# Runtime Values Driven by Access Control Policies

## Statically Enforced at the Level of Relational Business Tiers

Óscar Mortágua Pereira[1], Rui L. Aguiar[2]
Instituto de Telecomunicações
DETI, University of Aveiro
Aveiro, Portugal
{omp[1], ruilaa[2]}@ua.pt

Maribel Yasmina Santos
Centro Algoritmi
DSI, University of Minho
Guimarães, Portugal
maribel@dsi.uminho.pt

***Abstract*—Access control is a key challenge in software engineering, especially in relational database applications. Current access control techniques are based on additional security layers designed by security experts. These additional security layers do not take into account the necessary business logic leading to a separation between business tiers and access control mechanisms. Moreover, business tiers are built from commercial tools (ex: Hibernate, JDBC, ODBC, LINQ), which are not tailored to deal with security aspects. To overcome this situation several proposals have been presented. In spite of their relevance, they do not support the enforcement of access control policies at the level of the runtime values that are used to interact with protected data. Runtime values are critical entities because they play a key role in the process of defining which data is accessed. In this paper, we present a general technique for static checking, at the business tier level, the runtime values that are used to interact with databases and in accordance with the established access control policies. The technique is applicable to CRUD (create, read, update and delete) expressions and also to actions (update and insert) that are executed on data retrieved by Select expressions. A proof of concept is also presented. It uses an access control platform previously developed, which lacks the key issue of this paper. The collected results show that the presented approach is an effective solution to enforce access control policies at the level of runtime values that are used to interact with data residing in relational databases.**

***Keywords-security; access control; database, business tiers; software architecture.***

## I. INTRODUCTION

Sensitive data is growing every day as an immediate consequence of the increasing usage of software systems. The data is related not only to personal information, as it happens for example in social networks, but it is also related to other important and critical areas such as commercial, institutional and security organizations. To prevent any security violation, several security measures are taken such as user authentication, data encryption and secure connections. Another relevant security concern is access control. There are two main approaches to enforce access control policies: the one provided by vendors of database management systems and XACML [1] (eXtensible Access Control Markup Language). Both approaches rely on additional security layers built by security experts leading to a clear separation between the security mechanisms and business tiers. Moreover, current commercial tools that are used to develop business tiers do not support access control policies, this way hampering the process of bridging the gap between access control mechanisms and business tiers built from those tools. To overcome this situation, several access control techniques have been proposed [2-13] but none of them effectively models the values that are defined at runtime. The runtime values are critical because they are dynamically defined by users at runtime, this way enabling users to request the access to different data in each execution cycle. We present three examples to justify our claims. The first one is based on a native Select expression, the second one is based on a native Update expression and, finally, the third one is based on modifying the contents of a record set containing data retrieved by a Select expression (in these cases the modifications are also committed to the host database). The following example is a simple Select expression.

```
Select t1.* from table1 t1, table2 t2
      where t1.id = t2.t1_id and
            t1.value > pValue
```

The parameter (runtime value) *pValue* plays a key role to decide which data are retrieved from *table1*. In each individual execution cycle, the parameter may have a different value, this way retrieving a different set of records from *table1*. To overcome this source of possible security gaps, two approaches are used to implement the access control mechanisms: centralized approach and distributed approach. Regarding the centralized approach, the most common technique is the use of views (with [10] or without query rewriting techniques). This technique conveys several drawbacks among which the lack of scalability is emphasized [14, 15]. Regarding the distributed approach, two techniques were proposed: in [4] is proposed a new predicate, identified as *known*, to model which information users already know, this way covering the points here under discussion but only superficially; in [2] the policies are statically enforced at the table columns level and not at the CRUD (Create, Read, Update and Delete) expressions level, leading to lack of flexibility.

The following example is the second example, which is a simple Update expression:

```
Update table1 t1 set t1.value=pValue
       Where t1.id=pId
```

Similar to the Select expression, this Update expression also uses parameters. The parameter *pValue* updates the attribute *value* of *table1* of a record identified by another parameter *pId*. Once again, parameters are user defined and play a key role on Update expressions to decide the data to be updated. The current techniques and their limitations, previously described for Select expressions, are also applied to Update expressions. The remaining types of CRUD expressions, Insert and Delete, convey similar limitations.

The last example is a very common situation on current tools that are used to develop business tiers, such as JDBC [16], Hibernate [17], ADO.NET [18] and LINQ [19]. The example shows that beyond the use of CRUD expressions, databases are also modifiable by executing protocols on data retrieved by Select expressions. The example shows that after retrieving data from a database, it is kept in record sets (*recordSet*) and then applications are allowed to update their content through an update protocol. In this case the attribute *attributeName* was updated to *value* and then the modification was committed. This case is different from the two previous ones because there is no evidence of any CRUD expression and users are modifying data they have been previously authorized to retrieve. Even so, we cannot despise the need to control the runtime values being used to modify the contents of those record sets and, therefore, used to modify the contents of databases. Beyond the update protocol, current tools also provide an insert protocol where users are also allowed to use runtime values.

```
recordSet=executeSelectExpression(sql)
recordSet.update("attributeName", value)
recordsSet.commit()
```

Currently, there isn´t any known access control technique to enforce policies at the business tier level and able to statically control the provenance of runtime values that are used on actions issued against databases. To overcome this situation we propose a technique where parameters are statically driven by access control policies enforced at the business tier level. Additionally, we present a proof of concept to validate the proposed technique. The proof of concept leverages an existent and internal access control platform, partially based on [13].

This paper is organized as follows. Section II presents the related work. Section III presents the required background to keep the paper self-contained. Section IV describes the conceptual architecture and, finally, section V presents the final conclusion.

## II. RELATED WORK

Views have been widely used to restrict the access to protected data. In spite of their relevance, the use of views to implement access control conveys a key drawback: lack of scalability [14, 15]. Basically the number of views increases with the number of policies. Access control based on views is easily managed in database applications with a short number of policies. But access control in database applications with a large number of policies may become unmanageable as in cases where they depend, for example, on data stored on databases. Moreover, the problem is not restricted to the level of views. Users accessing the same table but with different authorization levels use different views and, therefore, different CRUD expressions. In order to minimize this scalability gap, Rizvi et al. [10] present a query rewriting technique to determine at runtime if a CRUD expression is authorized, without the need of creating different versions of views. It uses security views to filter contents of tables and simultaneously to infer and check at runtime the appropriate authorization to execute any CRUD expression issued against the unfiltered table. The user is responsible for formulating the CRUD expression properly. They call this approach the Non-Truman model. Non-Truman models, unlike Truman models, do not change the original CRUD expressions. The process is transparent for users, and CRUD expressions are rejected if they do not have the appropriate authorization. This approach has some disadvantages: 1) performance - the inference rules to check the appropriate authorization at runtime are complex and time consuming; 2) productivity - authorizations are checked against security views and not against original data in a transparent way, hampering the debugging process when any syntax error or security violation occurs; 3) awareness - programmers cannot statically check the correctness of CRUD expressions because the policies and the mechanisms are centralized in a server; 4) incompleteness - the inference rules are complex and their completeness is not assured by the authors.

In [4], Chlipala et al. present a tool, *Ur/Web*, that allows programmers to write statically-checkable access control policies as CRUD expressions. Basically, each policy determines which data is accessible. Then, programs are written and checked to assure that data involved in queries is accessible through some policy. To allow policies to vary from one user to another, their CRUD expressions use actual data and a new extension to the standard SQL to capture '*which secrets the user knows'*. This extension is based on a predicate referred to as *´known*' used to model which information users are already aware of to decide upon the information to be disclosed. Ur/Web is a promising solution, but beyond introducing a new programming technique, it presents two key drawbacks: 1) it does not check the use of runtime values of *where* clauses, allowing queries to implicitly leak protected data; 2) authors say that their implementation "…*only handles a subset of the common SQL features.*".

Caires et al. [2] introduces a new programming language, name as λ, to define and enforce access control policies by static typing. The security model comprises tables, their attributes and the access control policies associated to each attribute. Authors show that runtime values are checked against the policies before being used. Beyond introducing a new programming language, policies are enforced at the attribute level of tables, this way hindering or even preventing the use of multiple policies on each attribute.

The paper [13] presents an access control-driven architecture with dynamic adaptation (ACADA). Business tiers are automatically built from a business architectural model, enforcing access control policies defined by a security expert. Access control mechanisms are statically

implemented by typed objects driven by security policies at the business tier level. ACADA effectively controls which CRUD expressions are authorized to be used but does not control the runtime values being used.

## III. BACKGROUND

To ease the development process of business tiers, system architects use tools specially designed to that end. Two main groups of tools are considered: Call Level Interfaces (CLI) [20] and Object-to-Relational Mapping (O/RM) tools. ODBC [21], JDBC [16] and ADO.NET [22] are three examples of CLI and Hibernate [17], LINQ [23] and JPA [24] are three examples of O/RM tools. These tools provide services to allow applications to interact with databases. These services need to be understood before advancing to any security solution implemented at the business tier level. In spite of the diversity of tools and the difference between the paradigms of the two groups, there is a common basis between them. This is very important to promote the use of a single technique in all tools and mainly on both groups. The common basis is centered on their two main access modes to stored data: direct access mode and the indirect access mode. The direct access mode allows the execution of CRUD expressions written in the native SQL language and the indirect access mode allows applications to interact with data returned by Select expressions. While the direct access mode is widely used and easily understood, the indirect access mode needs a more detailed explanation. When a Select expression is executed, it returns a relation containing the retrieved data. These relations are locally managed by local memory structures (LMS). There are four protocols to interact with the data managed by LMS: read protocol (to read data from LMS), update protocol (to update data contained in LMS), insert protocol (to insert new data in LMS) and delete protocol (to delete data contained in LMS). Any modification on the contents of LMS is replicated on the host database. Figure 1 and Figure 2 depict a simple example based on JDBC and LINQ, respectively. The method *updateStudentMobilePhone* updates numbers of mobile phones of every student whose *id* is contained in the first argument (*sId*). The Select expression is built with two parameters (line 29-31, 116-118) and executed (line 32, 119-120) through the direct access mode (*rs.executeQuery* and *jpa.ExecuteQuery*). Then LMS (*rs* (ResultSet [25]) for JDBC and *ord* (typed object) for LINQ) are iterated row by row (line 33, 121). *mobilePhone* is updated (line 36-37, 126-127) if the student *id* (*rs.getInt* and *s.id*) is contained in the list *sId* (line 34-35, 123-124) through the indirect access mode. This update on the LMS is equivalent to the following Update expression

```
Update Student s
       Set s.mobilePhone=mobilePhone
       Where s.id=sId(idx)
```

and, therefore, *sId* and *modiblePhone* in Figure 1 and Figure 2 behave as runtime values for the two parameters of the equivalent Update expression. From this example it is also easily inferred the equivalency between the insert and delete protocols and the correspondent Insert and Delete

```
26  void updateStudentMobilePhone(List<Integer> sId,
27                   List<String> mobilePhone)
28                   throws SQLException {
29    String sql="Select * from dbo.Student " +
30               "where id between "+
31                  max(sId)+" and " + min(sId);
32    rs=st.executeQuery(sql);
33    while (rs.next()) {
34      int idx=sId.indexOf(rs.getInt(1));
35      if (idx!=-1) {
36          rs.updateString(4,mobilePhone.get(idx));
37          rs.updateRow();
38      }
39    }
40  }
```

Figure 1. Example based on JDBC.

```
113 void updateStudentMobilePhone(Collection<int> sId,
114                   Collection<string> mobilePhone)
115 {
116   string sql="Select * from dbo.Student " +
117              "where id between "+
118                    sId.Max()+" and "+sId.Min();
119   IEnumerable<Student> student =
120                   jpa.ExecuteQuery<Student>(sql);
121   foreach (Student s in student)
122   {
123     int idx = sId.IndexOf(s.id);
124     if (idx != -1)
125     {
126        s.mobilePhone = mobilePhone.ElementAt(idx);
127        jpa.SubmitChanges();
128     }
129   }
130 }
```

Figure 2. Example based on LINQ.

expressions. These two simple examples have shown the usage of the two access modes that are provided by current tools and also the usage of runtime values. Additionally, the examples also show that JDBC and LINQ, akin to the remaining tools, are not driven by access control policies. Their access modes allow programmers to write any CRUD expression (using the direct access mode) and also allow the use of any protocol on LMS. These latter two issues have been addressed in [13].

## IV. ARCHITECTURE PRESENTATION

In this section we present an access control technique which enforces policies at the business tier level which is able to statically control the provenance of runtime values that are used on actions issued against databases. The technique supervises the runtime values that are used on both access modes of current tools that are used for developing business tiers. Nevertheless, access control policies can only be effectively enforced if other complementary aspects are also considered. Among them the authorized CRUD expressions and the actions on LMS are emphasized. Those aspects are not addressed in this paper because they were already addressed in [13]. From [13], a platform has been designed and developed. The platform will be used and modified to present the proof of concept. This section is organized as follows: the sub-section *A* presents the proposed technique; sub-section *B* briefly presents the used platform;

sub-section *C* presents the proof of concept and, finally, subsection *D* presents a use case.

### *A. Proposed Technique*

We start by introducing the concept of Business Access Point (BAP). A BAP is an entity responsible for managing the runtime values of the two access modes in accordance with the established access control policies. Each access mode type has its own particular characteristics. As such, their conceptual architecture is presented separately.

Direct Access Mode

The direct access mode allows the execution of CRUD expressions based on the native SQL language. In a general context, each CRUD expression comprises a hard coded part and eventually one or more parameters of which the values are defined at runtime. The values for these parameters are not mandatory to be driven by any access control policy. It is up to the security expert to decide for each CRUD expression which parameters are driven by access control policies and which parameters are not driven by any access control policy. Thus, the direct access mode (DAM) is formalized by the next triplet:

$$DAM(\ RTV, RTV_{acp}, execute)$$

where *RTV* is a set of RunTime Values for parameters not driven by any access control policy, $RTV_{acp}$ is a set of RunTime Values for the parameters driven by access control policies and, finally, *execute* is a method responsible for setting the runtime values for parameters and also for the execution of CRUD expressions. As initially announced, $RTV_{acp}$ are statically enforced and, therefore, their implementation will have this in consideration. Eventually, each runtime value may be encapsulated as an interface that provides a service aimed at returning values driven by access control policies.

Indirect Access Mode

The indirect access mode provides four protocols for the interaction with the data contained by LMS that is returned by native Select expressions. A first solution has been proposed to provide the four protocols driven by access control policies [13]. Basically, it includes two aspects: 1) the availability of each protocol is individually configured and 2) each protocol that is made available provides methods to access only the attributes that are authorized by the established policies. This approach is not complete because it does not support parameters driven access control policies. Next follows the proposed approach to overcome this security gap. The indirect access mode (IAM) is formalized as follows:

$$IAM(readP, insertP, updateP, deleteP)$$

where *readP* is the read protocol, *insertP* is the insert protocol, *updateP* is the update protocol and, finally, *deleteP* is the delete protocol. Only the insert and the update protocols use runtime values. The read protocol does not modify the contents of LMS and the delete protocol is executed as an atomic operation on all attributes of the selected row. Thus, each individual method of the insert and update protocol that is used to modify each attribute of the returned relation (contained in LMS) needs to be configured to be or not to be driven by access control policies. They are formalized as:

$$method(RTV) \text{ or } method(RTV_{acp})$$

where *method* is the method's name, *RTV* and $RTV_{acp}$ have the meaning previously presented for the direct access mode. The only difference is that either *RTV* or $RTV_{acp}$ represent a single runtime value. The indirect access mode is only available after a Select expression is executed through the direct access mode. The remaining CRUD expressions do not create LMS. This leads to the need of defining two facets for the BAP: one for the Select expressions ($BAP_s$) and another for the remaining expressions ($BAP_{iud}$). $BAP_{iud}$ supports the direct access mode only and is formalized as follows:

$$BAP_{iud}(DAM)$$

$BAP_s$ supports both modes and is formalized as follows:

$$BAP_s(DAM,IAM)$$

### *B. Used Platform*

The proof of concept here presented leverages the work previously presented in [13]. The work has been used to design a new architecture known as DACA (Dynamic Access Control Architecture). Figure 3 presents a simplified block diagram of DACA. DACA is able to dynamically, at runtime, build and keep updated business logic of relational database applications in accordance with the established access control policies. It comprises 2 main components: a client side component for the application and business tiers and a server side component where metadata of access control mechanisms are kept. The basic operation of DACA is as follows (see Figure 3): 1- application tier instantiates a Dynamic Access Control Component (DACC); 2- DACC, through the Business Manager, establishes a connection with the Policy Server; 3- The Policy Server transfers and keeps security metadata and CRUD expressions continuously updated on DACC, in accordance with the established access control policies; 4- DACC, through the Business Manager, dynamically builds and keeps business logic updated; 5- application tiers ask Business Manager to execute authorized CRUD expressions; 6- Business Manager delegates the execution of CRUD expressions on the implemented

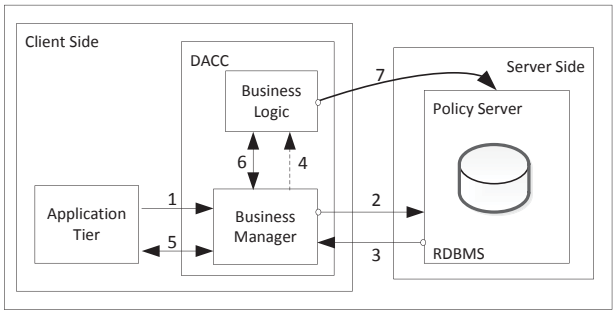


Figure 3. Simplified block diagram of DACA.

Business Logic; 7- CRUD expressions are executed (the RDBMS server may or may not be the same as the one responsible for the Policy Server).

### C. Proof of Concept

The initial version of DACA was redesigned to address the issues of this research and it is hereafter known as RDACA (Redesigned-DACA). We have decided that the policy to be followed for $RTV_{acp}$ requires that the values can only come from data previously retrieved by authorized Select expressions. To address this new security requirement the original DACA security access control mechanisms were redesigned. To give a complete view of the implemented solution, class diagrams of BAP will be provided.

The client-side of RDACA was implemented in Java and JDBC and, therefore, all examples are based on those tools. In the RDACA each $RTV_{acp}$ is defined as an interface comprising a unique method which is responsible for retrieving the authorized value. The proposed approach, as it will be shown, allows a static validation for all $RTV_{acp}$ at development time.

Figure 4, Figure 5 and Figure 6 present simplified class diagrams for the approach followed for the BAP to enforce access control policies. In a first step, one interface is defined for each individual $RTV_{acp}$ as shown in Figure 4: IRTV_a, IRTV_b, …, IRTV_n. Each interface is related to a unique $RTV_{acp}$ and it comprises also one unique method responsible for ensuring that the values are effectively authorized. *rA*, *rB*, …, *rN* are the defined methods and *DT_a*, *DT_b* and *DT_n* are the data types of the $RTV_{acp}$ in the host programming language. The concrete implementation of each method depends on the adopted security strategy. In case of the RDACA, these methods retrieve data from data previously retrieved by authorized Select expressions and also managed by $BAP_s$.

Figure 5 presents a simplified class diagram for one $BAP_s$. The constructor of the base class, BAPs, receives a connection to the database and the CRUD *id* to be executed. Programmers do not write CRUD expressions anymore. They are only allowed to select, though the CRUD *id*, which CRUD expression is necessary. In case she is not authorized to use the requested CRUD expression, an exception will be raised. Other important aspects are the *IExecute* and the *ILMS* interfaces. *IExecute* is associated with the direct access mode and ILMS is associated with the indirect access mode. *IExecute* comprises one unique method (*execute)*. It accepts as arguments RTV and $RTV_{acp}$ for the runtime values of the clause conditions for the Select expression to be executed. In this particular case, it accepts an RTV of type DT_a and an $RTV_{acp}$ of type IRTV_b. Thus, to execute the requested Select expression it is necessary to be a holder of an $BAP_s$ providing an IRTV_b. Regarding the *ILMS* interface, it comprises several interfaces being *IRead* and *IUpdate* presented with some detail. They are enough to convey a complete understanding about the followed approach. *IRead* implements the read protocol on LMS providing all the necessary methods to that end. Each method retrieves the value of one attribute of the returned relation. There are two types of methods: one type retrieves values that can only be used as RTV and the other type retrieves values that can be used as $RTV_{acp}$. Methods retrieving RTV are directly defined in the *IRead* interface, such as *rB* and *rC* as shown in Figure 5. Methods retrieving $RTV_{acp}$ are defined by extending *IRead* with the interfaces that provide $RTV_{acp}$, see Figure 4. The shown IRead interface provides two methods for RTV (rB and rC) and one interface for one $RTV_{acp}$ (IRTV_a). This distinction allows Business Manager (see Figure 3), by analyzing the schema, to be able to distinguish between RTV from $RTV_{acp}$ and, therefore, to provide, during the automatic building process of Business Logics, different implementations for the two types of methods. Regarding the *IUpdate* interface it is associated with the update protocol. In this particular case it comprises two methods: a) *uA* updates the attribute *a* and it accepts an $RTV_{acp}$ (IRTV_a); b) *uB* updates the attribute *b* and it accepts any RTV of type *DT_b*.

Figure 6 presents a simplified class diagram for one $BAP_{iud}$. The description for the base class and also for the *IExecute* interface is identical to the previous $BAP_s$. Regarding *ISet*, it comprises one unique method (*set*), which accepts as arguments RTV and $RTV_{acp}$ for the runtime values of the column list of the Update expressions. In t his case it accepts two $RTV_{acp}$ and one RTV. Thus, to be able to use

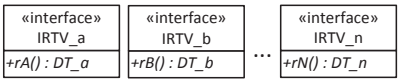


Figure 4. Set of $RTV_{acp}$.

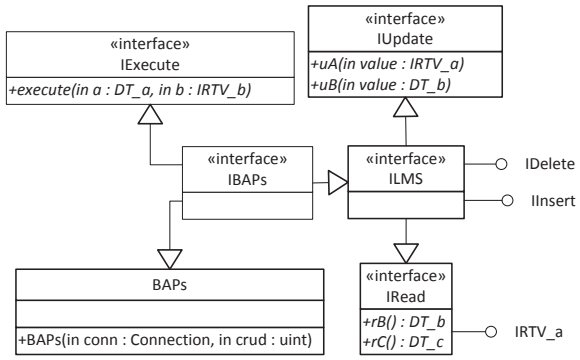


Figure 5. Simplified class diagram for a concrete $BAP_s$.

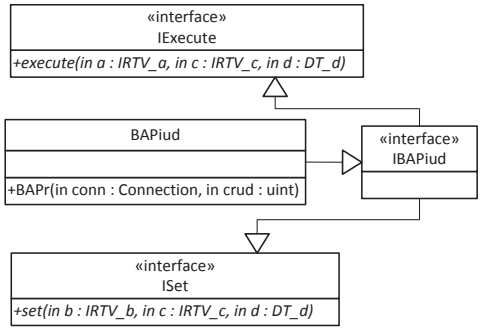


Figure 6. Simplified class diagram for a $BAP_{iud}$.

this $BAP_{iud}$ it is required to be authorized to execute the required CRUD expressions and to hold three $RTV_{acp}$ (IRTV_a, IRTV_B and IRTV_c) provided by one or more $BAP_s$.

### *D. Use Case*

We are now prepared to present a real use case implemented with Java, JDBC and Microsoft Northwind database[1]. The use case is based on an actor responsible for managing orders coming from customers in the USA only. The actor is authorized to execute the two following CRUD expressions:

```
Select * from Customers
        where customerId=?          // (RTV)
              and Country='USA'
Select * from Orders
         Where CustomerId=?         // (RTVacp)
               and ShipCountry=?    // (RTV)
```

The first Select expression allows the access to information about the customers residing in the USA and the second Select expression allows the access to orders only from customers the user is authorized to know ($RTV_{acp}$ – in this case residing in USA) and whose ship county is user defined (RTV). The $BAP_s$ associated with the latter Select expression is updatable and one particularity is that the attribute *employeeId* requires an $RTV_{acp}$ when using the indirect access mode.

To address this case, two $BAP_s$ are needed, one for each CRUD expression. We have used the table names to identify each $BAP_s$, *Customers* and *Orders*. From the two Select expressions we see that, when using the direct access mode, the second one requires an $RTV_{acp}$ for the first parameter - *CustomerId*. Figure 7 shows an example of how the two $BAP_s$ (Customers and Orders) may be used. A new instance of Customers is created (line 30) and the CRUD expression is executed (line 31) to select data about the customer identified by the RTV of *customerId*. Then, the first and only row of the LMS (rs) is selected (line 32). Some attributes are read (line 33-34). Then an instance of Orders is created (line 35) and the CRUD is executed (line 36). The CRUD has two parameters, the first one is an $RTV_{acp}$ and the second one is an RTV. The $RTV_{acp}$ is for *customerId* and it is passed as the instance of Customers, which implements the required interface for the $RTV_{acp}$. The ship country is an RTV and, therefore, it is user defined. Some attributes are read (line 37-38) and the programmer tries to update *employeeId* but the NetBeans indicates an error because the correct data type cannot be an *integer* (line 39). To update *employeeId* through the indirect access mode the programmer needs an $RTV_{acp}$ of the required type. To convey a deeper understanding some additional details are provided for the two $BAP_s$. Figure 8 shows the interface herein named as *ICustomerId* for the $RTV_{acp}$ *customerId*. This interface is used not only to be implemented by $BAP_s$ but also used whenever identifications of customers need to be used as $RTV_{acp}$ for arguments of BAP methods, as shown in Figure 7. The implementation of

[1] http://www.microsoft.com/en-us/download/details.aspx?id=23654

```
28  private void go(Connection conn, int customerId,
29                  String shipCountry)throws Exception{
30      Customers c=new Customers(conn,crudCustomersId);
31      c.execute(customerId);
32      if (c.moveNext()) {
33          companyName=c.rCompanyName();
34          //... read more attributes
35          Orders o=new Orders(conn,crudOrdersId);
36          o.execute(c,shipCountry);
37          orderId=o.rOrderId();
38          // ... read more attributes
            o.uEmployeeId(5);
40          //... more code
```

Figure 7. Example to show the use of the two $BAP_s$: Customers and Orders.

```
    public interface ICustomerId {
        int rCustomerId() throws SQLException;
15  }
```

Figure 8. Interface for the $RTV_{acp}$ to be used for the parameter CustomerId.

this interface should comprise some validation procedures to prevent its misuse. As previously explained, BusinessManager automatically generates the required source code for Business Logic. In this particular case, it creates the required source code for *rCustomerId()* in accordance with the established security requirements.

The IRead interface for *Customers* is presented in Figure 9. It provides a set of methods to read the attributes of the returned relation. *CustomerId* is the only attribute with the ability to be used as an $RTV_{acp}$ and, therefore, the IRead interface extends the *ICustomerId* interface.

Figure 10 shows the *IExecute* interface for the $BAP_s$ *Orders*. It comprises two arguments. The first argument is an $RTV_{acp}$ for *customerId* and, therefore, it requires the correspondent interface (*ICustomerId*). The second argument is an RTV for the ship country. Figure 11 presents its implementation in which a main aspect is emphasized. The $RTV_{acp}$ (*customerId*) is passed as an interface (line 28) and

```
    public interface IRead extends ICustomerId {
        String rCompanyName() throws SQLException;
        String rContactName() throws SQLException;
16      //... other attributes
```

Figure 9. IRead interface for Customers.

```
public interface IExecute {
    void execute(ICustomerId customerId,
16                 String shipCountry ) throws SQLException;
17  }
```

Figure 10. IExecute interface for Orders.

```
    @Override
28  public void execute(ICustomerId customerId,
29                      String shipCountry)
30                      throws SQLException {
31      ps=conn.prepareStatement(crud);
32      ps.setInt(1,customerId.rCustomerId());
33      ps.setString(2,shipCountry);
34      rs=ps.executeQuery();
35  }
```

Figure 11. *execute* method implementation of Orders.

the run time value (line 32) is obtained from the method specifically created for the effect and defined in the *ICustomerId* interface.

There is a runnable demo available at https://dl.dropboxusercontent.com/u/71192544/Work/Confers/SEKE/SEKE_2013/Example.7z .

## V. CONCLUSION

This paper presents a technique aimed at enforcing access control policies statically at the level of the runtime values that are used on business tiers to interact with data stored on relational database management systems. The technique is applicable to commercial tools geared up to develop business tiers, such as JDBC, ODBC, Hibernate and LINQ, and supports their two most common access modes: the direct and the indirect access mode. Security experts are able to decide the policies to be used, which runtime values are driven by those policies and which are not. Runtime values driven by access control policies are managed at the business tier level to ensure the use of authorized values only. The presented proof of concept is based on an existent platform that has been redesigned to support a new security requirement. The new security requirement says that only previously retrieved values from the database are allowed to be used for the runtime values driven by access control policies. The implemented technique is based on interfaces comprising a unique method of which the implementation ensures the new security requirement. Beyond the presented proof of concept a runnable demo is also available.

It is expected that the outcome of this research will have impact on future proposals addressing access control on relational databases, mainly when policies are enforced at the level of client business tiers.

As future work, we intend to apply the techniques used in [26, 27] to design a thread-safe version of DACC. These techniques have proved to be not only simple to implement but above all conveying a significant performance improvement.

## REFERENCES


[1] OASIS. "XACML - eXtensible Access Control Markup Language," Feb, 2012; http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[2] L. Caires, J. A. Pérez, J. C. Seco et al., "Type-based access control in data-centric systems," in 20th European conference on Programming Languages and Systems: part of the joint European conferences on theory and practice of software, Saarbrucken, Germany, 2011, pp. 136-155.

[3] S. Chaudhuri, T. Dutta, and S. Sudarshan, "Fine Grained Authorization Through Predicated Grants," in IEEE 23rd ICDE - Int. Conf. on Data Engineering, Istanbul, Turkey, 2007, pp. 1174-1183.

[4] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in 9th USENIX Conf. on Operating Systems Design and Implementation, Vancouver, BC, Canada, 2010, pp. 1-14.

[5] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, Label-based Security Enforcement for Web Applications," in 35th SIGMOD Int. Conf. on Management of Data, Providence, Rhode Island, USA, 2009, pp. 269-282.

[6] J. Fischer, D. Marino, R. Majumdar et al., "Fine-Grained Access Control with Object-Sensitive Roles," in 23rd ECOOP - European Conference on Object-Oriented Programming, Italy, 2009, pp. 173-194.

[7] Q. Wang, T. Yu, N. Li et al., "On the correctness criteria of fine-grained access control in relational databases," in 33rd Int. Conf. on Very Large Data Bases, Vienna, Austria, 2007, pp. 555-566.

[8] W. Gary, G. Carl, S. Zhendong et al., "Static checking of dynamically generated queries in database applications," ACM Transansactions on Software Eng. Methodology, vol. 16, no. 4, pp. 14:01-14:27, 2007, doi: http://doi.acm.org/10.1145/1276933.1276935.

[9] B. Hicks, S. Rueda, D. King et al., "An architecture for enforcing end-to-end access control over web applications," in 15th ACM symposium on Access Control Models and Technologies, Pittsburgh, Pennsylvania, USA, 2010, pp. 163-172.

[10] S. Rizvi, A. Mendelzon, S. Sudarshan et al., "Extending Query Rewriting Techniques for Fine-grained Access Control," in ACM SIGMOD Int. Conf. on Management of Data, Paris, France, 2004, pp. 551-562.

[11] K. LeFevre, R. Agrawal, V. Ercegovac et al., "Limiting disclosure in hippocratic databases," in 30th Int. Conf. on Very Large Databases, Toronto, Canada, 2004, pp. 108-119.

[12] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," SIGPLAN Not., vol. 47, no. 1, pp. 85-96, 2012, doi: 10.1145/2103621.2103669.

[13] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, " ACADA - Access Control-driven Architecture with Dynamic Adaptation," in SEKE - 24th Intl. Conf. on Software Engineering and Knowledge Engineering, San Francisco, CA, USA, 2012, pp. 387-393.

[14] M. I. Y. d. Valle, A. Mana, J. Lopez et al., "Secure Content Distribution for Digital Libraries," in Proceedings of the 5th International Conference on Asian Digital Libraries: Digital Libraries: People, Knowledge, and Technology, 2002, pp. 483-494.

[15] J. Lopez, A. Mana, and M. I. Y. d. Valle, "XML-Based Distributed Access Control System," in Proceedings of the Third International Conference on E-Commerce and Web Technologies, 2002, pp. 203-213.

[16] M. Parsian, JDBC Recipes: A Problem-Solution Approach, NY, USA: Apress, 2005.

[17] B. Christian, and K. Gavin, Hibernate in Action: Manning Publications Co., 2004.

[18] C. Pablo, M. Sergey, and A. Atul, "ADO.NET entity framework: raising the level of abstraction in data programming," in ACM SIGMOD International Conference on Management of Data, Beijing,China, 2007, pp. 1070-1072.

[19] M. Erik, B. Brian, and B. Gavin, "LINQ: Reconciling Object, Relations and XML in the .NET framework," in ACM SIGMOD Intl Conf on Management of Data, Chicago,IL,USA, 2006, pp. 706-706.

[20] ISO. "ISO/IEC 9075-3:2003," [2011 May; http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134.

[21] Microsoft. "Microsoft Open Database Connectivity," Jul, 2012; http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx.

[22] G. Mead, and A. Boehm, ADO.NET 4 Database Programming with C# 2010, USA: Mike Murach & Associates, Inc., 2011.

[23] D. Kulkarni, L. Bolognese, M. Warren et al., "LINQ to SQL: .NET Language-Integrated Query for Relational Data," Microsoft.

[24] D. Yang, Java Persistence with JPA, pp. 390: Outskirts Press, 2010.

[25] Oracle. "ResultSet," Jul, 2012; http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSet.html.

[26] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "A Concurrent Tuple Set Architecture for Call Level Interfaces," in ICIS - 12th IEEE/ACIS International Conference on Computer and Information Science, Niigata,Japan, 2013, pp. (accepted).

[27] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Assessment of a Enhanced ResultSet Component for Accessing Relational Databases," in ICSTE-Int. Conf. on Software Technology and Engineering, Puerto Rico, 2010, pp. V1:194-201.