A Translation Layer for Automatic Conversion of High-Level Access Control Policies to SQL Procedures

By

Zahid Rashid
(2009-NUST-MS PhD-IT-26)

Supervisor

Dr. Zahid Anwar

A thesis submitted in partial fulfillment of the requirements for the degree of

Masters of Science in Information Technology (MS IT)

In

School of Electrical Engineering and Computer Science

National University of Sciences and Technology (NUST)

H-12, Islamabad, Pakistan

(May 2013)

APPROVAL

It is certified that the contents and form of thesis entitled **"A Translation Layer for Automatic Conversion of High-Level Access Control Policies to SQL Procedures"** submitted by **Zahid Rashid**, have been found satisfactory for the requirement of degree.

Advisor: ___________________

(Dr. Zahid Anwar)

Committee Member: __________________

(Dr. Sharifullah Khan)

Committee Member: __________________

(Dr. Osman Hasan)

Committee Member: __________________

(Mr. Muhammad Bilal)

IN THE NAME OF ALMIGHTY ALLAH

THE MOST BENEFICENT AND THE MOST MERCIFUL

TO MY LOVING FAMILY

CERTIFICATE OF ORIGINALITY

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Zahid Rashid**

Signature: ______________

ACKNOWLEDGEMENTS

First of all I am extremely thankful to Almighty Allah for giving me courage and strength to complete this challenging task and to compete with international research community. I am also grateful to my family, my parents who have supported and encouraged me through their prayers that have always been with me.

I am highly thankful to Dr. Zahid Anwar for his valuable suggestions and continuous guidance throughout my research work. His foresightedness and critical analysis of things taught me a lot about valuable research which will be more helpful to me in my practical life.

I am highly grateful to Dr. Sharifullah Khan, Dr. Osman Hasan and Mr. Mohammad Bilal for their help and guidance throughout the research work. I am also thankful to all of my teachers who have been guiding me throughout my course work and have contributed to my knowledge. Their knowledge, guidance and training helped me a lot to carry out this research work.

Zahid Rashid

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In enterprise and cloud environments where employee and customer data is rapidly and constantly changing there is a need for fine grained and flexible access control policies which are easy to administer. Traditional models like access control lists (ACL) and database views fall short and enterprises typically resort to embedding access controls in the applications itself, a process which is developer error prone and results in increased application complexity. As a consequence of these problems, the use of reflective security policies is becoming popular where database privileges are expressed as database queries themselves rather than a static privilege contained in an access control matrix (ACM). Actual data in the database is used for its own protection and any updating of queries results in automatic update of policies. The focus of this work is the proposal of a mechanism to further reduce the task of database security policy administration. Transactional Datalog (TD), an extension of classical datalog has been proposed as a medium for authoring access control policies by which high-level policies may be automatically converted to reflective SQL procedures to be stored in the database. This mechanism provides a reflective way of implementing security policies instead of static privileges contained in ACLs. In this thesis we have provided a translation layer for compiling TD rules into appropriate SQL statements and storing as user defined functions in the database. Our translation layer allows a security administrator to express powerful access control policies in the high-level language of TD while having minimal knowledge of the underlying database schema or database implementation. We have evaluated our translation layer by authoring four popular and reasonably complex policy models namely (i) Chinese Wall (ii) Bell La Padula (iii)

Role Based Access Control (iv) and Temporal Policies. Detailed rule-sets and their corresponding database schemas have been discussed along with examples. Security administrators new to usage of reflective access control policies can tailor the four policy models to almost any variation they desire because the policy models we have picked in literature serve as foundation for many policy models today. Finally to verify that our translation layer does not compromise security or degrade performance we have tested our translation algorithm using two different approaches. Formal verification of access control policies using SPIN model checking tool shows that the security of the automated translation is as good as the manual approach and timing analysis of realistic applications demonstrate that it adds negligible impact on performance.

INTRODUCTION

This chapter is about the introduction of research work that has been undertaken in this thesis. The contents of this chapter include problem background, problem in existing access control models, motivation and problem definition and contributions of our research work.

1.1 Problem Background

Access control is the core component of database security. It is the ability to allow only authorized users, programs or processes to access the database based on the access control policies. The functionality of database access control is based on the implementation of access control policies. The proper implementation of access control policies in an efficient access control model would ultimately enhance the efficiency and performance of a database system in terms of confidentiality, availability, integrity, non-repudiation, authentication and accountability. There should be a flexible mechanism for enforcing access control policies based on data contents and other relevant contextual information (Bertino & Sandhu, Jan-Mar 2005).

There are many access control models and policies available for different types of environments. Bell LaPadula Model (BLP) (Bell & LaPadula, Mar. 1973) and (Bell & LaPadula, Mar. 1975) was proposed mainly considering the confidentiality of information in government and military organizations. For instance, "Unclassified personnel cannot read data at confidential levels" and "Top-Secret data cannot be written into files at unclassified levels". Chinese Wall Policy (CWP) (Brewer & Nash, May 1989) is a hybrid policy that addresses both confidentiality and integrity which targets the information flow policies for data owned by commercial and business entities. The environment of an investment house is an example to CWP, "a single consultant should not have access to information about two corporations that are in competition" because such information creates a conflict of interest in a consultant's analysis. However, a consultant is free to advise corporations that are not in competition with each other.

In other situations Role Based Access Control (RBAC) (Ahn & Sandhu, 2000) suits the need of commercial to civilian government organizations and focuses on separation of duty, integrity, availability, confidentiality and privacy. Roles are created for various job functions in an organization and users are assigned roles based on their responsibilities and qualifications, e.g. doctors, nurses, patients, managers, accountants, help desk representatives. For example "Doctors may write a prescription for any patient". There may be temporal dependencies among roles which is addressed in RBAC extension of Temporal-RBAC (TRBAC) (Bertino, Bonatti, & Ferrari, 2000) and (Rashid, Basit, & Anwar, 2010). For example, "Doctor-on-night-duty role is enabled during the night".

1.2 Problems in Existing Access Control Models

There are different mechanisms for implementing the access control models and policies and are based on views and tables (Opyrchal, Cooper, Poyar, & Le, April, 2011). Access Control Lists (ACL) and Access Control Matrices (ACMs) are mostly used for implementing the access control policies or it may be combined with roles by many database systems in which access right is stored in the form of a triple <user, resource, operation>. These mechanisms provide flexibility by attaching an ACL or ACM with each table or view which are describing the authorized operations on them (Olson L. E., Gunter, Cook, & Winslett, 2009). For implementing more fine grained access control policies, a separate view is created describing the portion of the data, and the users are granted access to that view. Besides the flexibility of ACLs and views, these models face difficulties in implementation, administration, scalability and expressiveness for reflective policies. These issues arise particularly in the environments where data is continually changing and more views are required (Olson, Gunter, & Madhusudan, Oct. 2008) and (Opyrchal, Cooper, Poyar, & Le, April, 2011). Consider the policy “each employee can view their own data”, where there is a need to create a view for each employee and then grant access to each employee on her own view, which makes policy administration a tedious task for administrators (Olson L. E., Gunter, Cook, & Winslett, 2009) and (Opyrchal, Cooper, Poyar, & Le, April, 2011). Further, these types of access control models do not have a formal framework for expressing the policies in a formal way. The same problem of creating a large number of views and roles also arises when implementing the fine grained reflective policies in RBAC along with ACMs. As a result, currently most of the access controls are performed at application level instead of at the database level and the access control mechanisms provided by most of the DBMS are not used effectively (Opyrchal, Cooper, Poyar, & Le, April, 2011). This approach requires that application embedded queries should be written based on the views which ultimately results in additional complexity in the application code. In such cases, the application requires full database access rights which is mandatory for running all the queries and eventually provides security vulnerability loop holes. Furthermore, every newly developed module of the application needs to be thoroughly tested for the consistency of the access control policies hence increasing the burden on quality assurance. Other issues related to these kinds of approaches include the performance of the application and change management. To overcome these issues the research community suggested that access control policies should be evaluated and enforced outside of applications (Rizvi, Mendelzon, Sudarshan, & Roy, 2004).

1.3 Motivation

The issues discussed in section 1.2 were addressed in Reflective Database Access Control model (RDBAC) (Olson, Gunter, & Madhusudan, Oct. 2008) in which policies are reflective and more expressive and depend on data contained in the database rather than data contained in ACLs or ACMs. In RDBAC, a database privilege is expressed as a database query itself, rather than as a static privilege which is contained in the ACM. In this manner, we can define more flexible policies and these policies can refer to any part of the database instead of only permissions found in the ACM (Olson L. E., Gunter, Cook, & Winslett, 2009). In this manner, actual data in the database is used for its own protection and any update in the data results in automatic update of query results and ultimately policies are updated that overcome the complex task of policy management. The benefits of this approach are that the policies are stored, implemented, enforced and evaluated at the database level. We can implement access control policies on database tables, entire row(s) in a database table or even at individual fields in a row which is required by most database applications (Chaudhuri, Dutta, & Sudarshan, April 2007). The applications don't need to embed code for the access control, rather they only need to call queries and get data back hence overcoming the problems of change management, performance, consistency in access control policies to some extent (Opyrchal, Cooper, Poyar, & Le, April, 2011). Further, the issue of expressiveness of the access control policies is also addressed in RDBAC by using Transaction Datalog (TD) (Bonner, 1998) which is an extension of classical Datalog as a formal framework for writing policies in a formal way and these polices can be formally analyzed (Olson, Gunter, & Madhusudan, Oct. 2008) before their implementation in a real world environment.

1.4 Problem Definition and Contributions

We have provided a translation layer for converting TD rules into equivalent SQL queries and have automated the process of writing the reflective policies. Moreover, we have presented the design of some of the most well known and established policies in literature and in commercial use together with their supporting relational schemas so that it is easy for DBAs to mold our designs for their purposes.

The main contributions of our research work are listed below:

(1) Translation layer for converting TD rules into equivalent SQL queries.

(2) Formulation of TD rules and relational schema design of (a) CWP (b) BLP Model (c) RBAC Model and Temporal policies in RDBAC framework.

(3) The algorithm for translating the TD to SQL queries as discussed in (Olson L. E., Gunter, Cook, & Winslett, 2009), (Hajiyev, Verbaere, & de Moor, 2006) and (Hajiyev, Sep. 2005) has been extended by incorporating the compilation of temporal policies and SQL update statement.

(4) Formal verification of access control policies using SPIN model checking tool and

(5) Timing analysis of realistic applications.

We have conducted experiments for calculating translation time of TD rules to SQL queries which are translated by our proposed algorithm. Further, we have also provided the results of evaluating performance of implemented prototype applications on the basis of translated SQL queries on varying size data sets, which shows practicality and feasibility of RDBAC framework.

1.5 Outlines of Thesis

There are nine chapters in this thesis which are organized in a following way: Chapter 2 provides the background and Chapter 3 provides the related work. The overall system flow model is discussed in the Chapter 4. The Chapter 5 provides the detail discussion about the algorithms for translating TD rules to SQL queries. In Chapter 6 the implementation is discussed in detail. The formal verification of the proposed implementation is presented in Chapter 7. The Chapter 8 provides the evaluation details and in Chapter 9 there are concluding remarks and discussion of future work.

BACKGROUND

In this chapter we describe what is Transactional Datalog (TD) - a high-level programming language to specify policy rules followed by brief description of different policies that we will design and implement throughout this paper namely the BLP, CWP, RBAC and Temporal.

2.1 Transaction Datalog (TD)

Classical Datalog was extended in syntax and semantics to TD (Bonner, 1998) which allows Datalog rules to add and delete the data of underlying database. It was designed as a high-level programming language to model workflows, where programmers can specify transactions containing both queries and updates, composing them using sequential and parallel constructs. TD also has precise mathematical semantics that includes atomic updates to databases that prevent nontrivial interference between transactions and maintain serializability. TD rules can be partitioned into head and body literals and are in the form:

$$p \text{ :- } q_1, q_2, \ldots, q_n \tag{1}$$

where p and q are the literals. The p is called the head of the rule and is separated by (: -) from $q_1, q_2, \ldots, q_n$ body literals. A literal is a string of the form:

$$p_{name}\,(t_1, t_2, \ldots, t_n) \tag{2}$$

where p_{name} is a predicate name with arity n and each ti for $1 \leq i \leq n$ is either a constant or a variable. In any literal the string $(t_1, t_2, \ldots, t_n)$ is called a tuple with arity n. The tuples are composed of variables and constants i.e. $t_1, t_2, \ldots, t_n$ may be constants and variables and the variable assignment is a functional mapping of variables to constants. The rule is called a fact if there are no variables in the head and the body of the rule is empty. For writing the fact we can eliminate the colon and hyphen separator, e.g. $p\,(t_1, \ldots, t_n)$.

There are two types of predicates (1) built-in or base and (2) derived or database. The built-in predicate is a predicate with pre-defined mapping and remains constant over every database interpretation normally having name with a non-alphanumeric string. For instance, the equality predicate is a built-in predicate containing the rules = (1, 1) and =(X, Z):- = (X, Y), = (Y, Z). Those predicates which are not built-in predicates are called database predicates. When we are describing database definitions we only list database predicates because the semantics of built-in predicates remains the same avoiding the need to list them every time.

We define the insertion (ins.p) and retraction (del.p) predicates of arity n for each base predicates p with arity n. The literal at the head of the rule must have either a base predicate name or a derived predicate name i.e. not assertion or retraction predicate names. If the head predicate is a base predicate then the rule must be a fact and its body must be empty.

As there are database update predicates in TD therefore inference system is defined for answering the queries. The inference system of TD rules is similar to Datalog and also keeps tracks of the sequence of database states required to reach the conclusion. The update predicates change the state of the database while the other predicates do not and their truth values are computed based on whether the tuple exist as a fact in the database. Whenever the ins.p or del.p predicates are inferred, these are always true if the state of the database changed, while the inference of derived predicates are similar to the inference in the classical Datalog.

The Extensible Access Control Markup Language (XACML) is also used for specification of flexible policies however it is not specifically designed for database access control policies (Olson, Gunter, & Madhusudan, Oct. 2008). Some other languages have also been used for policy writing in specialized databases such as P3P which has the capability of writing reflective and fine-grained control policies (Agrawal, Kiernan, Srikant, & Xu, 2002) and (Agrawal, Bird, Grandison, Kieman, Logan, & Rjaibi, 2005). Prolog has also been used as a language for writing the accesses control policies (Draxler, 1991). TD is used because it is naturally supporting to SQL queries because of augment of classic Datalog and has precise mathematical semantics, incorporating recursive definitions, transaction-based atomic updates, assuring serializable execution of transactions and has the capability to directly map onto most of the functions of SQL.

2.2 Bell La Padula (BLP)

The BLP (Bell & LaPadula, Mar. 1973) and (Bell & LaPadula, Mar. 1975) is a state machine model used for enforcing access control in government and military applications. It concentrates on confidentiality policies and describes access control rules with the sole goal of preventing information from being leaked to those who are not privileged to access the information. In BLP, the entities in an information system are divided into a set of subjects 'S', a set of objects 'O', a set of access operations i.e. "A = {execute, read, append, write}" and a set 'L' of security levels with a partial ordering. Classification labels are assigned to objects and clearance labels to subjects to implement the set of access control rules. Security labels range from the most sensitive (e.g."Top Secret"), down to the least sensitive (e.g. "Unclassified" or "Public").

The BLP model is based on two rules "No Read Up" (NRU) also called simple security property and "No Write Down" (NWD) also called *-property. Let L(s) be the clearance level of subjects and L(o) is the classification level of objects:

(1) The simple security property states that a subject can read an object if the object's classification level is less than or equal to the subject's clearance level i.e. *"Subject s can read object o iff, $L(o) \leq L(s)$ and s has permission to read o"*.

(2) The *-property states that a subject can write to an object, if the subject's clearance level is less than or equal to the object's classification level i.e. *"Subject s can write object o iff L(s) ≤ L(o) and s has permission to write o".*

The notion of security level in BLP was extended by the inclusion of categories and the security level is transformed to *(clearance, category set) e.g. (Top Secret, {NUC, EUR, ASI}), (Confidential, {EUR, ASI}) etc.* The categories are a set of non-hierarchical attributes such as "US", "EU" etc. In the extended security level, there may be zero or more categories. The security level is a combination of clearance/classification and categories for example *{Secret/US, EUR}* or *{Top Secret/US}*. The definition of simple security property and the *-property after the inclusion of categories becomes:

(1) "Subject s can read object o iff, L(s) dom L(o) and s has permission to read o" and

(2) "Subject s can write object o iff, L(o) dom L(s) and s has permission to write o" respectively.

Where *dom* operator is a partial order over the lattice. The major limitation of BLP model is the scope of implementation i.e. it can be implemented easily in few governmental organizations where the levels of authorization can be identified. It was originally designed for systems in which there are no changes in the security levels. In BLP there is a need to assign security labels to everything in the system; however it is not always possible to assign security labels to users and data of a commercial system. Therefore, implementation of BLP is not suitable for most of the commercial applications. Besides the flexibility issues, BLP also faces the downgrade in performance due to checking of security clearance for each item in the system. BLP does not consider the integrity of data only dealing with the confidentiality. The other limitation of BLP is that, it does not address the policies for modification in access rights, and may contain covert channels causing information to leak from a high to low security level.

2.3 Chinese Wall Policy (CWP)

The CWP provides access controls that change dynamically based on access rules and mitigates conflicts of interests (Bell & LaPadula, Mar. 1975). It uses the concept of Conflict of Interest (COI) classes to implement the access control system. For instance, companies which are in competition with each other are placed in one group. For instance, Bank A, Bank B and Bank C are in one COI class while Gasoline Company A and Gasoline Company B in another COI class.

The idea behind the CWP is that once you access any information from any company dataset you are allowed to access that information, but you are no longer allowed to access information from another company dataset within that conflict of interest class. In CWP *Objects* are known as items of information related to a company, *Company dataset (CD)* contains objects related to a single company and groups of competing companies are called *COI classes.* All the information is maintained in a hierarchical order with objects at the lowest level, CD and COI respectively.

The simple security condition in the CWP can be defined as: Let *PR*(*s*) denote the set of objects that *s* has already read

s can read o iff any of the conditions holds:

1) *There* $\exists$ *o´ satisfying* $o' \in PR(s)$ *and* $CD(o')=CD(o)$;
2) *For all objects o,* $o' \in PR(s) \Rightarrow COI(o') \neq COI(o)$;
3) *o is a sanitized object*

Initially, $PR(s) = \varnothing$*, so initial read request is always granted*

Formally the *-property can be defined as:

s can write to o iff both of the following hold:

1) *The CW-simple condition permits s to read o*

   – *No blind write like in BLP*

2) *For all unsanitized objects o′, if s can read o′, then CD(o′) = CD(o)*

   – *All s can read are either within the same CD, or sanitized*

2.4 Role Based Access Control (RBAC)

In RBAC (Ahn & Sandhu, 2000), policies are described in terms of users, subjects, roles, role hierarchies, operations, relationships, and constraints. The users are granted membership into roles based on their competencies and responsibilities. Roles can be defined as an organizational job function with a clear definition of inherent responsibility and authority (permissions). Formally the RBAC is as under:

- *PA: Roles → Permissions,* the permission assignment function, that assigns to roles the permissions needed to complete their jobs;
- *UA: Users → Roles,* the user assignment function, that assigns users to roles;
- *user: Sessions → Users,* that assigns each session to a single user;
- *roles: Sessions →* 2^{Roles}, that assign each session to a set of roles ; and
- *RH* $\subseteq$ *Roles x Roles,* a partially ordered role hierarchy

The users must be active in a role and authorized as a member of the role by a security administrator for performing operations. RBAC provides the capability to administrators to place constraints on role authorization, role activation, and operation execution and facilitating administrators to control the access at a level of abstraction.

The RBAC model is quite flexible and is used in many commercial applications with slight modifications. Besides the flexibility of RBAC, its major limitation is that it has no formal way of expressing complex policies. It is not well suited for a highly distributed environment because it usually needs centralized management of user-to-role and permission-to-role assignments. The centralized management affects the flexibility of RBAC and it becomes difficult to implement discretionary policies and separation of duty controls. For implementing the strong security, more granular roles need to be engineered which is itself a complex task. As more attributes are involved, the number of roles and permissions needed to encode these attributes will grow exponentially, thereby making *UA* and *PA* difficult to manage.

2.5 Temporal Policies

Temporal polices are those policies that are based on temporal constraints as described in (Bertino, Bonatti, & Ferrari, 2000) i.e. time periods e.g. employees in day shift can log into the system between 8 AM to 4 PM. The temporal constraints has attained attention of the researchers because in many organizations, functions may have limited or periodic temporal duration. Consider the case of part-time staff in a company, and assume that part-time staff is authorized to work within the given organization only on working days, between 9 AM and 1 PM. If part-time staff is represented by a role, then the above requirement entails that this role should be enabled only during the aforementioned temporal intervals. Let us take the example of "TRBAC: Temporal Role Based Access Control" (Bertino, Bonatti, & Ferrari, 2000) which is an extension of RBAC specifically considering the temporal constraints. Often roles are characterized by a temporal dimension i.e. job functions may have limited or periodic time duration and there may be activation dependencies among roles e.g. Role A should be activated only after activating Role B. Some examples of formally written temporal policies are

- *([7/1/12,12/31/12, night-time, VH, activate, doctor-on-night-duty)*
- *([7/1/12,12/31/12, day-time, VH, deactivate, doctor-on-night-duty)*

RELATED WORK

In this chapter we will discuss the related work in which the concept of reflection has been implemented in various access control models for the implementation of access control policies. Reflective access control and access control polices at fine grained level i.e. up-to individual fields have been studied for a long time (Griffiths & Wade, 1976), (Lunt, Denning, Schell, & Heckman, 1990), (Stonebraker & Wong, 1974) and (Chaudhuri, Dutta, & Sudarshan, April 2007). Some commercial DBMSs such as Oracle provide mechanisms for implementing the access control policies at a fine grained level (Feuerstein & Pribyl, Oct. 2009) and (Oracle-Corporation, June 2005). The policies are setup by the administrators and the data queries are rewritten automatically by adding additional *WHERE* clauses based on the policies. Several other research contributions also present the work on enforcing access control at the row level as well as at column level (Agrawal, Kiernan, Srikant, & Xu, 2002), (Bobba, Fatemieh, Khan, & Gunter, 2006), (Goodwin, Goh, & Wu, 2002), (Rizvi, Mendelzon, Sudarshan, & Roy, 2004), (Agrawal, Bird, Grandison, Kieman, Logan, & Rjaibi, 2005), (Jahid, Hoque, Okhravi, & Gunter, 2009) and (De Capitani di Vimercatii, Foresti, Jajodia, Paraboschi, & Samarati, 2008). Another system proposed in (Zhang & Mendelzon, 2005) uses authorization views in which the database query is checked for validity and determines automatically whether it can be completely rewritten using the authorization views. In (Jahid, Gunter, Hoque, & Okhravi, 2011) XACML polices were compiled for database access into ACLs which are natively supported by the database. This system was proposed keeping in view the context of reflective database access control where attributes used in access decisions are stored in the database itself. Now we discuss the benefits and limitations of some existing well known models which are inherently reflective, flexible and have the ability to implement policies at fine grained level.

3.1 An authorization mechanism for a relational database system

One of the most popular models, named Griffiths and Wade Model (Griffiths & Wade, 1976) was proposed in 1976 and is still largely used in modern commercial databases. It is a view based discretionary access control model which uses the ACM as a mechanism for implementing the access control authorizations. The views provide a powerful and flexible security mechanism by hiding parts of the database from certain users. The views are already reflective in their nature and are widely used in database access control systems. It is conceptually a simple model for access control: the database maintains an ACM listing about the resources provided by the database, such as tables, views, and functions; the users that are allowed to access each resource; and which operations each user is allowed to perform on the resource, such as read, insert, update, or execute. If access control is needed at a fine-grained level, in which a user should only be granted access to certain portions of a database table, then a separate view is created to define those portions, and the user is granted access to the view. This model is flexible enough to allow users to define access privileges for their own tables, without requiring super user privileges. However, ACMs are limited in expressing the extent of the policy, such as "Alice can view data for Alice" or "Bob can view data for Bob", rather than the intent of the policy, such as "each employee can view their own data". This makes access control administration more tedious in the face of changing data, such as adding new users, implementing new policies, a large number of users or modifying the database schema.

One of the main limitations of Griffiths-Wade Model is that the complex policies can be difficult to implement in this model e.g. "Every employee can access their own records" and "Every employee can view the name and position of every other employee in their department". If there are a large number of users in the system than the policy administration is a difficult task.

Similarly if we want to implement the policies at fine-grained level then each requires their own view, and there is no formal way to see that the views are created on the intended target table. The other limitations are that the view re-definitions require dropping the view, redefining and then re-issuing privileges which are most likely to introduce errors. In case of update in the schema of the database, the updates need to be made at multiple places which also increases the chances of errors. The administration of this model becomes increasingly difficult in an environment where changes in the data occur frequently e.g. adding new users, deleting users etc. and the schema of the database changes.

3.2 Hippocratic Databases

In Hippocratic database (Agrawal, Kiernan, Srikant, & Xu, 2002), privacy is a main concern and the database supports built-in privacy controls. Hippocratic databases make a distinction between users that own a database table and users that own the data contained in the table. Studies on this paradigm have shown how policies for such databases might depend on data contained within a table and invoking the idea of allowing user to define arbitrary policy logic. These databases show how reflection is used in the implementation of privacy policies particularly in medical databases and requires merging security policies from database owner (s) and from data owner (s). But these studies do not further examine any security implications focusing more on using boolean values in query optimization. Further, in (LeFevre, Agrawal, Ercegovac, Ramakrishnan, Xu, & DeWitt, 2004) a system was presented for limiting the disclosure specifically in Hippocratic databases but not tested for general purpose databases. It uses the query modification technique on the basis of already stored privacy policies in the database.

3.3 Oracle Virtual Private Databases

Oracle's Virtual Private Database (VPD) technology was provided by (Oracle-Corporation, June 2005) and (Feuerstein & Pribyl, Oct. 2009) as an additional feature of Oracle database management system to enhance the capability of implementing access control policies in a more flexible manner. It allows implementing the concept of reflection in access control policies. VPD has provided a mechanism for writing logic of policies in the form of arbitrary code using the PL/SQL statements wrapped in user defined functions (UDF). This mechanism has provided the flexibility to policy writes for writing more expressive policies.

Every query on database table is rewritten transparently by UDFs. These UDFs return policy conditions which are added in the SQL *WHERE* clause of the queries. We can attach multiple UDFs to a table and different UDFs can be defined depending on operation (read vs. write) and columns being accessed. The UDFs act as query filters for granting access to current users, access to other table data (excluding current table). These UDFs are executed each time the table is accessed. In this technology there is no formal framework for describing, evaluating and analyzing database access control policies.

3.4 Reflective Database Access Control Model (RDBAC)

In RDBAC (Olson, Gunter, & Madhusudan, Oct. 2008) model, a database privilege is expressed as a database query itself, rather than as a static privilege contained in an ACM. In this model access control policy decisions can depend on data contained in other parts of the database, such as attributes of the user, attributes of the data being queried, or relationships between the user and the data. Hence policies can refer to any part of the database and overcoming the limitation of stratification of access control data than the actual information in the database.

One of the main advantages of RDBAC is that it aids in improving the expressiveness of access control policies. For example the policy “employees who are registered as managers may view contact data for the employees they manage” illustrates the benefits of RDBAC i.e. suppose we have a database that contains a table listing a company’s employees, along with their position in the company and the department in which they work. We want to grant all employees having manager role to access the data of other employees in their department. When a manager queries this table, the policy will first check that the user is indeed a manager, then retrieve the manager’s department, and finally return all employees in that department. The main benefits of this approach are it uses the actual data stored in the database and thus, privileges are automatically updated when the database is updated (for instance, when an employee receives a promotion to manager), preventing update anomalies that leave the database in an inconsistent state. All the existing implementations lack in expressing the policies formally and analyzing these polices before implementing in the real world environment. One of the lacking of RDBAC is that, there is no efficient way for formal mathematical molding of access control policies for practical database systems. For this purpose TD language was used which provides a powerful syntax and semantics for expressing RDBAC policies. TD provides a very concise syntax that is capable of expressing a wide range of policies but it is not implemented and tested for large scale environments and also lacks syntax and semantics that correspond to certain commonly used operations in SQL.

CHAPTER 4

SYSTEM FLOW MODEL

In this chapter we will describe the overall flow of our system.

4.1 System Flow Model

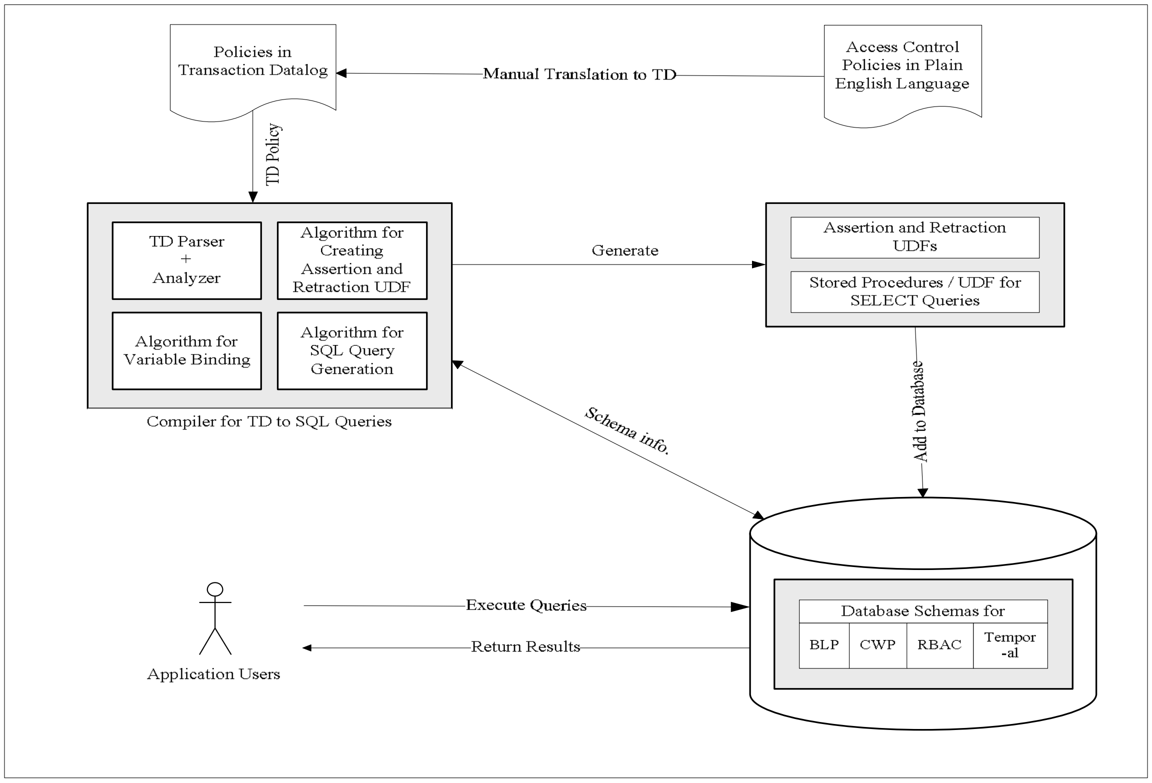
Policies in Transaction Datalog
Manual Translation to TD
Access Control Policies in Plain English Language
TD Policy
TD Parser + Analyzer
Algorithm for Creating Assertion and Retraction UDF
Algorithm for Variable Binding
Algorithm for SQL Query Generation
Compiler for TD to SQL Queries
Generate
Assertion and Retraction UDFs
Stored Procedures / UDF for SELECT Queries
Add to Database
Schema info.
Application Users
Execute Queries
Return Results
Database Schemas for
BLP
CWP
RBAC
Tempor -al

Figure-1: System Flow Model

The flow of system as shown in Figure-1 starts by writing access control policies in plain language according to security requirements by team of experts from different domains of organization e.g. database security experts, database administrators, management or expert domain users. For instance few access control policies are: "Managers can view data of employees working in their departments only", "Each employee can only view their own data", "Doctors can only view contact information of patients for whom they have written prescriptions" and "Teachers can only view the grades of those students who they are currently teaching".

All plain language policies written by team of experts are provided to database administrator. The database administrator is responsible for translating plain language policies into TD rules during the design of system as well as at later stages. The process of translating natural language policies into TD rules is facilitated by graphical user interface of compiler.

The information about target database such as name of database and path is required by compiler to connect to database for retrieving information about database schema. The compiler takes TD rules and retrieved schema information as input. The compiler uses this information for variable binding (see section 5, Algorithm 5 for details) of TD rules and translates into equivalent SQL SELECT queries. The generated SQL queries are wrapped into User Defined Functions (UDF). The information about current user from the user session is passed as parameters to UDFs along with other variables. The parameters received by the UDFs are used in the SQL SELECT queries for filtering the data. The generated SQL queries can also be wrapped into parameterized views if the database management system allows update statement in parameterized views. The UDFs are added to target database for further use and executed by user applications and results are returned.

The algorithms for TD parsing, creating of assertion and retraction UDFs, variable binding, SQL query generation which are used by the compiler for translating TD rules into SQL queries are discussed in detail in section 5.

Whenever a new security policy is needed during the use of system or if there is a need to modify existing policy, the same procedure of translation from plain text to SQL queries will be applied. The generalized schemas for BLP, CWP, RBAC and temporal polices are already provided for use by different users. These schemas can be used in any environment with slight modifications in the tables.

CHAPTER 5

ALGORITHMS FOR TRANSLATING TD RULES TO SQL QUERIES

In this Chapter, algorithms are presented which are used by the compiler for translating TD rules into SQL SELECT queries and for generation of assertion and retraction UDFs. In order to compile TD rules into SQL SELECT queries, we follow a similar approach as used in (Olson L. E., Gunter, Cook, & Winslett, 2009), (Hajiyev, Verbaere, & de Moor, 2006) and (Hajiyev, Sep. 2005).

5.1 Parsing of TD Rules

The Algorithm 1 is used for parsing TD rules. It takes the TD rule as input and the output are head predicate, body predicates, head variables and body variables. After getting input, the TD rule is passed a function named Split_Rule ( ) which parses the TD rule and then splits it into head and body literals. The (:-) symbol is used as a separator between head and body literals and are stored in arrays HL and BL. The head literal is further separated to get the head predicate and variables, which are stored in arrays of strings for further use. The function named Get_Variables () is used for retrieving the variables from a literal. This function first gets the head literal as input in the form "p (t_1, t_2, ..., t_n)" and retrieves the variables in it. The symbol ',' is used as a separator among the variables. The function named Get_Predicate () is used for retrieving the predicate name from a literal. This function first gets head literal as input in the form "p (t_1, t_2 ..., t_n)" and retrieves the predicate name i.e. p. Similarly the body literals, body predicates and variables are retrieved, which are then stored in multi-dimensional arrays for use in further steps. The parsing of TD Rules is shown in Algorithm 1 and an example is shown in Figure-2:

Algorithm 1. Parsing of TD Rules

```
Input: TD_Rule ▶ It describes the policy which is compiled into SQL SELECT statement
Ensure: TD Rule is in correct format ▶ Input in the form "p: - $q_1$, $q_2$, …., $q_n$"
1.  HL, BL :- Arrays ▶ Arrays for Head and Body Literals
2.  HV, BV:- Multidimensional Arrays ▶ Arrays for Head and Body predicates and variables
3.  HL, BL ← Split_Rule ( TD_Rule )
4.  HV ← Get_Variables (HL)
5.  BV ← Get_Predicate (HL)
6.  for i = 0 to length of BL do
7.      HL ← Get_Variables (BL)
8.      BL ← Get_Predicate( BL)
9.  end for
Output: HL, BL, HV, BV
```

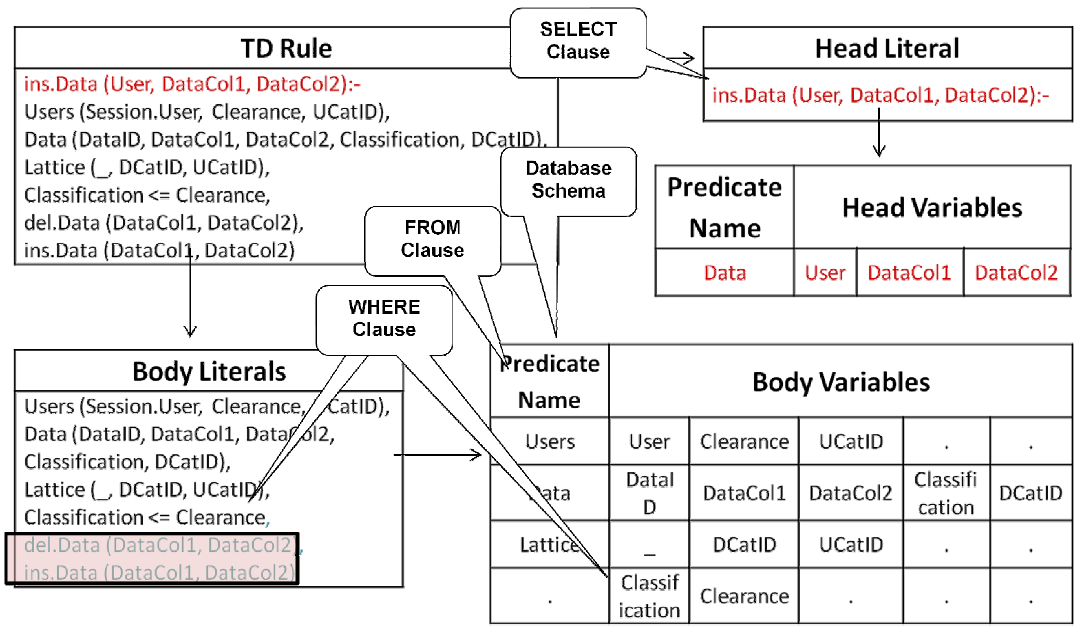
TD Rule
ins.Data (User, DataCol1, DataCol2):-
Users (Session.User, Clearance, UCatID),
Data (DataID, DataCol1, DataCol2, Classification, DCatID),
Lattice (_, DCatID, UCatID),
Classification <= Clearance,
del.Data (DataCol1, DataCol2),
ins.Data (DataCol1, DataCol2)
SELECT Clause
Head Literal
ins.Data (User, DataCol1, DataCol2):-
Database Schema
FROM Clause
WHERE Clause
Predicate Name
Head Variables
Data
User
DataCol1
DataCol2
Body Literals
Users (Session.User, Clearance, UCatID),
Data (DataID, DataCol1, DataCol2,
Classification, DCatID),
Lattice (_, DCatID, UCatID),
Classification <= Clearance,
del.Data (DataCol1, DataCol2),
ins.Data (DataCol1, DataCol2)
Predicate Name
Body Variables
Users
User
Clearance
UCatID
Data
DataID
DataCol1
DataCol2
Classification
DCatID
Lattice
_
DCatID
UCatID
Classification
Clearance

Figure-2: Parsing of TD Rule

5.2 Reading Database Schema

The algorithm 2 is used for connecting to the target database. It takes information about the target database which is used by the compiler for establishing connection. The array BV which is the output of Algorithm 1 is also passed as input. It contains the lists of tables whose detailed schema needs to be retrieved from the target database and is stored in multidimensional array named DS. After establishing connection with the target database, the compiler reads the schema of the database tables corresponding to the predicate names in the BV array. The schema of the database tables along with its order and data types are stored in DS which are further used in the variable binding. We assume for the implementation of this algorithm that the database imposes a stable full ordering on table fields i.e. columns. The algorithm for Reading Database Schema is shown in Algorithm 2:

Algorithm 2. Reading Database Schema

```
Input: BV, Database_Info ► BV output of Algorithm 1 and Information about the target database
Ensure: Connection with the target is established
1.  DS :- Multidimensional Array
2.  Connect_to_Database (Database_Info)
3.  for i = 0 to length of BP do
4.      DS ← Read_Database_Schema (BV)
6.  end for
Output: DS
```

5.3 Creating UDFs for Assertion and Retraction

The algorithm 3 is used by the compiler for creating the Assertion and Retraction UDFs. It takes the TD rule as input. It first checks the TD rule for the existence of assertion and retraction i.e. *ins* and *del* predicates. If the assertion and retraction predicates exist in the body of rule than corresponding UDFs for assertion and retraction are created. The algorithm for creating the assertion and retraction UDFs is shown in Algorithm 3:

Algorithm 3. Creating UDFs for assertion and retraction

```
Input: TD_Rule
Ensure: TD Rule is in correct format ► Input in the form "p :- $q_1$, $q_2$, ..., $q_n$"
1.  AL, AV, RL, RV:- Arrays
2.  HL, BL ← Split_Rule (TD_Rule)
3.  for i=0 to length of BL do
4.      If Exits Assertion Literal ►String "ins." is used for the identification of the assertion predicate.
5.          Add assertion literal to the AL array
6.      End If
7.      If Exits Retraction Literal ►String "del." is used for the identification of the retraction predicate.
8.          Add retraction literal to the RL array
9.      End If
10. end for
11. for i=0 to length of AL do
12.     AV ← Get_Variables (AL) ►  Variables in assertion literals are stored in array named AV.
13.     Create_Assertion_UDF (AL, AV)
14. end for
15. for i=0 to length of RL do
```

```
16.     RV ← Get_Variables (RL) ▶  Variables in retraction literals are stored in array named RV.
17.     Create_Retraction_UDF (RL, RV)
18. end for
```

Output: *Assertion and Retraction UDFs*

The UDF for assertion predicate consist of the SQL “INSERT” statement and the parameters passed to this UDF are used for inserting the values in database. Similarly the UDF for the retraction predicate consist of the SQL “DELETE” statement and the parameters passed to this UDF are then used for deleting the records from the database. If the assertion or retraction predicates exist in TD rules than a flag is set to true for each assertion or retraction, and based on true values of flag, UDFs are called and parameters are passed to them. The function named *Create_Assertion_UDF()* gets the arrays named AL and AV as input and generates the code of T-SQL. The execution of this code results in the generation of UDF on target database. This UDF gets the values as input parameters which need to be inserted in the database. The SQL INSERT statement in the UDF will ultimately insert the received values in the input parameters into the database. Similarly the function named *Create_Retraction_UDF ()* gets the arrays named RL and RV as input and generates the code of T-SQL. The execution of this code results in the generation of UDF on target database. This UDF gets the values as input parameters which need to be deleted from the database. The SQL DELTE statement in the UDF will ultimately perform the deletion from the database.

5.4 Creation of FROM Clause

The algorithm 4 is used by compiler for creating FROM clause of SQL SELECT queries. The SQL FROM clause is created by getting predicate names i.e. BV from Algorithm 1 of body literals, assigns an alias to it and are saved in a string for concatenating with the SQL SELECT clause. The algorithm for the creation of FROM clause is shown in Algorithm 4:

Algorithm 4. Creating FROM Clause

```
Input: BV ► Output of Algorithm 1
1. FC:- Arrays
2. FC ← Create_From_Clause (BV) ► Takes BV as input and creates the SQL "FROM"
   clause by combining all the values (i.e. table names) which are placed at the index "0" of the
   multidimensional array BV.
Output: FC
```

5.5 Variable Binding

The algorithm 5 is used by compiler for binding the variables of TD rules with database schema. The function named Variable_Binder () simply maps the ith term in literals to ith column in tables. The variable binder function uses the information from BV and fields from corresponding table schema. These variables and table fields are then mapped and bind according to location of BV and table fields. This multidimensional array named VB is used to store the binding information among the literal variables and the columns of the database target tables. The function named Variable_Binder() is used to bind the variables and constants of the body literals in the TD rule with the columns of target database tables. The algorithm for Variable Binding is shown in Algorithm 5 and an example is shown in Figure-3:

Algorithm 5. Variable Binding

```
Input: BL, BV ► Output of Algorithm 1
1.  VB:- Arrays
2.  for i=0 to length of BL
3.      for j=0 to length of BV
4.          VB ← Variable_Binder ()
5.      end for
6.  end for
Output: VB
```

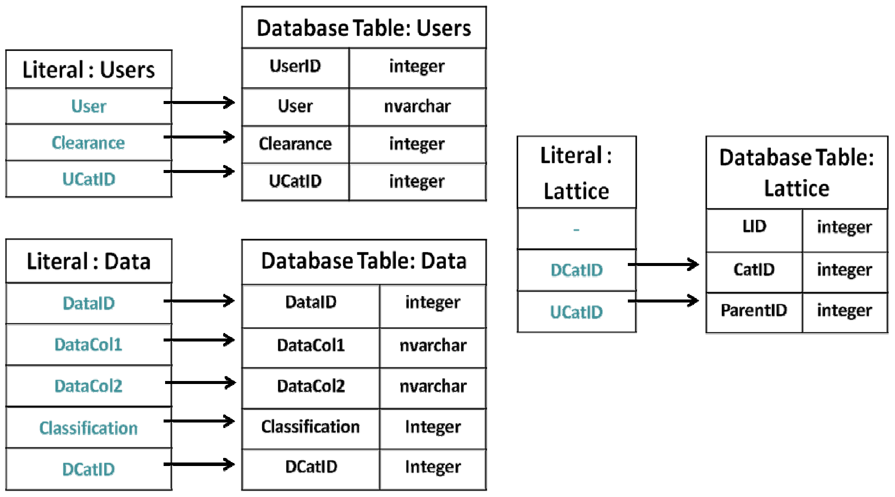

Figure-3: Variable Binding

5.6 Creating SELECT Clause

The SELECT query will be created after performing necessary steps in Algorithm 1 - 5. The already identified constants which are mentioned in single quotes in TD rules of the body literals are then mapped to the corresponding fields of the tables using the "=" symbol. If some built-in predicates exist than the corresponding conditions are imposed by using built-in predicates by passing the BL which is output of Algorithm 1. After imposing the conditions, these are concatenated in the SQL “WHERE” clause. The WHERE clause is then created by the use of algorithm provided in the (Hajiyev, Verbaere, & de Moor, 2006) and (Hajiyev, Sep. 2005).

The built in predicates, constants in literals mapped with columns and variables mapped with database columns are joined by the "=" operator i.e. joins by the use of variable binding information from algorithm 5 and is stored in an array named WC. All the strings in WC are then concatenated by the use of AND operator.

The SELECT clause is then created and stored in an array named SC by the use of variables in the head literal. The information about the variables in head literal are stored HV (which is output of Algorithm 1).

Finally, the strings SC, FC (Output of Algorithm 4) and WC are concatenated to form the final SELECT statement.

CHAPTER 6

IMPLEMENTATION OF ACCESS CONTROL POLICIES AND MODELS IN RDBAC FRAMEWORK

In this Chapter we will implement BLP, CWP, RBAC and Temporal policies in RDBAC framework by providing database schemas and TD rules for each policy. These TD rules are translated into equivalent SQL queries by the compiler which uses the algorithms as discussed in Chapter 5.

6.1 BLP

The implementation of BLP Model has used the following database schema as shown in Table 1.

Table 1. Database Schema for Implementation of BLP with Categories

No.	Table Name	Columns
1.	Users	UserID:- integer, User:-nvarchar, Clearance:-integer, UCatID:- integer
2.	Data	DataID:- integer, DataCol1:-nvarchar, DataCol2:-nvarchar, Classification:-integer, DCatID:- integer
3.	Security_Level	LevelID:- integer, Level:-nvarchar
4.	Category	CatID:- integer, Category:-nvarchar
5.	Set_Categories	ID:- integer, Set_Categories:-nvarchar
6.	Lattice	LID:- integer, CatID:- integer, ParentID:- integer

In Table 1, the database table named "*Users"* contains the list of users in the system along with their clearance level. The second table named "*Data"* contains the data on which the BLP read and write access is to be granted. This table also contains the classification level as well as the category assigned to each record in the system. The third table named "*Security_Level” is* used to keep the list of available clearance and classification levels in the system. The fourth table named *"Category"* is used to keep the list of categories in the system which is used for creating the lattice on partially ordered function. The purpose of table named *"Lattice"* is to maintain the information about the lattice which is used for the checking the security level in the system.

6.1.1 Simple Security Property of BLP Model

The TD rule for implementing the simple security property of BLP with categories is shown in the Table 2:

Table 2. TD Rule of Simple Security Property of BLP Model

```
ReadData    (User, DataCol1, DataCol2):-

            Users (Session.User, Clearance, UCatID),

            Data (DataID, DataCol1, DataCol2, Classification, DCatID),

            Lattice (_, DCatID, UCatID),

            Classification <= Clearance
```

The equivalent SQL query or view which actually implements the above property of BLP Model with Categories after compiling the above TD Rule is shown in Table 3:

Table 3. Equivalent SQL Query of Simple Security Property of BLP Model

```
SELECT      Session.User, d. DataCol1, d. DataCol2, u.UCatID,

            d.DCatID, u.Clearance, d.Classification

FROM        Data AS d, Lattice AS l, Users AS u

WHERE       u.UCatID = l.CatSetID AND d.DCatID = l.ParentID AND
            d.Classification <= u.clearance AND

            u.[User] = Session.User
```

6.1.2 *- Property of BLP Model

The TD rule for the implementation of the *-property of BLP with categories is shown Table 4:

Table 4. TD Rule of *- Property of BLP Model with Categories

```
insert.Data     (User, DataCol1, DataCol2):-

                Users (Session.User, Clearance, UCatID),

                Data (DataID, DataCol1, DataCol2, Classification, DCatID),

                Lattice (_, DCatID, UCatID),

                Classification <= Clearance,

                del.Data (DataCol1, DataCol2),

                ins.Data (DataCol1, DataCol2)
```

The equivalent SQL SELECT query which actually implements the *-property of BLP Model with Categories after compiling TD Rule is shown in Table 5:

Table 5. Equivalent SQL Query of *- Property of BLP Model

```
 SELECT    u.[user], d.data1, d.data2, u.UCatID, d.DCatID, u.Clearance,
           d.Classification

  FROM     Data AS d, Lattice AS l, Users AS u

 WHERE     u.UCatID = l.CatSetID AND d.DCatID = l.ParentID AND  u.clearance <=
           d.Classification AND u.user = @UserName AND d.DataID =  @DataID;

User Defined Function is called for retraction and insertion predicate
```

Now we consider a simple scenario, for simulating the BLP according to our implementation. Consider there are 3 categories in the system NUC, EUR and US. These categories are stored in the table named *Category*. We have generated the complete list of sets by using the subset operator (⊆) e.g. {NUC, US} ⊆ {NUC, EUR, US} and stored in the table named *Set_Categories*.

The complete lattice is stored in the table named *Lattice.* The generated lattice uses the list of subsets in *Set_Categories* tables. The clearance and classifications are Unclassified, Confidential, Secret and Top Secret. Now consider Mr. xyz is cleared into the security level (Secret, {NUC, EUR}), DocA is classified as (Confidential, {NUC}), DocB is classified as (Secret, {EUR, US}), and DocC is classified as (Secret, {EUR}). Mr. xyz tries to read the DocA, the information about the clearance level of Mr. xyz is retrieved from the tables *Users, Lattice* and information about the classification level of DocA retrieved from the tables named *Data, Lattice*. After that *dominates (S dom O)* relationship is checked and the read access is granted to Mr. xyz on DocA. Similarly in order to implement the *-property the *dominates (O dom S)* is checked and the write access is granted.

6.2 CWP

In this subsection we will present the database schema for implementation of CWP, TD rules and equivalent SQL queries for the implementation of Simple Security Property and *-Property of CWP. The database schema for CWP implementation in RDBAC is shown in Table-6:

Table 6. Database Schema for Implementation of CWP

No.	Table Name	Columns
1.	Users	UserID:- integer, User:-nvarchar
2.	UsersAccess	UserAccessID:- integer, UserID:- integer, CoiClassID:- integer, ClientID:- integer, ReadAccess:- integer, WriteAccess:- integer
3.	CoiClasses	CoiClassID:- integer, CoiClass:-nvarchar
4.	Clients	ClientID:- integer, Client:-nvarchar, CoiClassID:- integer
5.	ClientData	CdID:- integer, CoiClassID:- integer, ClientID:- integer, Data1:-nvarchar, Data2:-nvarchar, Data3:-nvarchar

In Table-6, the table named "*Users*" contains the list of users in the system. The second table named "*UsersAccess*" contains the references to conflict of interest (Coi) classes and clients. This table also contains references to user id and the read / write access rights on the client datasets. The *ReadAccess* and *WriteAccess* columns in this table are used for maintaining the read and write access and are updated on each new access. The third table named *"CoiClasses"* contains the list of Coi classes in the system. The table named *"Clients"* contains the list of clients along with their Coi class references and data of clients will be stored in table named *"ClientData"*.

6.2.1 Implementation of Simple Security Property of CWP

In Table 7, TD rule for simple security condition of CWP is shown:

Table 7. TD Rule of Simple Security Property of CWP

```
ClientData      (UserID, User, CoiClassID, CoiClass, ClientID, Client, Data1, Data2, Data3):-

                Users (UserID, Session.User),

                CoiClass (CoiClassID, @CoiClass),

                Clients (ClientID, @Client),

                UsersAccess (UserAccessID, UserID, CoiClassID, ClientID, ReadAccess,  _),

                ClientData (CdID, CoiClassID, ClientID, Data1, Data2, Data3)

                del.UserAccess (_, UserID, CoiClassID, ClientID, _, _),

                ins.UserAccess (_, UserID, CoiClassID, ClientID, 1,0)
```

The equivalent SQL SELECT query which actually implements the simple security property of CWP after compiling TD Rule is shown in Table 8:

Table 8. Equivalent SELECT Query of Simple Security Property of CWP

```
SELECT    u.User, u.UserID , cc.CoiClass, cc.CoiClassID, c.Client, c.ClientID, cd.Data1,

          cd.Data2, cd.Data3

FROM      Clients AS c, CoiClasses AS cc, Users AS u, UsersAccess AS ua, ClientData AS cd

WHERE     cc.CoiClassID = c.CoiClassID      AND   ua.UserID = u.UserId        AND
          ua.CoiClassID = cc.CoiClassID     AND   ua.ClientID = c.ClientID    AND
          ua.ClientID = cd.ClientID         AND   ua.ReadAccess = 1           AND

          cc.CoiClass = @CoiClass           AND   c.client = @Client          AND

          u.User = @UserName;

UDF for updating the access rights are called after this select query
```

6.2.2 Implementation of *-Property of CWP

For implementing the *-property the read access is granted to the user by the simple security property, whenever the user requests for some write access the WriteAccess column is updated by the UDF and is shown Table 9:

Table 9. TD for updating the write access rights

ClientData	(UserID,User, CoiClassID, CoiClass, ClientID, Client, Data1, Data2, Data3):-
	Users (UserID, Session.User),
	CoiClass (CoiClassID, @CoiClass),
	Clients (ClientID, @Client),
	UsersAccess (UserAccessID, UserID, CoiClassID, ClientID,ReadAccess, _),
	ClientData (CdID, CoiClassID, ClientID, Data1, Data2, Data3)
	del.UserAccess (_, UserID, CoiClassID, ClientID, _, _),
	ins.UserAccess (_, UserID, CoiClassID, ClientID, 1, 1)

The equivalent SQL SELECT query which actually implements the *-property of CWP after compiling TD Rule is shown in Table 10.

In order to verify our implementation of CWP in RDBAC, we have discussed and executed different scenarios related to the example of CWP presented in the Section 1. The data for the scenarios are shown in Table-11 to Table-15.

Table 10. Equivalent SELECT Query of *-Property of CWP

```
SELECT    u.User, u.UserID , cc.CoiClass, cc.CoiClassID, c.Client, c.ClientID, cd.Data1,

          cd.Data2, cd.Data3

FROM      Clients AS c, CoiClasses AS cc, Users AS u, UsersAccess AS ua, ClientData AS cd

WHERE     cc.CoiClassID = c.CoiClassID      AND   ua.UserID = u.UserID        AND
          ua.CoiClassID = cc.CoiClassID     AND   ua.ClientID = c.ClientID    AND
          ua.ClientID = cd.ClientID         AND   ua.ReadAccess = 1           AND

          cc.CoiClass = @CoiClass           AND   c.client = @Client          AND

          u.User = @UserName;

UDF for updating the access rights are called after this select query
```

Table 11. CoiClasses

CoiClassID	CoiClassName
1	Bank
2	Oil

Table 12. Users

UserID	User
1	a
2	b
3	c

Table 13. Clients

Coi Client ID	Client	Coi Class ID
1	Bank1	1
2	Bank2	1
3	Bank3	1
4	Oil1	2
5	Oil2	2
6	Oil3	2
7	Oil4	2

Table 14. ClientData

Cd ID	CoiClassID	CoiClientID	Data1	Data2	Data3
1	1	1	Bank11	Bank12	Bank13
2	2	4	Oil11	Oil12	Oil13
3	2	5	Oil21	Oil22	Oil23

Table 15. UsersAccess

UserAccessID	UserID	CoiClassID	CoiClientID	ReadAccess	WriteAccess
1	1	1	1	1	0
2	1	1	2	1	0
3	1	1	3	1	0
4	1	2	4	1	0
5	1	2	5	1	0

a) Consider the first scenario, in which a new user may freely choose any dataset i.e. Bank1, Oil1 and Oil2 datasets from the database according to his request. Whenever a new user came into the system it is first entered into the *Users* table, and as an initial setup he is given read access on all the clients data. This is achieved by setting the *ReadAccess* column of table 15 to '1' for all the records related to this new user. In this way a user can initially selects any dataset of any client from the database. But after the initial access of a dataset the access rights are updated i.e. all the other clients of that particular conflict of interest (Coi) class are set to '0' except the accessed client data.

b) In the second scenario, a new user accesses the Oil1 dataset first and now possesses information about the Oil1 dataset. Later the same user requests to access to the Bank1 dataset, this is quite permissible since the Bank1 and Oil1 datasets belong to different Coi classes. The user first selects the Oil1 dataset, the *ReadAccess* of this dataset is set '1' and the *ReadAccess* of all the other clients of this Coi class are set to '0'. Now the user can only access the Oil1 dataset as read access to all the other datasets in this Coi class are rejected. But the user can access any other client which lies in any other Coi class, as the *ReadAccess* column of table 15 was initially set to '1' for all the clients. It was updated after the first access only for that particular Coi class and for the remaining Coi classes it remained unchanged. Now the user requests to access the client data of Bank1 which lies in the Bank Coi class. According to initial set up it is permissible and after read access is granted to Bank1 dataset the same procedure is applied for updating the access rights of this Coi class as after the first read access.

c) In the third scenario, the same user now requests to access the Oil2 dataset which must be denied. Since a conflict does exist between the requested dataset Oil2 and the one which is already possessed dataset Oil1 by the user. As discussed in the second scenario, when the user first access the Oil1 dataset the *ReadAccess* of all other clients in this Coi class are set to '0' means that read access to all clients in this Coi class are rejected. So when this user wants to access the Oil2 dataset the access is denied automatically.

d) In the fourth scenario, a new user first accesses the dataset of Bank1 before the Oil1. This will the same as the second scenario, the user has initially the rights to access any dataset so he accesses Bank1 and then the same process can be applied to update the read access rights and similarly he can access Oil1 after that because this client belong to entirely different Coi class. So the results would be the same as expected by the simple security property of CWP.

e) In the fifth scenario, the user accesses Oil2 before the request to access Oil1 dataset and the restrictions would be quite different. The access to Oil1 dataset would be denied and the user would possess Oil2 and Bank1. In this case the user first accesses Oil2 so the access rights of this user are updated for all the clients of this Coi class i.e. *ReadAccess* column of table 15 is set to '0' except the Oil2 client dataset. Now if the user wants to access the Oil1 dataset the request is denied because there would be a conflict of interest if the access is granted to the user. The user can easily access Bank1 as there is no change in the access rights because Bank1 belongs to a different Coi class. The access rights are updated after accessing the Bank1 dataset. So after executing this scenario the user possesses only the datasets of Oil2 and Bank1.

f) In the sixth scenario, there are 3 clients in one Coi class the minimum number of users which access all the clients datasets would be same as the number of clients in that Coi class. Because a user can only be granted access to only client in one Coi class otherwise there would be conflict of interest among the users. The updating of access rights as discussed in the first and second scenarios automatically enforces this restriction of minimum users for a Coi class.

g) In the seventh scenario, we discuss the *-property of the CWP. Whenever a user wants to get the write access on client dataset, he should have the read access by following the simple security property of CWP. After that he is granted write access to that client dataset and their read access rights are updated accordingly. Consider a user has read access to Oil1 data after the second scenario, now he wants to get the write access on this client dataset. The *WriteAccess* column of table 15 is set to '1' for this client dataset and also the *ReadAccess* column is also updated i.e. set to '0' for all the clients except Oil1. Because according to *-property of CWP "no object can be read which is in a different company dataset to the one for which write access is requested". By using the above discussed mechanism our proposed system implements the *-property of CWP.

6.3 RBAC

The details of the RBAC are discussed in section 2.4 of chapter 2, for implementing this model in RDBAC we have used the following database schema shown in table 16:

Table 16. Database Schema for Implementation of RBAC

No.	Table Name	Columns
1.	Users	UserID:- integer, UserName:- nvarchar, PatientID:- integer, EmpID:- integer
2.	Roles	RoleID:- integer, RoleName:- nvarchar, IsActive:- integer
3.	Permissions	PermissionID:- integer, Operation:- integer, PerName:- nvarchar
4.	RolePermissions	RolePerID :- integer, RoleID:- integer, PermissionID:- integer
5.	UserRoles	UserRoleID :- integer, UserID :- integer, RoleID :- integer
6.	Employees	EmpID :- integer, SSN :- nvarchar, FullName :- nvarchar
7.	Patients	PatientID :- integer, PatientName :- nvarchar, CaredBy :- integer
8.	ContactInformation	ContactID :- integer, EmpID :- integer, PatientID :- integer, Email :- nvarchar, CompleteAddress :- nvarchar, TelephoneNo :- nvarchar
9.	Prescription	PrescriptionID :- integer, PatientID :- integer, PrescribedBy :- integer, DatePrescribed :- datetime, DateFilled :- datetime, DrugName :- nvarchar, Quantity :- integer

The "*Users*" table contains list of users in the system. The identified roles in the system are stored in the table named *"Roles"*. The list of permissions are stored in table *"Permissions"* e.g. read, write etc. The table *"RolePermissions"* contains the reference to roles and permissions which is used to store the information about the allowed permissions to the roles i.e. assigned permissions to roles. The table named *"UserRoles"* contains the reference to users and roles used to store the information that each user is assigned to which role(s) i.e. one user may be assigned to more than one roles. The *"Employees"* table contains the information about the employees in the system. As the tables from sr. no. 1- 6 in Table-16 are used for the implementation of RBAC, the remaining tables in table 16 are specific to an environment. For instance we have chosen the medical database which can store the contact information of patients and the prescriptions written by the doctors for patients. In the following section, we will discuss the implementation of the policy *"Doctors may view the contact data of the patients"* in an RBAC environment. Table 17 and 18 shows this policy written in TD and its equivalent SQL SELECT query respectively:

Table 17. TD for Policy "Doctors may view the contact data of patients"

```
ContactInfo     (PatientID, PatientName, Email, CompleteAddress, TelephoneNo):-
                Users (UserID, @UserName, _, EmpID),
                Employees (EmpID, _, _),
                Patients (PatientID, PatientName, _),
                Roles (RoleID, 'Doctors', 1),
                Permissions (PermissionID, 1, _ ),
                RolePermission(_, RoleID, PermissionID),
                UserRoles(_, UserID, RoleID),
                ContactInformation(ContactID, EmpID, PatientID, Email, CompleteAddress,
                TelephoneNo)
```

Table 18. Equivalent SELECT Query of "Doctors may view the contact data of patients"

SELECT	p.PatientID, p.PatientName, c.Email, c.CompleteAddress, c.TelephoneNo
FROM	Users u, Employees e, Patients p, Roles r, Permissions pr, RolePermissionrp, UserRolesur, ContactInformation c
WHERE	u.EmpID = e.EmpID AND r.RoleID = rp.RoleID AND pr.PermissionID = rp.PermissionID AND u.UserID = ur.UserID AND r.RoleID = ur.RoleID AND p.PatientID = c.PatientID AND u.UserName = @UserName AND r.RoleName = 'Doctors' ;

Another policy *"Doctors may view the contents of prescription of patients for whom they have written prescriptions"* for RBAC is shown in the table 18 and 19 respectively.

Table 19. TD for Policy "Doctors may view the contents of prescription of patients for whom they have written prescriptions"

view.Prescription	(PrescriptionID, PatientID, PatientName, PrescribedBy, DatePrescribed, DateFilled, DrugName, Quantity):- Users (UserID, @UserName, _, EmpID), Employees (EmpID, _, _), Patients (PatientID, PatientName, _), Roles (RoleID, 'Doctors', 1), Permissions (PermissionID, 1, _), RolePermission (_, RoleID, PermissionID), UserRoles (_, UserID, RoleID), Prescription (PrescriptionID, PatientID, PrescribedBy, DatePrescribed, DateFilled, DrugName, Quantity)

Table 20. Equivalent SQL SELECT query of "Doctors may view the contents of prescription of patients for whom they have written prescriptions"

```
SELECT   p.PatientID,   p.PatientName,   u.UserName,   ps.PrescriptionID,   ps.DatePrescribed,
         ps.DateFilled, ps.DrugName, ps.Quantity, ps.Refills

FROM     Users u, Employees e, Patients p, Roles r, Permissions pr, RolePermission rp,
         UserRolesur, Prescription ps

WHERE    u.EmpID = e.EmpID                  AND     r.RoleID = rp.RoleID          AND


         pr.PermissionID = rp.PermissionID  AND     u.UserID = ur.UserID          AND

         r.RoleID = ur.RoleID               AND     p.PatientID = ps.PatientID    AND

         u.UserID = ps.PrescribedBy         AND     u.UserName = @UserName        AND

         r.RoleName  = 'Doctors'
```

6.4 TRBAC

In order to implement the Temporal RBAC policy, we use the same schema as shown in table 16 with one more table named *TimeConstraint* for storing the temporal information about the roles as shown in table 21:

Table 21. Database Schema for Implementation of Simple TRBAC

No.	Table Name	Columns
1.	TimeConstraint	TCID :- integer, RoleID :- integer, TCName :- nvarchar, StartDate :- datetime, EndDate :- datetime, StartTime :- nvarchar, EndTime :- nvarchar

The column named *TCName* is the name of time constraints e.g. night time, day time, part time etc. The columns named *StartDate* and *EndDate* are used for storing the start and end date of the time constraint. Similarly the timing information of the activation of roles are stored in columns named *StartTime* and *EndTime* e.g. 09:00 and 17:00 respectively. In the following section, we will discuss the implementation of the policy *"Doctors on night duty may view the contact information of patients during night time"* in an TRBAC environment. Table 22 and 23 shows this policy written in TD and its equivalent SQL SELECT query respectively:

Table 22. TD for Policy "Doctors on night duty may view the contact information of patients during night time"

```
view.ContactInfo   (PatientID, PatientName, Email, CompleteAddress, TelephoneNo):-
                   Users (UserID, @UserName, _, EmpID),
                   Employees (EmpID, _, _ ),
                   Patients (PatientID, PatientName, _),
                   Roles (RoleID, 'DoctorsOnDayDuty ', 1),
                   Permissions (PermissionID, 1, _ ),
                   RolePermission (_, RoleID, PermissionID),
                UserRoles (_, UserID, RoleID),

                ContactInformation (ContactID, EmpID, PatientID, Email,
                CompleteAddress, TelephoneNo),

                TimeConstraint (_, RoleID, 'Night', StartDate, EndDate, StartTime,
                EndTime),

                GETDATE() >= StartDate,

                GETDATE() <= EndDate,

                GETTIME()  >= StartTime,

                GETTIME()  <= EndTime
```

Table 23. Equivalent SQL SELECT query of "Doctors on night duty may view the contact information of patients during night time"

```
SELECT   p.PatientID, p.PatientName, c.Email, c.CompleteAddress, c.TelephoneNo

FROM     Users u, Employees e, Patients p, Roles r, Permissions pr, RolePermission rp,
         UserRoles ur, ContactInformation c, TimeConstraint tc

WHERE    u.EmpID = e.EmpID              AND     r.RoleID = rp.RoleID       AND

         pr.PermissionID = rp.PermissionID  AND  u.UserID = ur.UserID     AND

         r.RoleID = ur.RoleID           AND     p.PatientID = c.PatientID  AND

         tc.RoleID = r.RoleID                   AND

         GETDATE   ()   BETWEEN     StartDate       AND       EndDate
         AND

         GETTIME  ()   BETWEEN   StartTime     AND              EndTime
         AND

         u.UserName =  @UserName              AND

         r.RoleName  =  'DoctorsOnDayDuty' ;
```

FORMAL VERIFICATION THOROUGH MODEL CHECKING

In this chapter we will verify our implemented BLP model (Holzmann G. , 1990) and (Holzmann G. , April 93) using the (SPIN) model checker which uses the "Process Meta Language PROMELA" as a modeling language. In model checking we verify models instead of actual systems which are high level descriptions of actual systems. In order to verify the database system, it is modeled having the same characteristics of original database using some modeling language.

7.1 Constructing the Model

In order to model the schema of BLP in Promela which is presented in Table 1 of section 6.1.1, every database table is defined as a typedef in the model. The BLP schema modeled in Promela is shown in Table-24:

Table 24. BLP Schema modeled in Promela

```
1.   typedef  Category { int CatID; int category;};
2.   typedef  Clearance { int clearanceID; int clearance; };
3.   typedef  Data { int DataID; int Data1; int Data2; int Classification; int DCatID; int y; };
4.   typedef  Lattice { int LID; int CatSetID; int ParentID; int x; };
5.   typedef  SetCategories { int ID; int setCategories; };
6.   typedef  Users { int userID; int users1; int clearance; int UcatID; };
```

The arrays of these typedef are then created which models the records in the database tables which are shown Table-25.

Table 25. Arrays of typedef for modeling the records in tables

```
1.    Category category [number_of_Catgories];
2.    Clearance clearance [number_of_Clearance];
3.    Data data [number_of_Data];
4.    Lattice lattice [number_of_Lattice];
5.    SetCategories setcategories [number_of_SetCategories];
6.    Users users [number_of_Users];
```

In order to model the functionality of the SQL query for simple security property of BLP shown in Table-3 of section 6.1.2, three loops are used for implementing the functionality of joins in the WHERE clause. The Promela code for modeling this functionality is shown in Table-26.

Table 26. Promela code for modeling the functionality of simple security property of BLP

```
int i,j=0;
---
do
        :: lattice [i].CatSetID == CUser.UcatID ->  lattice[i].x =1;
        if
                :: i == number_of_Lattice -> break;
                :: else ->; i++;
        fi
od;
---
for (i : 0..number_of_Lattice) {
        do
                :: lattice [i].x == 1 && lattice[i].ParentID == data[j].DCatID
                ->  data[j].y =1;
                :: j == number_of_Data -> break;
                :: j++;
        od;
}
---
for (i : 0..number_of_Data) {
        if
                :: (data[i].Classification <= CUser.clearance) && (data[i].y
                == 1) -> {
                        ResultData.Data1 = data[i].Data1;
                        ResultData.Data2 = data[i].Data2;
                        ResultData.Classification = data[i].Classification;
        }
        fi
}
```

7.2 Verification through SPIN model checker

After molding the query for simple security property of BLP, the created schema in Table-24 needs to be populated with data. Similarly, another array named *ExpectedResult* is created that contains manually calculated expected results of simple security property. The model of simple security property query as shown in Table-26 is executed on test data and the results are saved in *ResultData* array.

In order to verify the generated results, *Assert* statement of Promela is used. This statement is always executable and takes any valid Promela expression as its argument. The expression is evaluated each time the statement is executed. If the expression evaluates to false (or, equivalently, to the integer value zero), an assertion violation is reported during verifications with SPIN. The expected results are compared with generated results of model by the use of following *Assert* statement:

Assert (ExpectedResult.Data1 = ResultData.Data1)

Assert (ExpectedResult.Data2 = ResultData.Data2)

The results of this model shows that it is valid and does not violates simple security property of BLP. The model generated results are according to expected results and we tested for positive and negative results also no violation in accordance with the BLP security property found.

PERFORMANCE EVALUATION

In this chapter we will evaluate the performance of our implementation of the translation algorithm for translating TD policies into equivalent SQL SELECT queries by calculating the translating time. We also evaluated the performance of implementation of BLP, CWP, RABC and temporal policies by calculating their execution time of running equivalent SQL SELECT queries generated by our algorithm implementation.

In order to calculate execution time of translated SQL queries we used Microsoft's SQL Server 2008 Express R2 database management system (DBMS) running on a 2.4 GHz Intel Core i5 machine with Windows 7 Home Premium 64-bit operating system. The calculation of execution time was performed by using external applications written in C# and compiled by Microsoft's Visual C# 2008 compiler version 3.5. Both the test applications and the DBMS were run locally on the same machine and the network latency is not included for the execution time.

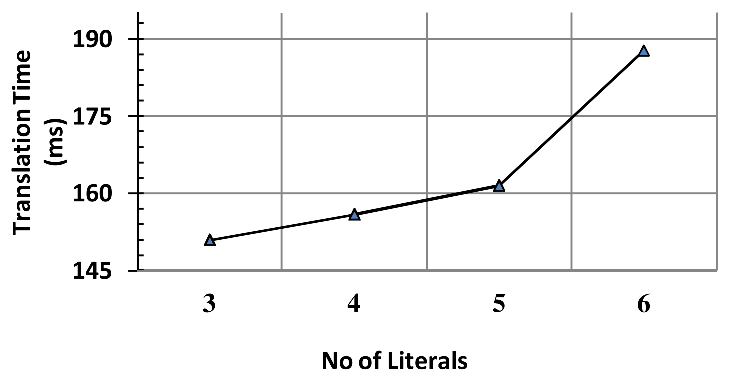

Figure-4 : Performance of TD to SQL Compiler

The performance of translating algorithm discussed in section 4.1 is evaluated on the basis of number of literals in the head and body of TD rules. This algorithm was implemented in C#.Net which reads the database schema directly from the target database given the target database name and location. The tests were performed by providing TD rules having different number of literals. The time taken for translating the TD rules to SQL queries was calculated in milliseconds and is shown in Figure-4. Each test was performed 20 times to get the stable values of execution times, and an average calculated for translation times for each 3,4,5 and 6 number of literals in TD rules. Our tests show that the translation time slightly increases with the increase in number of literals in the TD rules. Further, the increase in number of variables and constants in each literal of TD rule also effects the translation time because it depends upon the number of columns in the target database tables.

In order to evaluate the performance of generated SQL SELECT queries for each of the polices we developed prototype applications which call the queries wrapped in UDFs as discussed in section 4.1. For each implemented policy the tests were performed on separate datasets because the database schemas are implementation specific. The arrangement of the datasets for BLP is shown in Table-27.

Table 27. Datasets for BLP

No. of Records in		
Users	Lattice	Data (No. of Records)
10	12 (3 categories and 4 clearance)	100, 1000, 2000, 10,000, 20,000, 50,000, 100,000
50	43 (4 categories and 5 clearance)	
100	162 (5 categories and 6 clearance)	

We calculated the execution time of queries by using varying number of users, categories, clearance levels and number of data records. The size of lattice depends upon number of categories and clearance levels. For instance if there are 3 categories and 4 clearance levels than the size of Lattice tables will be 12 records, the subset operator has been used for creating the lattice. We use combinations such as {Users = 10, Lattice Size = 12 and Data = 100 records}, {Users = 10, Lattice Size = 12 and Data = 1000 records} and {Users = 10, Lattice Size = 12 and Data = 2000 records} and so on. The execution time of all the combinations are shown in Figure-5 to Figure-7 for simple security property of BLP and Figure-8 to Figure-10 for *- property of BLP. Each test was performed 20 times and then average values are taken for plotting the graphs. It was observed in our implementation the number of users and the size of the lattice has reasonably small impact on the execution time of queries. However, the execution time is linear in the size of the database (number of the records) which is typical of database systems.

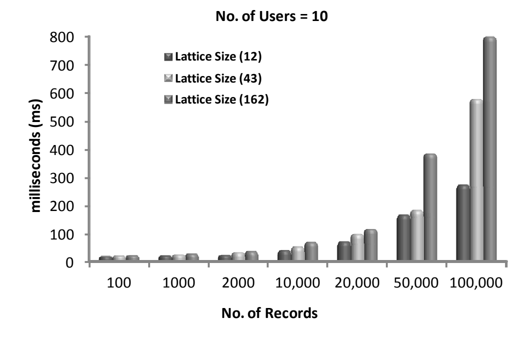

Figure-6: Simple Security Property of BLP (10 Users)

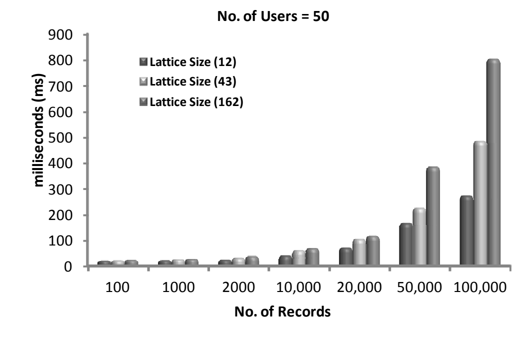

Figure-5: Simple Security Property of BLP (50 Users)

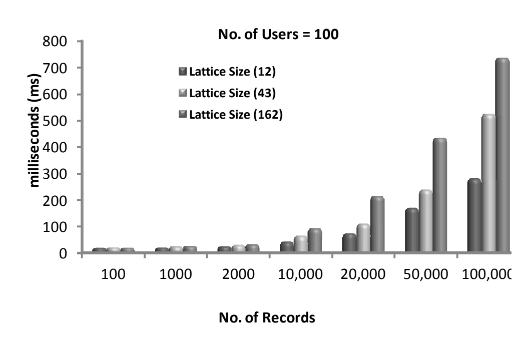

Figure-7: Simple Security Property of BLP (100 Users)

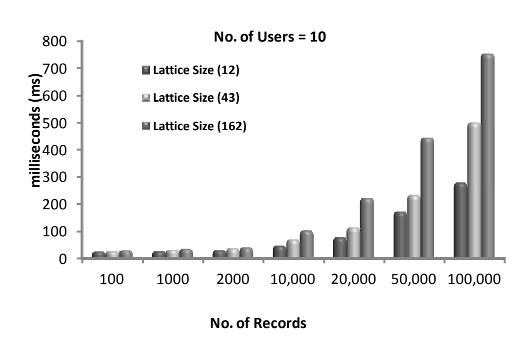

Figure-8: *-Property of BLP (10 Users)

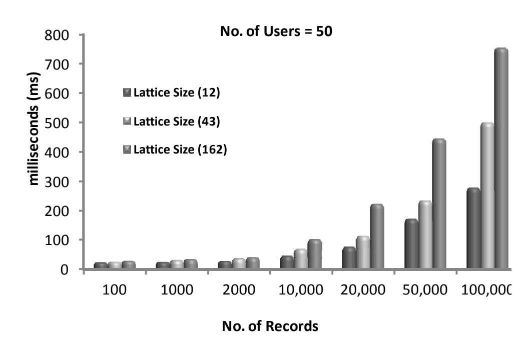

Figure-10: *-Property of BLP (50 Users)

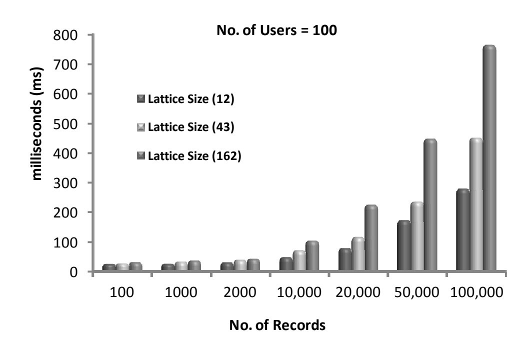

Figure-9: *-Property of BLP (100 Users)

In order to evaluate the execution time of CWP, the arrangement of datasets is shown in Table-28.

We use different combinations of number of Users, CoiClasses, Clients size and number of records for calculating the execution time of both properties of CWP e.g. {Users = 10, CoiClasses = 10, Clients = 100 and Data = 100 records}, { Users = 10, CoiClasses = 10, Clients

= 100 and Data = 1000 records } and { Users = 10, CoiClasses = 10, Clients = 500 and Data = 100 records } and so on. The execution time of all the combinations are shown in Figure-11 to Figure-16 for simple security property of CWP and Figure-17 to Figure-22 for *- property of CWP. Each test was performed 20 times and then average values are taken for plotting the graphs. It was observed in our implementation the number of users, CoiClasses and Clients has reasonably small impact on the execution time of queries. However, execution time depends upon the number of records in database which majorly affects execution time and typical to database systems.

Table 28. Datasets for CWP

No. of Records in			
Users	CoiClasses	Clients	Data (For each dataset)
10	50	100	100, 1000, 10,000, 20,000, 50,000, 100,000
10	50	100	
50	500	1000	

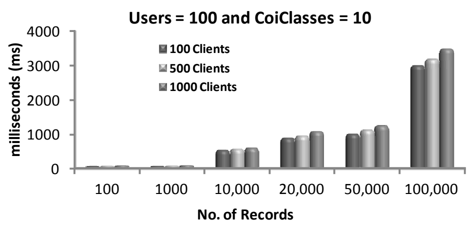

Figure-12: Simple Security Property of CWP (100 Users)

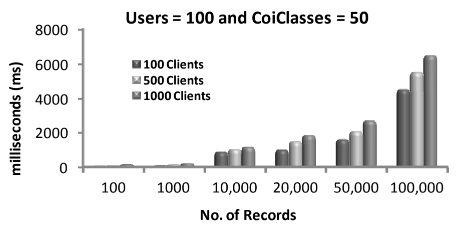

Figure-11: Simple Security Property of CWP (100 Users)

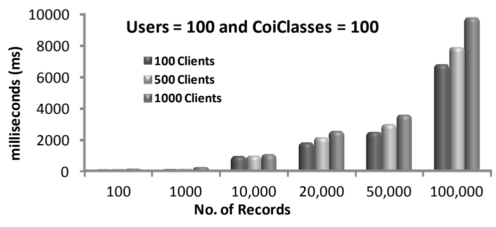

Figure-14: Simple Security Property of CWP (100 Users)

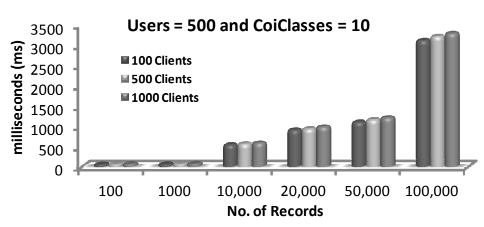

Figure-13: Simple Security Property of CWP (100 Users)

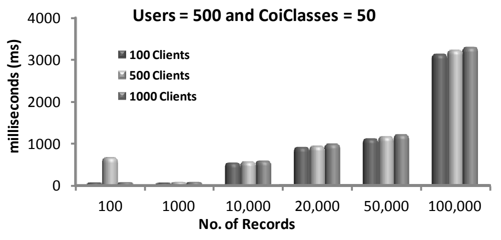

Figure-16: Simple Security Property of CWP (500 Users)

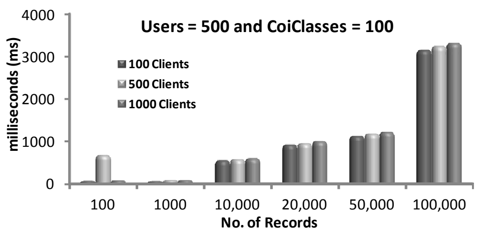

Figure-15: Simple Security Property of CWP (500 Users)

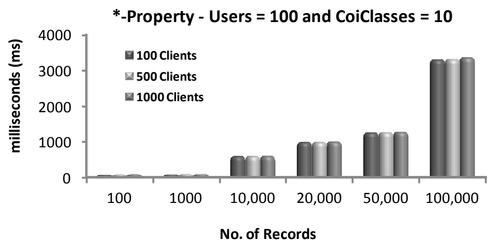

Figure-18: *- Property of CWP (100 Users)

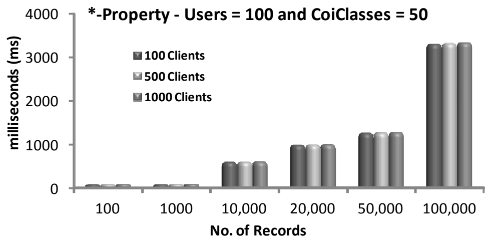

Figure-17: *- Property of CWP (100 Users)

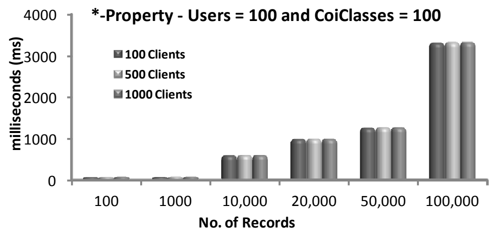

Figure-20: *- Property of CWP (100 Users)

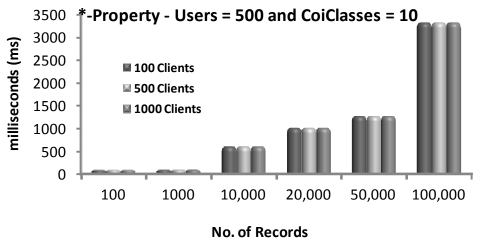

Figure-19: *- Property of CWP (500 Users)

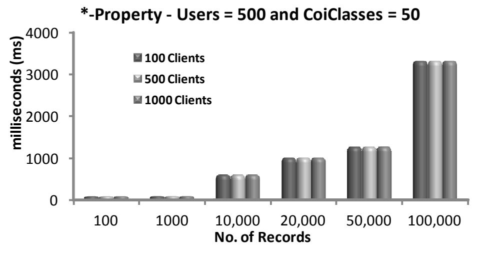

Figure-22: *- Property of CWP (500 Users)

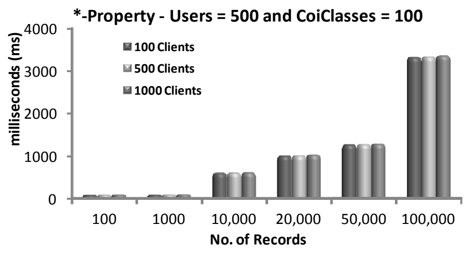

Figure-21: *- Property of CWP (500 Users)

Table 29. Datasets for RBAC

No. of Records in			
Users	Roles	Patients	ContactInfo.
10	10	100, 1000, 10,000, 20,000, 50,000, 100,000	100, 1000, 10,000, 20,000, 50,000, 100,000
50	20		
100	30		

The same strategy of different combinations of data for calculating the execution time of RBAC and TRBAC policy is used as in BLP and CWP and the dataset is shown in Table-29. Some of combinations of Users, Roles and number of patients records for calculating execution time are {Users = 10, Roles = 10, Patients and ContactInfo = 100 records}, {Users = 10, Roles = 10, Patients and ContactInfo = 1000 records} and {Users = 10, Roles = 10, Patients and ContactInfo = 10,000 records} and so on. The calculated execution time is shown in Figure-23 to Figure-25 for RBAC policy and Figure-26 for TRBAC. The same criteria of 20 tests were used for plotting the graphs. It was observed in our implementation the number of users and roles and grouping of users in different roles does not put significant effect on the execution time. However, the number of the records in the data is the major factor effecting execution time.

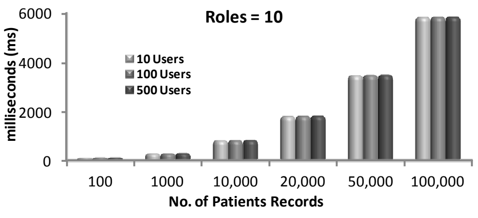

Figure-23: RBAC (10 Roles)

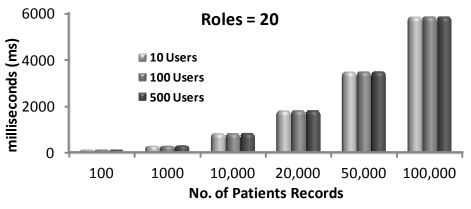

Figure-24: RBAC (20 Roles)

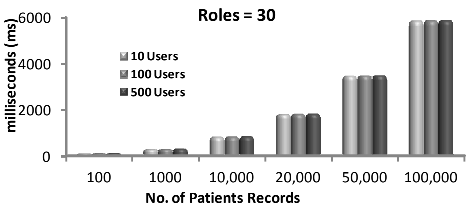

Figure-25: RBAC (30 Roles)

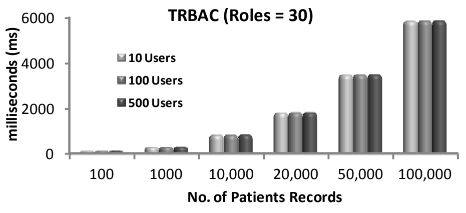

Figure-26: TRBAC (30 Roles)

CONCLUSIONS AND FUTURE WORK

In our research work we have implemented BLP, CWP, RBAC and TRBAC in the RDBAC framework. For this purpose we have provided the database schemas for each implemented policy and TD rules which are based on these schemas. We have implemented the TD to SQL query translation algorithm with inclusion of new feature of translating the insertion and deletion predicates into SQL UPDATE statement. The evaluation of generated SQL queries is performed by developing external prototype applications for each policy. It has been observed during the implementation that some initial setup may be required for some type of policies to be implemented in RABAC framework e.g. CWP simple security property discussed in section 6.2.1. The complete functionality of SQL can be mapped on TD by the inclusion of semantics for the negation in TD rules i.e. TD rules only provide the positive authorizations and no semantics for the denying logic is available. The inclusion of negation semantics provides more flexibility in writing policies and also improves the performance of the policies. Similarly the inclusion of semantics of the SQL sub queries will improve the performance significantly and in this way we can bypass the recursion in some policies.

There is a need of a mechanism or visual aided tool that helps the administrators to aid in writing TD rules from plain English language policies. The formal analysis of the security policies provided in the (Olson, Gunter, & Madhusudan, Oct. 2008) does not cover all type of the policies. The read and append only policies can be formally analyzed but the policies which have side effects cannot be analyzed so a formal analysis mechanism needs to be developed which analyze each policy for its safeness before compiling into the SQL queries.

REFERENCES

Agrawal, R., Bird, P., Grandison, T., Kieman, J., Logan, S., & Rjaibi, W. (2005). Extending relational database systems to automatically enforce privacy policies. *In Proceedings of 21st International Conference on Data Engineering (ICDE' 05)* (pp. 1013-1022). Washington, DC, USA: IEEE Computer Society.

Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. (2002). Hippocratic databases. *In Proceedings of the 28th international conference on Very Large Data Bases (VLDN'02)*, (pp. pp. 143–154.).

Ahn, G., & Sandhu, R. (2000). Role-based authorization constraints specification. *ACMTrans. Inf. Syst.* , Sec. 3, 4 (Nov.).

Bell, D., & LaPadula, L. (Mar. 1973). *Secure Computer Systems: Mathematical Foundation.* Technical Report MTR-2547, Vol. I MITRE Corporation, Bedford, MA.

Bell, D., & LaPadula, L. (Mar. 1975). *Secure Computer Systems: Unified Exposition and Multics Interpretation.* Bedford, MA: Technical Report MTR-2997 Rev. 1, MIRTE Corporation.

Bertino, E., & Sandhu, R. (Jan-Mar 2005). *In Proceedings of the IEEE Transactions on Dependable and secure computing, vol. 2, No. 1* .

Bertino, E., Bonatti, P., & Ferrari, E. (2000). TRBAC: A temporal role-based access control model. *In Proceedings of the Fifth ACM Workshop on Role Based Access Control*, (pp. pp 21–30).

Bobba, R., Fatemieh, O., Khan, F., & Gunter, C. K. (2006). Using attribute-based access control to enable attribute-based messaging. *In Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)* (pp. pp. 403–413.). IEEE Computer Society.

Bonner, A. J. (1998). Transaction datalog: A compositional language for transaction programming. *Lecture Notes in Computer Science, 1369:373–395* .

Brewer, D. D., & Nash, D. M. (May 1989). The chinese wall security policy. . *In IEEE Symposium on Security and Privacy*, (pp. pp. 206–214). Oakland, CA.

Chaudhuri, S., Dutta, T., & Sudarshan, S. (April 2007). Fine grained authorization through predicated grants. *ICDE 2007. IEEE 23rd International Conference on Data Engineering* (pp. pp. 1174–1183). IEEE.

De Capitani di Vimercatii, S., Foresti, S., Jajodia, S., Paraboschi, S., & Samarati, P. (2008). Assessing query privileges via safe and efficient permission composition. *In Proceedings of the 15th ACM conference on Computer and communications Security (CCS'08)* (pp. pp. 311-322.). New York, NY, USA: ACM.

Feuerstein, S., & Pribyl, B. (Oct. 2009). *Oracle PL/SQL Programming.* O'Reilly Media, 5th ed.

Goodwin, R., Goh, S., & Wu, F. (2002). Instance-level access control for business-to-business electronic commerce. *IBM Systems Journal 41(2)* , 303–321.

Griffiths, P. P., & Wade, B. W. (1976). An authorization mechanism for a relational data base system. . *In: SIGMOD '76: Proceedings of the 1976 ACM SIGMOD international conference on Management of data* (pp. pp. 51–51). New York, NY, USA : ACM.

Hajiyev, E. (Sep. 2005). CodeQuest: Source Code Quering with Datalog. *MSc Thesis, Oxford University Computing Laboratory* . Available at http://progtools.comlab.ox.ac.uk/projects/codequest/.

Hajiyev, E., Verbaere, M., & de Moor, O. (2006). Codequest: Scalable source code queries with datalog. *Thomas, D. (ed.) ECOOP 2006. LNCS vol. 4067* , pp. 2-27.

Holzmann, G. (1990). *Design and validation of computer protocols. , ISBN:0-13-539925-4.* Prentice Hall Inc.

Holzmann, G. (April 93). Design and validation of protocols: a tutorial. *Special issue on specification, testing and verification, In Computer Networks and ISDN Systems, Volume 25, Issue 9* , pages 981-1017.

Jahid, S., Gunter, C. A., Hoque, I., & Okhravi, H. (2011). MyABDAC: Compiling XACML Policies for Attribute-Based Database Access Control. *In Proceedings of the first ACM conference on Data and Application Security and Provacy* (pp. pp. 97-108). ACM.

Jahid, S., Hoque, I., Okhravi, H., & Gunter, C. A. (2009). Enhancing Database Access Control with XACML policy.

LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., & DeWitt, D. (2004). Limiting disclosure in Hippocratic databases. *In proceedings of 13th internatinal conference on very large data bases (VLDB Endowment)*, (pp. pp. 108-119).

Lunt, T. F., Denning, D. E., Schell, R. R., & Heckman. (1990). The seaview security model. *IEEE Trans. Softw. Eng. 16(6)* , 593–607.

Olson, L. E., Gunter, C. A., & Madhusudan, P. (Oct. 2008). A formal framework for reflective database access control policies. *In Proceedings of CCS'08.* Alexandria, VA.

Olson, L. E., Gunter, C. A., Cook, W. R., & Winslett, M. (2009). Implementing reflective access control in SQL. *In proceedings of DBSec'09.* Montreal, QC.

Opyrchal, L., Cooper, J., Poyar, R., & Le, B. (April, 2011). Bouncer: Policy-Based Fine Grained Access Control in Large Databases. *International Journal of Security and Its Applications* , Vol. 5 No. 2.

Oracle-Corporation. (June 2005). *Oracle Virtual Private Database.* Oracle Corporation.

Rashid, Z., Basit, A., & Anwar, Z. (2010). TRDBAC: Temporal Reflective Database Access Control. *In Procedings of ICET.* Islamabad, PAK: IEEE.

Rizvi, S., Mendelzon, A., Sudarshan, S., & Roy, P. (2004). Extending query rewriting techniques for fine-grained access control. *In Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (pp. pp. 551–562). New York, NY, USA: ACM.

SPIN. (n.d.). *http://www.spinroot.com. SPINs basic manual can be found at the following address: http://www.spinroot.com/spin/Man/Manual.html*. Retrieved from SPIN: http://www.spinroot.com

Stonebraker, M., & Wong, E. (1974). Access control in a relational data base management system by query modification. . *In Proceedings of the 1974 annual conference* (pp. pp. 180–186.). New York, NY, USA : ACM.

Zhang, Z., & Mendelzon, A. O. (2005). Authorization views and conditional query containment. *In 10th International Conference on Databaase Theory (ICDT 2005)*, (pp. pp. 259-273).