using the verification of simple update operation.

In conclusion, the approach to transform the verification of update operation to these two steps is correct.

4.3. Some Problems in Verification

We wrote twenty thousands lines of COQ codes to analyse and verify a domestic DBMS including all SQL operations mentioned in Section 3.4. And when we did the verification we meet some problems. Such as how to make sure that the verification for the post-state really needs the precondition in the specification of the operation rules, so that the rigor of the proof is ensured; the FTLS is closer to the implementation of the DBMS than the model, how to simplify its verification due to the details of the implementation; the function of certain SQL operations seems conflicted to the security policy, etc. Here we give an example in the verification of delete operation.

The set *s_B* in the state represents the records of accesses which we need to analyse for the verification of Simple-Security Property, Star-Security Property and Discretionary-Security Property. However, the target tuple of delete operation has already been deleted from the object set *s_ObjectSet* after the operation. For the consistency of the state specification, the relevant element in the set *s_B* should also be deleted. Then the *s_B* in the post-state of the delete operation is the subset of the *s_B* in the pre-state. So it will be obvious that the post-state satisfies the safety properties on the premise pre-state satisfies the safety properties without the specification of the delete operation rule, which is not exactly right. To resolve this problem, when we verify the operations such as delete and drop, we introduce a temporary state whose set *s_B* includes the elements like (*s, o, delete_op*) and (*s, o, drop_op*). After the verification, we only keep the final consistent state. As a result, the qualifications in the specification of the operation rules can not be skipped in the verification, and the rigor of the proof is ensured.

5. Related Work

First, we will show some related work of the formal verification for some general systems. Then for the formal verification of DBMS, especially for the classical SeaView project, we will give some introduction. At last, we will introduce some relevant work about SQL formal specification.

5.1. Formal Verification for General Systems

Goguen and Meseguer say that building a secure system should be comprised of four stages [14]: 1) Determine the security needs of the system; 2) Express those needs as a formal requirement; 3) Model the system (at least the security relevant components and functions); 4) Verify that this model satisfies the formal requirement.

Maximiliano Cristia specifies and verifies an extension of a secure, compatible UNIX file system [15]. The paper indicates that a secure file system should include subjects and their groups, list of privileges, security level information, files, indices, etc. The operations should include create, open, close files and folders, etc. The paper also defines the safety properties, and introduces the concept of state machine. The operations in the file system are treated as queries or changes to the state. Their target for verification is to verify whether the operation keeps the security of the state. The specification and verification are also written in COQ proof tool. However, the target object in their work is file system, which differs a lot from the DBMS. The latter is more complex in operations. And the transformation and simplification for verification in our paper is never mentioned before in general systems.

Antonio Coronato, et. al., propose a method to formally specify and verify the correctness and security of the general application system [16]. They extend the basic formal tools and introduce static and dynamic verification briefly. However, this paper gives the specification so abstractly that doesn’t refer to any practical problems in general application systems.

Hejiao Huang, et. al., point out that the security policy design is concerned with the composition of components in security systems and interactions among them [17]. The paper addresses the problem in a formal way and uses CPNP to specify and verify security policies in a modular way. They define fundamental policy properties, e.g. completeness, termination, consistency, and confluence in Petri net terminology and get some theoretical results.

5.2. Formal verification for DBMS

In the research of formal specification and verification for DBMS, the classical one is the SeaView model proposed by Teresa F.Lunt et al. [5, 6]. It is a formal security model for multi-level security RDBMS. The object is to design a multi-level secure DBMS with the A1 security level in TCSEC.

SeaView security policies include mandatory access control policy, discretionary access control policy, data marker, data consistency, etc. These policies are formalized into two model layers: the inner layer is MAC model, and the outer layer is TCB model. The MAC model includes a security kernel that supports the A1 security level. The TCB model is based on the MAC model and implemented by the extended TCB, which includes the multi-level security relation abstract, integrity constraint, discretionary authorization, etc. And the specification of the operations is also in this layer.

However, in SeaView project, only simple SQL operations are considered. For example, the target object of the update operation they specified is only certain tuples whose primary keys are already known, while the where clause that may contain complex expressions is not considered. Actually, the complex clause determines the tuples to be updated and also the objects whose security levels will be compared, so it has much to do with the security. This is normal in current complex update operation.

5.3. Formal Specification for SQL

Li Xu-shuai, et. al., define a formal three-valued predicate model EP^MC based on the medium logic predicate calculus system MF^M to express a SQL query, and also define the rules by which the SQL query sentences are transformed into EP^MC [3]. They formally analyse the SQL query in the semantics. But their object is to prove the equivalence of two SQL sentences rather than security analysis. And they only consider the SQL query sentence.

Li Hai-long et al. propose a model in which they define a standard SQL clause object ANSISQLO and build some rules [4]. They analyse the SQL in two phases: formal-rule phase, in which they generate the ANSISQLO for a SQL sentence by syntax analysis; logic-rule phase, in which they analyse the constraints for database entities and attributes. They focus on the simple SQL. Though they mention the complex SQL with nesting SQL syntax, the analysis is not sufficient.

Maryam Lotfi Shahreza et al. point out that the theory of the relational databases has much in common with the mathematical structures central to the Z notation, and formally specify the applications in DBMS [18]. They first specify the database by UML class diagram in Z. Then they refine the specification until its corresponding database and program is obtained. Finally, they get the SQL operations corresponding to the specifications above. However, they pay more attention on the specification of database applications instead of security.

6. Conclusion

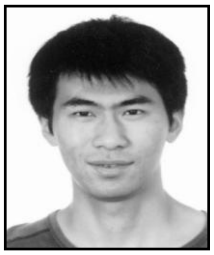
In this paper, we propose a novel approach for the formal specification and verification of the SQL operations in FTLS of a secure DBMS. First, we formally define the SQL operations in FTLS and build the rules to transform the SQL sentences to these specifications. Then, on top of the specifications, we give the definitions of the simple SQL operations, and propose a method to verify those simple SQL operations. Finally, we transform the verification of the SQL operations in FTLS to the verification of the component simple SQL operations. We also give the proof of the correctness of this approach. From the process of the specification and verification, we can see that our approach makes a comprehensive and clear specification of the SQL operations in FTLS, and also makes an easier verification for proof tool COQ. So we resolve the problems of the security verification of FTLS mentioned in the introduction.

To the practical problems in the specification and verification of FTLS, such as the verification of objects deletion, verification of some specific queries, etc. we also did the research. And we find that it is very useful to specify and verify the FTLS of a DBMS for design of a secure DBMS.

References

[1] DEPARTMENT OF DEFENSE, “Trusted Computer System Evaluation Criteria”, DoD 5200.28-STD, **(l985)**.
[2] Common Criteria, “Common Criteria for Information Technology Security Evaluation”, ISO/IEC 15408, **(1999)**.
[3] X. Li and Y. Mao, “Formal Semantics of SQL”, Microcomputer Development, vol. 15, no. 3, **(2005)**.
[4] H. Li, W. Zhang, X. Li and W. Xiao, “Custom Standard SQL Grammer Analysis Model”, MINI-MICRO Systems, vol. 24, no. 11, **(2003)**.
[5] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman and W. R. Shockley, “The SeaView Security Model”, IEEE Transactions on Software Engineering, vol. 16, no. 6, **(1990)**.
[6] R. A. Whitehurst and T. F. Lunt, “The SeaView Verification”, Proceedings of the Computer Security Foundations Workshop II, **(1989)** June 11-14; Franconia, NH, USA.
[7] S. Zou and G. Zheng, “Comparison of Z and VDM with B”, Computer Science, vol. 29, no. 10, **(2002)**.
[8] L. Yu and Y. Dajun, “Analysis of Theorem–proving Aiding Tool—PVS”, Computer Engineering, vol. 26, no. 9, **(2000)**.
[9] M. Wenzel, “The Isabelle/Isar Reference Manual”, **(2009)**.
[10] S. Owre, N. Shankar, J. M. Rushby and D. W. J. Stringer-Calvert, “PVS Language Reference”, **(2001)**.
[11] The Coq Development Team, “The Coq Proof Assistant Reference Manual”, **(2006)**.
[12] L. J. LaPadula and D. E. Bell, “MITRE Technical Report 2547”, Volume II [Secure computer systems: rules of operation], Journal of Computer Security, vol. 4, **(1996)**, pp. 2-3.
[13] Z. Hong, Z. Yi, L. Chenyang, S. Jie, F. Ge and W. Yuanzhen, “Formal Specification and Verification of an Extended Security Policy Model for Database Systems”, Proceedings of the 3rd Asia-Pacific Trusted Infrastructure Technologies Conference, **(2008)** October 14-17; Hubei, China.
[14] J. A. Goguen and J. Meseguer, “Security Policies and Security Models”, Proceedings of the 1982 Symposium on Security and Privacy, **(1982)** April 26-28; Oakland, CA, USA.
[15] M. Cristia, “Formal Verification of an Extension of a Secure”, Compatible UNIX File System, Master's thesis, Instituto de Computación, Universidad de la República, Uruguay, **(2002)**.
[16] Antonio Coronato and G. De Pietro, “Formal Specification and Verification of Ubiquitous and Pervasive Systems”, ACM Transactions on Autonomous and Adaptive Systems, vol. 6, no. 1, **(2011)**.
[17] H. Huang and H. Kirchner, “Formal Specification and Verification of Modular Security Policy Based on Colored Petri Nets”, IEEE Transactions on Dependable and Secure Computing, vol. 8, no. 6, **(2011)**.
[18] M. L. Shahreza, A. Moeini and R. O. Mesbah, “Specification and Development of Database Applications Based on Z and SQL”, Proceedings - 2009 International Conference on Information Management and Engineering, **(2009)** April 3-5; Kuala Lumpur, Malaysia.

Authors



Zhipeng Wang

A PhD student in the School of Computer Sci & Tec, Huazhong University of Sci & Tec, China. He received the MS degree in Computer Software and Theory from HUST in 2009. His current research interests include database security and privacy, formalized analysis and verification.

E-mail: hkdwzp@gmail.com

Hong Zhu

A Professor in the School of Computer Sci & Tec, Huazhong University of Sci & Tec, China. She is an expert in the areas of DBMS, database security and privacy, data management and data mining.

E-mail: whzhuhong@gmail.com

Meiyi Xie

A lecturer in the School of Computer Sci & Tec, Huazhong University of Sci & Tec, China. She received the PhD degree in Computer Software and Theory from HUST in 2009. Her current research interests include database security and privacy, intrusion-tolerant database, and cloud computing.

E-mail: xiemeiyi@sina.com