

# XTOLS: Cross-tier Oracle Label Security

Jong-hoon (David) An

University of Maryland, College Park

davidan@cs.umd.edu

## Abstract

SELINKS allows cross-tier security enforcement between the application tier and the database tier by compiling policy functions and database queries into *user-defined functions* (UDFs) and SQL queries. Unfortunately, this kind of enforcement is restricted to the policies written within SELINKS framework; and therefore, it does not take into account the existing policies in the database. Furthermore, the data in the database may be vulnerable to unauthorized access because the database does not necessarily enforce the security policies intended by the application. To support fine-grained access control over sensitive data, Oracle introduced Oracle Label Security (OLS) technology, starting from Oracle 8i. However, there has been no previous work to incorporate this technology into the application framework. In this paper, we discuss how OLS security policies can be encoded in SELINKS and enforced between the application and the database. We have implemented an extension of current SELINKS, called Cross-tier Oracle Label Security (XTOLS), that provides a secure and extensible programming environment to programmers.

## 1. Introduction

Prior work on SELINKS[4] has shown that some security policies can be enforced across different tiers of a system. This idea of cross-tier enforcement has been addressed only recently in the database and programming language communities. Consequently, programmers could not reason about the security policies once the data has been stored into the database. SELINKS alleviates this problem by associating labels with sensitive data and compiling policy functions into *user-defined functions* (UDFs) that reside in

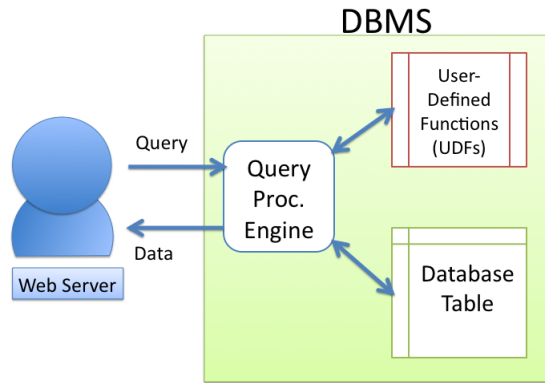
the database. This allows SELINKS applications to enforce security policies when querying data from the database. Figure 1 explains how the query (written in SELINKS) is processed in the DBMS, using user-defined functions.

Unfortunately, this cross-tier policy enforcement is limited to the policies written within SELINKS framework because the generated PL/SQL functions are only invoked by SQL queries generated from SELINKS code. In other words, other applications (not written in SELINKS) may access the data in the database directly (not via SELINKS framework) and violate the security policy. This may not be a problem and even manageable if the domain of the whole system is relatively small and administered by a single person or a small trusted group. However, in the environments where high security is critical or the domain of the system is large, this is definitely not desirable—e.g., in the Department of Defense.

To accommodate such environments in a DBMS, Oracle supports two exceptional technologies—Virtual Private Data (VPD)[2] and Oracle Label Security (OLS)[1], starting from Oracle 8i and Oracle 9i, respectively. Both allow more fine-grained access control than most DBMSs that offer only a discretionary access control (DAC), which enforces security policies based on the ownership of data[6][5]. DAC cannot enforce user dependent security policies because, for example, two owners of a data may disagree on granting a privilege to a third person. Thus, it becomes difficult to manage security policies among the users, especially as the size of data and the number of users increase.

In contrast, VPD and OLS technologies provide multi-level security which is closely associated with mandatory access control (MAC)[7], meaning that each data is specified with the least privilege a user needs in order to access the data. OLS technology, in particular, is implemented on top of VPD and provides extensive and flexible label based access control. Therefore, it is suitable for government and defense applications[8]. We only discuss OLS in this paper due to the time constraints.

Although SELINKS supports the cross-tier security enforcement using UDFs, it is impossible to integrate OLS into the SELINKS framework without writing the corresponding



**Figure 1.** Cross-tier, labeled-based security enforcement in SELINKS

policy functions. It is still possible to enforce identical policies within the SELINKS framework with the former method since it is capable of enforcing a superset of OLS policies. However, using this method, SELINKS cannot take into account the existing labels (both data and user) and label components because it is about compiling SELINKS policy functions into UDFs, not vice versa. Furthermore, there is no security policy defined in the database itself, and therefore, any direct access to the database may violate the security policy. Since OLS already provides a flexible environment for enforcing security in the database, it is worthwhile to look at how application can integrate OLS into its framework.

In this paper, we discuss OLS technology and present an extension to SELINKS, called XTOLS, which involves with some modifications to SELINKS language and writing APIs for SELINKS programmers. We also show the results of our experiment, comparing with the results from the previous work and discuss the challenges and future improvements.

## 2. Overview

We first describe the fundamental idea of OLS, along with some examples to show how a database administrator would setup OLS for sensitive data. Next we explain how SELINKS would incorporate OLS security policies into the framework and discuss the implementation detail of the library for OLS in the next section.

### 2.1 Oracle Label Security

OLS is a transparent access mediation between users and sensitive data in Oracle Database, meaning that a user obtain information based on her authorization without the knowledge of the existence of OLS. The database administrator defines a set of OLS *data labels* and associates each row of a table with exactly one label. Note that this causes the table to have an additional column, which can be hidden or shown to

users depending on the context. The database administrator also grants appropriate privilege, called a *user label*, to each user who has access to that particular table. It is important to distinguish discretionary access control (DAC) from OLS multilevel access control, which is closely associated with mandatory access control (MAC). In this paper, we assume that the readers are already familiar with these terms for simplicity. To the best of our knowledge, the latter is only supported by Oracle Database whereas the former has been supported by most DBMSs including Oracle Database.<sup>1</sup> We first look at the label components and next discuss how each label is defined. Lastly, we describe some of OLS access control policies in more detail.

#### 2.1.1 Label Components

An OLS label consists of three components: *levels*, *compartments*, and *groups*. Each component is internally stored as a tuple of a long form, a short form, a numeric value (and a parent group in the case of *groups*). In this section, however, we simply use the long form to represent each component. The following bullets describe the three dimensions in more detail:

- *Levels* indicate the level of sensitivity of the information where a greater number implies a greater sensitivity of the label. For example, typical levels used in a business environment are HIGHLY SENSITIVE, SENSITIVE, CONFIDENTIAL, and PUBLIC.
- *Compartments* represent the different areas of interest in the institution such as the departments. Compartments are also known as *categories*, as defined in Bell-La Padula model[3]. Some examples of compartments are FINANCIAL, MARKETING, and CHEMICAL.
- *Groups* identify the organizations that own or access the data, and this set is optional. This is how OLS supports DAC in a limited manner. Some examples are EASTERN REGION, ER NEW YORK, and ER DC. Unlike compartments, groups may be structured hierarchical, forming a forest of directed acyclic graphs. For instance, ER NEW YORK and ER DC are subordinates of EASTERN REGION.

#### 2.1.2 Data Labels

In OLS, each row of a sensitive database table has a label assigned to it as an additional column. Each data label consists of a level, a set of compartments, and a set of groups. OLS requires that the level is specified, but it does not require other components. For example, a label may have a level but no compartments may have been specified—in which case,

<sup>1</sup> There has been a prior work on SEPostgreSQL and SELinux. However, since this is an ad-hoc system configuration, we do not discuss them here.

any user who has authorization for specified level may access the data. We discuss OLS access control policies in Section 2.1.4 in more detail.

For convenience, we formalize a data label  $L_d$  as a 3-tuple  $\langle v, C, G \rangle$ , where

- $v$  is a level
- $C$  is a set of compartments
- $G$  is a set of groups

### 2.1.3 User Labels

The database administrator also assigns a set of labels, called *user labels*, to each user, specifying the *level*, the set of compartments, and the set of groups, for total six different modes. For the purpose of presentation, however, we consider only two labels—maximum read and minimum write. Oracle uses the term, *dominates*, to indicate that the user label satisfies the components of the data label. As an example, let us assume that user  $x$  has been assigned a label with level set to CONFIDENTIAL, compartments set to {FINANCIAL}, and groups set to {ER NEW YORK}. The user  $x$  can access any row with level set to CONFIDENTIAL or lower *and* the compartments set to any subset of {FINANCIAL} that belongs to the group ER NEW YORK or a higher echelon. We say that user label  $L_u^x$  dominates data label  $L_d^\alpha$  if user  $x$  can access the row  $\alpha$ . However, he cannot access any data labeled with compartments set to {FINANCIAL, CHEMICAL} even if the level of the data label is set to PUBLIC since the compartments of the user label is not a superset of the compartments of the data label.

Again, we can formalize a user label  $L_u$  as a 3-tuple  $\langle v, C, G \rangle$ , where

- $v$  is a maximum (minimum) level user  $u$  can read (write)
- $C$  is a set of compartments user  $u$  associates with
- $G$  is a set of groups user  $u$  belongs to or has access to

### 2.1.4 Access Control Policies

OLS has read and write access control policies based on the data label and the user label. It also supports special privileges for users—READ, FULL, COMPACCESS, PROFILE\_ACCESS, and more. Although we have taken into account these privileges for our implementation up to some degree, we do not discuss them here for simplicity. Figure 2 shows two functions *ReadAccess* and *WriteAccess* which determines whether the user label dominates the data label or not. In other words, the user has read (or write) access to the row if the corresponding data label  $L_r$  is dominated by a user label  $L_s$  (that is, the formula is satisfied). Note that  $\triangleleft$  refers to a subgroup relation—i.e., if  $g_1 \triangleleft g_2$  then  $g_1$  is a subgroup (inclusive) of  $g_2$ .

### 2.1.5 Example

Let us consider the following setting:

$$\begin{aligned} \text{ReadAccess}(L_d, L_u) = & \\ & (L_d[v] \leq L_u[v]) \wedge (L_d[C] \subseteq L_u[C]) \wedge \\ & (L_d[G] = \emptyset \vee \exists g \exists g'. g \in L_u[G] \wedge g' \in L_d[G] \wedge g' \triangleleft g) \end{aligned}$$

$$\begin{aligned} \text{WriteAccess}(L_d, L_u) = & \\ & (L_d[v] \geq L_u[v]) \wedge (L_d[C] \subseteq L_u[C]) \wedge \\ & (L_u[G] = \emptyset \vee \exists g \exists g'. g \in L_u[G] \wedge g' \in L_d[G] \wedge g' \triangleleft g) \end{aligned}$$

**Figure 2.** OLS access control policies for read and write

$$\begin{aligned} L_d^\alpha &= \langle \text{PUBLIC}, \{\text{MARKETING}\}, \\ &\quad \{\text{ER NEW YORK}, \text{ER BOSTON}\} \rangle \\ L_d^\beta &= \langle \text{SENSITIVE}, \{\text{FINANCIAL}, \text{CHEMICAL}\}, \\ &\quad \{\text{WESTERN REGION}\} \rangle \\ L_u^x &= \langle \text{CONFIDENTIAL}, \{\text{MARKETING}\}, \{\text{ER NEW YORK}\} \rangle \\ L_u^y &= \langle \text{SENSITIVE}, \{\text{CHEMICAL}\}, \{\text{ER NEW YORK}\} \rangle \end{aligned}$$

where  $\alpha$  and  $\beta$  are table rows and  $x$  and  $y$  are OLS users. Two examples of access control policy results are as the following:

$$\begin{aligned} \text{ReadAccess}(L_d^\alpha, L_u^x) &= \text{true} \\ \text{WriteAccess}(L_d^\alpha, L_u^x) &= \text{false} \\ \text{ReadAccess}(L_d^\beta, L_u^y) &= \text{false} \end{aligned}$$

## 2.2 SELinks

This section explains why it is not sufficient to enforce OLS only in the database for a web-based framework and why using UDFs cannot suffice in some cases.

### 2.2.1 OLS without Cross-tier Enforcement

OLS guarantees that no labeled data is accessed by a user whose user label does not satisfy the OLS policy. It is therefore legitimate to use OLS as a security enforcement for reading (writing) data from (to) the database, but the policies become useless once the data is read and remain in the SELINKS framework. For instance, it is possible that an application writes some of the retrieved data to an external storage such as a file or sends the data over the network. In this case, it is inevitable that we encode OLS security policies into SELINKS policy functions to prohibit any unauthorized tasks within the framework, which is our main contribution in this paper.

Of course, it is up to the policy writer (the trusted SELINKS administrator) whether to extend OLS policy functions to meet different criteria or not. It is, however, not desirable to violate the original OLS policy, especially if the data is written back to the database. For example, consider the following SELINKS code snippet:

```
var hdl =
  table "Table1"
  with (id: Int, description: String, label: Int)
  from (database "orcl");
for (var row ← hdl) { [row] };
```

This code connects to the database `orcl` and performs a `select` query on `Table1`. Because the table is labeled using OLS, only appropriate rows will be returned to the application based on the current user profile. The result of executing this code is a list of rows, which can be accessed without any policy code since they are not labeled. If a user programmer maliciously or mistakenly modifies any of the data and store it to a file, for example, the security policy would be violated once some other user accesses the data in the file without any security enforcement. Thus, it is important to detect such information flow even in the SELINKS framework. Furthermore, if the original user modifies one of the returned rows to have a level lower than his user label's minimum write level, the database will not process the query and send back an error message. In the latter case, we may assume that it is permissible to do so, but it is clearly desirable to issue a warning in the event of such an unauthorized action as early as possible—e.g., before the query is sent to the database.

### 2.2.2 Using UDFs

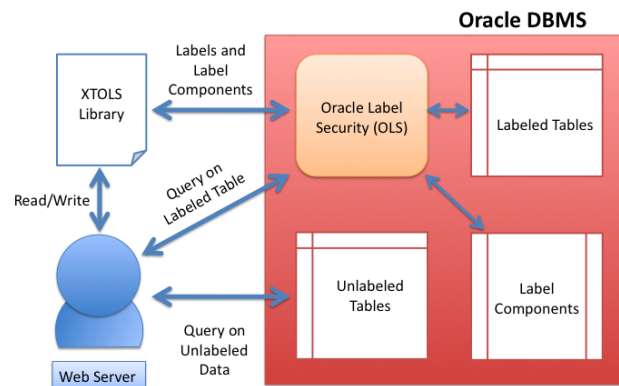
The cross-tier security label enforcement using UDFs, as introduced in [4], already has the capability to encode OLS policies within the SELINKS framework. This is possible because SELINKS can theoretically enforce a superset of the OLS security policies. As we mentioned before, however, the policy writer still has to provide policy functions that correspond to the OLS policies. Thus, by defining OLS policies in SELINKS beforehand, the policy writer can save his time. Some may argue that we can use the existing SELINKS code that reflects the OLS security policies. However, it is not possible to recognize an arbitrary SELINKS code that correspond to OLS (an undecidable problem). Furthermore, it is not desirable to use the method without the OLS security policies enforced in the database because any direct access to the database may violate the intended security policy.

Recall that we cannot simply prohibit users from accessing the table or the database since this kind of protection mechanism cannot enforce multilevel security. The reason OLS is used in the first place is that the multilevel security is preferred to DAC. For example, the government can

hire a database administrator to dispense some `SECRET` data, which suggests that we must not allow him to access `TOPSECRET` data. However, we cannot enforce such policy using just DAC because DAC controls data ownership, not data sensitivity. This clearly explains that the use of OLS is critical in some environments.

### 2.3 XTOLS

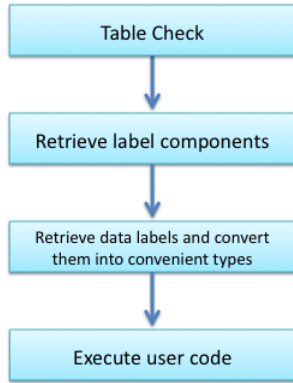
In this section, we describe the architecture of XTOLS, which is composed of a library package and an extension to the SELINKS language. Figure 3 shows the overview of XTOLS architecture, which also depicts the interactions with Oracle DBMS and OLS. To assure the safety guarantees in the SELINKS framework, XTOLS first verifies each table definition—that is, it checks whether or not any labeled table is defined as if it is not labeled. More detail on this issue is discussed in Section 2.3.1. Once the table definitions are verified, we must collect the set of label components and data labels defined in the database for convenience and efficiency. This is explained in Section 2.3.2. Figure 4 illustrates the execution steps of the XTOLS framework.



**Figure 3.** OLS security enforcement in SELINKS framework

#### 2.3.1 Table Checker

SELINKS requires that each database table used is statically defined—that is, the user programmers must annotate each table with its schema if the table is used in the code. This is because LINKS cannot infer types for tables. Unfortunately, this makes our safety guarantee to not hold true since anyone can maliciously define a table and obtain the rows from the database as not labeled. Some can argue that this is not an issue because the data returned from the database is accessible from that user anyway. However, the purpose of encoding OLS policies in SELINKS is to guarantee the same



**Figure 4.** Execution steps of a XTOLS application after parsing.

safety for the sensitive data throughout different tiers of the system, not just between the database and the server application. Therefore, it seems inevitable that SELINKS verifies every table definition to see whether the table rows are labeled by OLS or not. Once this is done, no user programmer can maliciously or mistakenly define a labeled row as not labeled.

### 2.3.2 Labels and Label Components

Encoding the OLS policies in SELINKS is a straight-forward task because the algorithms for various policies are described in [9], some of which we conveniently translated into first order logic in Section 2.1.4 for the purpose of presentation. In order to write efficient policy functions, we must first transform actual user labels and data labels into convenient forms since they are stored as database tables in unnormalized form. To do this, we initially obtain the set of label components (levels, compartments, and groups) and the set of data labels. Recall that the data labels must be defined prior to their usage, and therefore, an arbitrary data label cannot be used when inserting or updating a row (we will explain why it must be defined first in a moment). Because of this, we assume that no label is created, updated, or deleted outside SELINKS framework.<sup>2</sup> We also assume that the policy writer *always* writes correct policy functions—that is, the policy functions correctly encode the intended security policies, which is essential to prove the safety guarantee in FABLE, the underlying type system for SELINKS security enforcement[10].

Once the label components are retrieved from the database, it is up to the policy writer’s design decision whether or not

<sup>2</sup>It is possible to verify changes in OLS labels or label components, but it may be inefficient due to frequent runtime checks. Therefore, this concern is outside the scope of this paper.

to make that information public to user programmers (or users). For the purpose of presentation, we always reveal the label components and data labels in this paper. Notice that this is not a critical issue because data can be labeled and unlabeled only via policy functions, which are correctly written exclusively by the policy writers. The label components are stored as tables, each of which represent one kind of label component, and in default, can be accessed only by the system administrators and the label security administrators. At this initial stage, SELINKS accesses the database as a label security administrator, and therefore, can access all the label information.

The data labels are also retrieved from the database and maintained throughout the SELINKS framework. The primary reason for doing so is because the tags *used* to label sensitive data in the database are numeric values of the actual labels rather than pointers to actual labels. Because the numeric values are pre-defined by a label security administrator, we cannot use arbitrary label tags in the first place. Each (actual) data label stores the label information using a single string where each kind of component is delimited using ‘.’. For example, “A::B,C::D” would refer to a label that has the level set to *A*, the compartments set to  $\{B, C\}$ , and the groups set to  $\{D\}$ . In order to process labels more efficiently, we convert this attribute into a record type whose attributes are level, compartments, and groups. The user label is also retrieved, and the attributes are normalized for efficient process. Unlike data labels, however, we only retrieve one user label at a time after an appropriate authentication.

## 3. Implementation

In this section, we describe our prototype of XTOLS, which is implemented as library functions and as an extension to the SELINKS language. First, we explain how we modified SELINKS preprocessor to extend the language to provide syntax for OLS-related operations and to verify table definitions. Next, we present our implementation of OLS API for SELINKS.

### 3.1 The Language Extension

As mentioned before, it is necessary to verify table definitions to see whether any definition violates security policy or not. To do so, we extend the SELINKS language so that user programmers (or policy writers) can define tables accordingly. In other words, they must define a normal table (not labeled) using the conventional `table` syntax and define a labeled table using a newly introduced syntax, `olsdefine`. We also support `olsquery` to replace for loop of a table handle (which basically gets converted into a `select` query). Instead of modifying SELINKS parser, we added an additional task in the SELINKS preprocessor to support these constructs. Although this works perfectly, it

reduces the performance of the application dramatically. At this point, we have not attempted to modify the actual parser since it is very tricky to understand LINKS source code. The table checker is not implemented due to the limitation of the preprocessing. As it is done in [4], we assume that they are correctly defined by policy writers and user programmers.

### 3.2 XOLS APIs

First, we define a set of SELINKS types that XTOLS library internally uses and provides to user programmers. Figure 5 shows the actual type declarations for XTOLS that correspond to various label components and labels. Type names prefixed with OLS are record types that truly reflects the schemas used by OLS, and ones with XTOLS are converted (normalized) forms. For example, the type XTOLSDataLabel consists of three attributes: `policy_name`, `label`, and `label_tag`. Unlike its dual type OLSDataLabel, it has not only the label tag but also a label, which is a 3-tuple of label components—level, compartments, and groups. This helps XTOLS process labels and label components efficiently because we do not have to compute the label based on the tag every time a label needs to be evaluated.

Second, we provide functions to convert OLS types into some XTOLS internal representation—that is, from OLS types to XTOLS types. As already discussed in Section 2.3.2, all of the data labels and the user label are normalized for convenience. Of course, we assume that SELINKS has full privileges on the necessary tables that contain OLS information. Finally, we provide policy functions for reading and writing the labeled data. XTOLS library package consists of many policy functions but user programs can only access `read` and `write` functions whereas policy programs may invoke any functions for extended security policies. Of course, it is the policy writer’s design decision to extend these or even prohibit user code from invoking these functions directly.

## 4. Experimental Results

We ran our implementation on the same test case and query used in [4] with the same configuration—Intel Quad Core Xeon 2.66 GHz with 4 GB of RAM running Red Hat Enterprise Linux AS 4. The DBMS used is Oracle 11g Release 1 with Oracle Label Security component installed. The test case was automatically generated 1,000 rows, each of which has an associated label. Since we are using OLS, not the user defined types, we had to define the seven possible labels used in the test case in OLS (recall that all labels must be defined prior to use). We also had to change the actual labels used in the test case to be numbers, instead, to refer to the actual label. Figure 6 shows the results of our experiment with the fourth column showing the results using UDFs.

```

1 # types used for native OLS and XTOLS
2 typename Level = (policy_name:String, level_num: Int,
3                 short_name:String, long_name:String);
4 typename Comp = (policy_name:String, comp_num: Int,
5                 short_name:String, long_name:String);
6 typename Group = ( policy_name:String, group_num: Int,
7                   short_name:String, long_name:String,
8                   parent_num: Int, parent_name:String
9                 );
10
11 # types used for XTOLS
12 typename XTOLSLabel =
13   ( level :Level, comps:[Comp], groups:[Group]);
14 typename XTOLSDataLabel =
15   (policy_name:String, label :OSLabel, label_tag : Int );
16 typename XTOLSLabelUser = (
17   user_name:String,
18   policy_name:String, max_read:OSLabel, max_write:OSLabel,
19   min_write:OSLabel,
20   def_read :OSLabel, def_write :OSLabel, def_row:OSLabel
21 );
22 typename XTOLSLabelUserPriv = (
23   user_name:String, policy_name:String,
24   read:Bool, full :Bool, comp_access:Bool, profile_access : Bool,
25   write_up:Bool, write_down:Bool, write_across :Bool
26 );
27 typename XTOLSLabelUserPrivInfo =
28   ( label :OSLabel, privilege :OSLabelPriv);
29 typename XTOLSLabelUser = (l<-OSLabelUserPriv, String{1});
30 typename TupleOfComps =
31   ( levels :[Level], comps:[Comp], groups:[Group]);
32
33 # types used for native OLS
34 typename OLSDataLabel =
35   (policy_name:String, label :String, label_tag : Int );
36 typename OLSLabelUser = (
37   user_name:String, policy_name:String,
38   label1 :String, label2 :String, label3 :String,
39   label4 :String, label5 :String, label6 :String
40 );
41 typename OLSLabelUserPriv =
42   (user_name:String, policy_name:String, user_privileges :String);
43 typename OLSLabeledRow(a) = (label.col: Int | a);

```

**Figure 5.** Types defined in SELINKS for labels and label components.

Surprisingly, using XTOLS was much slower than using UDFs. Our preliminary interpretation of the results is that SELINKS parsing phase is very slow; and therefore, parsing all the library functions in XTOLS dramatically increases the execution time. We believe that once SELINKS supports more advanced modular programming interface, this problem will eventually disappear. Another interesting observation is that running the actual XTOLS code turned out to be a really slow task as well. The last row of the results table shows the time it took to retrieve the data and not read it, which turned out to be approximately four times faster. This is, however, one of the consequences that XTOLS users must face in order to satisfy the safety guarantee.

Tiers	Operations	XTOLS(s)	UDFs(s)
Server	Query & Unlabel	5.036	0.15
DB & Server	Query & Unlabel	2.764	0.04
DB & Server	Query	0.726	N/A

**Figure 6.** Experimental Results (means of 5 runs)

We estimated that most of these performance issues would go away once the test case is very large. Unfortunately, the second test case used in [4], which consists of 100,000 rows, causes current SELINKS implementation to crash with a stack overflow exception. We are, however, optimistic that the results will be much better in near future as SELINKS support modular programming and larger memory space.

## 5. Conclusion and Future Work

This paper presented XTOLS, an extension of SELINKS that allows enforcing OLS security policies in multiple tiers. While the previous approach using UDFs generalizes the cross-tier security enforcement for all databases, it is not able to incorporate existing security policies such as OLS. XTOLS architecture is designed to provide a reasonable programming interface to SELINKS programmers and to guarantee the safety property for applications written in SELINKS that need to interact with OLS. We have shown that it is not difficult to encode OLS in SELINKS, but it requires some language support such as syntax changes and table checker. The results also show that the performance may be an issue, but we believe most of the causes are from the LINKS parser and its inability to support modular programming. Despite the drawbacks, XTOLS guarantees the safety guarantee across the different tiers of the system. We believe that, in near future, the performance and reliability issues will eventually fade away as SELINKS becomes a more robust framework that supports native modular programming and new language constructs for OLS.

It is also worth mentioning that encoding more features of OLS in SELINKS is an interesting future work on this topic. For example, it is possible to associate each database table with a UDF (or predicate) in addition to a label column (this is how VPD works as well). The UDF will be invoked as a query that is processed on a particular table, taking into account the user context of current session. We can encode such policies in SELINKS as a function that takes the user profile as an argument and returns a boolean. This is possible since SELINKS is a functional language—i.e., it supports first-class functions.

## References

- [1] Oracle label security. <http://www.oracle.com/database/label-security.html>.
- [2] Virtual private database. <http://www.oracle.com/technology/deploy/security/security/virtual-private-database/index.html>.
- [3] E. D. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation, 1976. MITRE Corporation.
- [4] B. J. Corcoran, N. Swamy, and M. Hicks. Cross-tier, label-based security enforcement for web applications. *ACM SIGMOD*, June 2009. To appear.
- [5] G. S. Graham and P. J. Denning. Protection—principles and practices. In *Proceedings of the Spring Joint Computer Conference*. AFIPS Press, 1972.
- [6] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Science and Systems*, volume 8. ACM Operating Systems Review, January 1974.
- [7] T. F. Lunt and E. B. Fernandez. Database security. *ACM SIGMOD RECORD*, 19(4), December 1990.
- [8] Oracle. Oracle label security: Best practices for government and defense applications. *An Oracle White Paper*, June 2007.
- [9] Oracle Corporation. *Oracle Label Security: Administrator's Guide*, 10g release 1 (10.1) edition. Part No. B10774-01.
- [10] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *2008 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2008.