Security Analysis in Multi-level Databases

Submitted in partial fulfillment of the requirements of the degree of

Master of Technology (M.Tech)

by

Aniket Kuiri

Roll no. **163050059**

Supervisor:

Prof. R.K. Shyamasundar

Department of Computer Science & Engineering

Indian Institute of Technology Bombay

2018

Dissertation Approval

This project report entitled **"Security Analysis in Multi-level Databases"**, submitted by **Aniket Kuiri** (Roll No. **163050059**), is approved for the award of degree of **Master of Technology (M.Tech)** in Computer Science & Engineering.

Prof. R.K. Shyamasundar
Dept. of CSE, IIT Bombay
Supervisor

Prof. G. Sivakumar
Dept. of CSE, IIT Bombay
External and Internal Examiner

Prof. Bernard Menezes
Dept. of CSE, IIT Bombay
Internal Examiner

Date: 29 June 2018

Place: Mumbai

Declaration of Authorship

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature: Aniketkuiri

Aniket Kuiri

163050059

Date: 29 June 2018

Abstract

Databases play a crucial role in all major applications around us. Therefore their security is important. Medical or Military databases contain very sensitive information which if leaked can pose threat.There exist several mature database systems with varying claims about their efficiency, utility, and security. PostgreSQL is one of the prominent and widely used open-source database systems, which we extend for its utility, confidentiality, and privacy (i.e., security) guarantees. Classical PostgreSQL uses Views for security that is vulnerable to data leakage. To ensure better security of data in databases various models have been proposed like Bell-La Padula model, Biba Model, Denning's Information Flow model, etc. Here we will discuss about another efficient model RWFM (Reader-Writer Flow Model). In RWFM individual data elements and individual users of the database are assigned classification labels based on which privacy preserving access control is enforced. We will also look into a special type of database known as hippocratic database that facilitates purpose based privacy control. Hippocratic databases use policies to restrict access to data to different users. We will develop an algorithm to determine whether 2 policies are contradictory or not. Finally we will model a popular Conference Management System HotCRP as RWFM and check whether it satisfies all the constraints of the system.

Contents

Chapter 1

Introduction

Database systems are used to store and retrieve data efficiently. Most databases store data that contain vital information. Leak of these information to unintended users of the system is undesirable. Malicious activities can be done by intended users to get sensitive data. So security is of utmost importance in databases. In most scenarios security is enforced by the application level that uses the database. But there exist many scenarios in which a valid set of operations by authorized users of the system can reveal information that is not intended to be revealed.

- If a malicious user can bypass the application level security then he gets to all information directly from the database.
- Application handles lots of things like handling multiple connections at the same time, accepting request from the clients, synchronization, concurrency, etc. Adding security checks at the application level introduces overhead which is time consuming and difficult to manage. Performing security checks in database level is beneficial as at the same time application level can focus on other things thus saving time.

Most databases contain information that are sensitive to some users but not to others. So some users should get whole of the data in the database and some only a part of information. The part of information that is not sensitive enough can be disclosed to a user. This is the notion of Multilevel Database. Formally Multilevel Databases are databases that contain objects (data) with different levels of confidentiality and register subjects (who uses the data) with different abilities. This means objects are given labels

in the database and subjects have some clearance labels to access them. Let there be four classification levels for objects and subjects based on ascending order of confidentiality :

- Unclassified
- Confidential
- Secret
- Top Secret

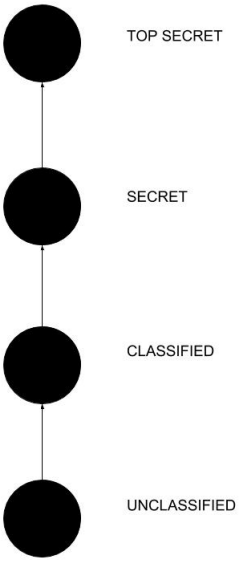

Figure 1.1: Classification labels in order

So if an object is classified as Secret it should not be accessed by a subject with clearance level Unclassified or Classified. This is an example of how security is implemented in multilevel databases.

Let us understand the importance Multilevel Database System with the help of a scenario. Lets take military database as an example. Data stored in military databases are extremely important and requires utmost security. Any leak of important data can compromise the security of a country. A military database may contain lots of information

- personal details of the officers and soldiers

- Location of army bases
- Artillery and other weapons available in a particular army base
- Undergoing operations by the army
- Expenditure in weapons, food, transportation, etc.

All of the above information are very sensitive in the following way :

- Disclosure of phone numbers of few army personnel at a time is not so sensitive but giving information of phone numbers of many personnel is sensitive as it may lead to tracking of base location or approximation of base location based on the initial codes of the phone numbers.
- Sometimes it is required to disclose the location of base camp to some person. But giving information of exact location is too risky. In these cases a concept called sanitization is employed which we will discuss in details later. In this case only approximated location are given and not the exact one.
- It is necessary to give information about the types of artillery and weapons present to someone but giving information of total number of artillery and ammunition is not desirable. It is because any information leak to enemies about the quantity is not good strategically. In terms of database use of aggregate functions is this case is not desirable.
- Similarly information about total expenditure in weapons is also sensitive. Also minute details about date and time of undergoing operations should not to disclosed to anyone other than army personnel as they are highly sensitive data.

Thus we can say that military database contains data of varying sensitivity which needs to be disclosed according to the levels of users. Hence multilevel database security here is desirable.

Multilevel Databases are also known as classful databases as all the objects and subjects have classification labels associated with them and so can be divided into classes

of security labels. On the other hand there are certain types of databases that do not have objects and subjects with labels associated with them. They are known as classless databases. Here we will discuss about some special type of classless databases known as hippocratic databases. In hippocratic databases access to a data element in the database to a user is defined based on the purpose for which the user wants to get access to the data. The creator of a data element in the database has the right to give its access to a different subject (user) for some purpose but not for something other. Lets take an example suppose there is a customer database of a shopping website where each of the customer enter their personal details. Now each of the customer has the right to give access to the data for different purposes to be used by the company. Suppose a customer only wants to allow its data to be used only for shipping purpose and nothing else. He may not want that the company to use his data for market analysis and deriving statistics to improve sales and productivity. Some customers may allow the company to use for the purpose mentioned. There also can be customers who may not want the company to share their personal data to to be shared to some other company while some customer may allow. So it is up to the customers to decide for what purpose their data can be used by the company. This form of privacy is known as purpose based privacy policy which we will discuss in detail later.

In the next chapter we will provide the background of this field and discuss the related work along with the work done specifically on this topic by our group. In chapter 3, we will understand the concept of RWFM and its implementation in this work. We will also analyze its efficiency over other models. In chapter 4 we will discuss about Hippocratic databases and its usage and figure out the possible issues of its original model and solve it using RWFM. In chapter 5, we will discuss look into a Conference Management System named HotCRP and model it into RWFM model and analyze how it is consistent with the original model.

Chapter 2

Background and Related Work

A lot of research has been done on database security. Here we would mention about some of the models along with their advantages and disadvantages. The models are used based on the security requirements of the database.

Ensuring Confidentiality and Integrity through labels Before going into understanding and implementation of the above paper let us understand how basic security works. We will mainly focus on confidentiality and integrity .
Confidentiality: protecting the information from disclosure to unauthorized parties.
Integrity: protecting information from being modified by unauthorized parties.
We will describe Bell-LaPadula model, Biba model and Denning's Information Flow model to understand how confidentiality and Integrity are achieved in Data Security. These can be applied in Database as well.

Bell-La Padula Model To achieve confidentiality the subjects (users) and objects (data) are assigned levels. The levels assignment for objects is termed as security classification and the same for subjects is termed as assigning clearance levels. Now based on them Bell-La Padula model follows transition from one state to another to denote security. The properties of Bell-LaPadula model are :

- No read up. Thus a subject s can read an object o only if L(s) $\geq$ L(o), where L(o) is the classification level of a object o and L(s) is the clearance level of subject s.

- No write down. A subject s can write an object o only if L(s) $\leq$ L(o), where L(o) is the classification level of a object o and L(s) is the clearance level of subject s.
- Discretionary Access Control : Every access should be constrained by Discretionary Access Control as well.

Biba Model Biba model is based on confidentiality. the main properties of Biba model are :

- No write up. A subject s can write an object o only if L(s) $\geq$ L(o), where L(o) is the classification level of a object o and L(s) is the clearance level of subject s.
- No read down. A subject s can read an object o only if L(s) $\leq$ L(o), where L(o) is the classification level of a object o and L(s) is the clearance level of subject s.

Denning's Information Flow Model Denning's Information Flow Model is a lattice form of security model that ensures confidentiality and integrity. Each of the data (objects) and users of the data (subjects) have classification labels and the classification labels form a lattice structure. To ensure confidentiality, a subject gets access to a data only if classification label of the subject is higher than or equal to that of the data and a subject can write to a data only if its classification label is lower than or equal to that of the data. To ensure integrity, a subject gets access to a data only if classification label of the subject is lower than or equal to that of the data and a subject can write to a data only if its classification label is higher than or equal to that of the data. Using Denning's model we can also derive classification labels of objects by using labels of other objects from which it cn be derived.

Security in databases can also be realized with the help of views. In the next paragraph we will look into it in details.

Views for Multilevel Database Security It mainly focuses on the use of views for enforcing privacy in database. It also takes into consideration of multilevel databases where each data object and each user of the database has levels assigned. Based on the levels of each user access to data objects is determined by comparing the users label with

the label of the data object. Initially views for the desired database are created for each of the users classification and corresponding access rights are given as desired. Access rights refers to reading (using **select** statement), writing (**insert** statement), updating (**update** statement), etc.
The above examples demonstrate how data security is done by classification of subjects and objects into levels. Classification is used in the View model of security mentioned in the paper. And it is with this classification of data we achieve a multilevel database owing to the fact that now each data in the database and subject using the database has classification levels assigned.

There are many databases where data elements are not classified into levels. They are known as classless databases. One such type of classless databases are hippocratic databases, which we will discuss next.

Extending RDBMS to automatically enforce privacy policies The main concept here is about privacy in single level or classless database i.e., the data have no labels assigned to them. It mainly focuses on the privacy of personal data present in the database of a user or a group of users. The main concept is to implement fine grained access control (FGAC) policies.

The main features of FGAC implementation are :

- The implementation must solve all the problems within the database itself without the concern of the application using it.
- All the users of the database must be covered in the implementation.
- Implementation of FGAC policies is an overhead so the complexity and maintenance must be minimal.
- It must have the ability to control access to rows, columns and cells in the database as desired.

The type of databases described in this paper are known as Hippocratic databases. In hippocratic databases there are two types of users who access the database. First set of users are known as data owners and others are the users (or subjects) who access data

from the database. Data owners put their personal information into the database while the other users access those data based on certain purposes on which the data owner has given the user(or subject) access to. Lets understand it by a simple example. Suppose there is a database that contains the personal information of the the customers of an online shopping company. So here the customers are the data owners who have given their personal information to the company. Now the data in the database can be used for many purposes by the company. Based on the things bought by any customer the company can recommend other items that the customer can buy. Also the company can give the personal information to other company for do some other activity or simply the company can use the data just to ensure the delivery of the product bought by the customers and nothing else. So we find their can be many different purposes for which data from database can be used. But it should be on customers (or data owners) discretion to allow for what purposes his data should be used by the company. A customer might not be happy if his data is shared to some other company or used for some statistical analysis by the company in finding out type of products being sold at a particular region. So the customer may allow to use his data for those purposes to the company, but some customers might want want to do so and does not want to allow to share the data for those purposes. Hence the notion of purpose is important in this scenario and data access to given to users (or subjects) based on the purposes for which they are given access to.

In SecPostgreSQL: A System for Flow-Secure View, Transaction, Sanitization and Declassication on MLS Database, we worked on multilevel databases and implemented RWFM model in PostgreSQL. I have extended her work into adding few more features of RWFM in PostgreSQL and detecting inconsistencies of classification in multilevel databases which we will discuss in later. Further implementation of hippocratic databases along with its implementation, analysis and detecting inconsistencies is done which we will discuss later.

Chapter 3

Extending PostgreSQL with RWFM Model

RWFM is based upon the lattice model for restricting the information flows. As there are classification labels for data and subjects in Denning's model similarly there are labels in RWFM. The labels are of the form of triplets as <**owner, reader-set,writer-set**>. Owner is the set of subjects who own the object, Reader-set defines the set of readers (subjects who have read access) for the particular subject or object. Writer-set defines the set of writers (subjects who can influence or write) for the subject or object. RWFM is an efficient model to realize the concept of purpose-based privacy policies which we will discuss in details in upcoming chapters.

3.1 Security in Normal Database Systems

Classical Databases use views to control data access. To understand how security is realized using views lets take an example. Suppose there is a medical database as:

Patient Id	**Name**	**Age**	**Condition**	**Date**	**Place**	**Ward no**	**Observer**
A101	Abhi	45	Dengue	10-08-2017	RJV,MUM	B-201	Gautam
A201	Ayush	52	Malaria	09-05-2017	RJV,MUM	C-263	Vipul
B107	Ravi	47	Jaundice	09-05-2017	RJV,MUM	B-213	Arsh
C224	Anik	29	Measles	05-05-2017	RJV,MUM	A-413	B.C. Bell

Table 3.1: Medical Data Table

The database will have lots of users like the patients, doctors, nurses for various purposes:

- Patient can see only their own data. They should not have the privilege to see data of other patients.
- The doctor should have access to all the data in the database.
- A nurse should have access to only patient-id, age, condition, ward no and observer. She should not have access to Name column as a patient would not like to disclose his name to a nurse . Thus name column is sensitive information which Nurse should not have access to.
- A statistic analyzer would need information regarding age, condition and place to derive statistical analysis such as number of people above a particular age suffering from a particular condition, etc. So he should be given access only to the column age, condition and place.

Views are used to give the desired access to the users. Let us see how views are created for nurse and statistic analyzer. Let us define the Views for 2 users: Nurse and Statistics Collector.

- **Nurse :** CREATE VIEW view1 AS SELECT age, condition, data, place, ward no, observer FROM medical_data ;
- **Statistics Collector :** CREATE VIEW view2 AS SELECT age, condition, place FROM medical_data ;

After the views are created access to the views are given using GRANT statements to the respective users as follows:

- GRANT SELECT ON view1 TO nurse ;
- GRANT SELECT ON view2 TO statistics_collector ;

View for nurse would be:
View for statistic collector would be:

Patient Id	Age	Condition	Date	Place	Ward no	Observer
A101	45	Dengue	10-08-2017	RJV,MUM	B-201	Gautam
A201	52	Malaria	09-05-2017	RJV,MUM	C-263	Vipul
B107	47	Jaundice	09-05-2017	RJV,MUM	B-213	Arsh
C224	29	Measles	05-05-2017	RJV,MUM	A-413	B.C. Bell

Table 3.2: Medical Data for Nurse

Age	Condition	Place
45	Dengue	RJV,MUM
52	Malaria	RJV,MUM
47	Jaundice	RJV,MUM
29	Measles	RJV,MUM

Table 3.3: Medical Data for Statistic Analyzer

3.2 Limitations of using Views

Use of views does not guarantee that data access is restricted. let us understand it with the help of an example. Let us take a generalised tablea as follows:

A	B	C	D	E	F	G
-	-	-	-	-	-	-

Table 3.4: Generalized Table

- Lets consider that view of column C is given to a user X and view of column D is given to user Y.
- Let us consider column C and D together form more sensitive data than individually C or D taken.
- Let user X give his own access to another user Z. Also user Y has given access of his data to user Z.
- Now Z can combine the data of C and D taken from users X and Y separately to derive the higher level data which is undesirable.

So we can see that the main threat of using views is that it may lead to data leak indirectly by combining data from two separate views by a third person. In order to prevent it we propose implementing RWFM model in databases.

3.3 RWFM: Readers Writers Flow Model

RWFM (Readers Writers Flow Model) model can overcome the drawbacks of the previous implementations . Following are the basic points of RWFM model :

- Each role(user) is classified as a **subject**.
- Each unit of data is classified as an **object**.
- Each subject or object is defined by a triplet (owner,permissible readers, permissible writers or influencers).
- Access rules :
  1. A subject is granted access rules on an object o iff
     $s \in R(o) \wedge R(o) \supseteq R(s) \wedge W(o) \subseteq W(s)$
  2. A subject s can inuence, that is, write to an object o if and only if
     $s \in W(o) \wedge R(s) \supseteq R(o) \wedge W(s) \subseteq W(o)$
- Each object created by a subject has RWFM labels equal to that of the subject.
- Labels on the objects can also be upgraded and downgraded

3.4 Addressing the limitations of views

In chapter 3.1 we saw how using views in database leads to data leak. Here we will take the same example to show how data leak by combining from two different sources can be prevented by implementing RWFM model. We had the following table: Column C was

A	**B**	**C**	**D**	**E**	**F**	**G**
-	-	-	-	-	-	-

Table 3.5: Generalized Table

access by user X and column D was accessed by user Y. User Z had access to data of both user X and Y which he combined to derive data of column C and D (a higher level data). The owner of the database would not have desired to give data to user Z but he

gets access to it. It can be prevented using RWFM model. So using RWFM each of the users will have their reader-writer sets. As user X gives access to its data to user Z, Z will be present in the reader-set of X. Similarly Z will be present in the reader-set of user Y. Now as the owner of the table does not want to share data of column C and D to user Z so in the reader-sets of C and D user Z will not be present. Now when user X tries to access column C he cannot get access to it according to RWFM access rule. User X has user Z in his reader-set while column C does not have user Z in its reader-set so reader-set of the subject X is not a subset of the reader-set of object C. Thus access is denied. As user X cannot get access to column C it cannot share the same to user Z, So Z cannot derive higher level data by combining Column C and D. This is how RWFM prevents leakage of data which cannot be done by implementing views in databases.

3.5 Integration of RWFM in PostGreSQL

- Each cell in a database is assigned a label (RWFM triplet).
- Read / write on each cell is done based on the labels.
- To implement RWFM model following changes are made in PostGreSQL source code :

    1. A table named main_table is created that has 4 attributes - user_id, o_attr, r_attr, w_attr. User _id attribute stores each of the subjects and rest 3 attributes stores the corresponding RWFM triplet for the subject. It means r_attr stores the reader set for subjects and w_attr stores the writer set.
    2. Change in **CREATE** Statement
    3. Change in **INSERT** statement
    4. Change in **SELECT** statement
    5. Change in **UPDATE** statement

3.5.1 Change in `CREATE` query

For each of the columns defined for a table in the create statement corresponding reader and writer sets are added in the following manner :

1. Each column defined in create statement is parsed. Changes are made in **gram.y** file CreateStmt portion. Location of gram.y is **src/backend/parser/gram.y**.

2. For each attribute say attr_name new column nodes are created in gram.y named o_attr _name, r_attr_name and w_attr_name for owner, reader set and writer set. All of the new columns data type are text.

3. While each column is parsed its name is added in a linked list. The linked list is traversed in CreateStmt in gram.y and 3 new column nodes for each column are created and added to the **OptTableElementList**.

Algorithm:

Algorithm 1 CREATE statement algorithm

Inputs: Q : query

Ensure: CREATE table

```
CREATE STATEMENT(Q)
Parse table element list of CreateStmt rule
for each column definition c in table element list do
    if c starts with "o_" or "r_" or "w_" then
        throw error
        break
    else
        Create column o_append c of type text
        Create column r_append c of type text
        Create column w_append c of type text
        Add the new 3 columns to OptTableElementList
    end if
end for
```

3.5.2 Change in `INSERT` query

data added to the table using insert statement is done in the following manner :

1. For each column say col_name mentioned in the insert statement list corresponding o_col _name, r_col_name and w_col _name are added in the same list.

2. The o_attr, r_attr and w_attr values from the main_table of the subject that is inserting in the table are taken.

3. o_, r_attr and w_attr values retrieved are given to the corresponding o_col_name, r_col_name and w_col_name in the insert statement.

4. Thus the tuple inserted by the subject has the same level initially as that of the subject itself.

Algorithm :

Algorithm 2 INSERT statement algorithm

Inputs: Q : query

Ensure: INSERT data into table

```
1: INSERT STATEMENT(Q)
2: Parse target list of InsertStmt rule
3: for each column definition c in target list do
4:    if c starts with "o_" or "r_" or "w_"  then
5:       throw error
6:       break
7:    else
8:       Add column o_append c to target list
9:       Add column r_append c target list
10:      Add column w_append c target
11:      Select o_attr, r_attr and w_attr from main_table of current role
12:      Add values to newly added columns sequentially
13:   end if
14: end for
```

3.5.3 Change in `SELECT` query

Selection of tuples from the table for reading them is restricted to a subject in the following manner :

1. Each of the column added in the select statement by the subject is parsed in gram.y and added to a linked list.

2. In the SelectStmt portion of gram.y we make changes in **from_clause**, **Opt_targetelement_list** and **where_clause** list. In from_clause main_table is added along with the table from which data is to be read.

3. In the opt_targetelement_list clause for each column col_name corresponding R_col_name and w_col_name are added. Also r_attr and w_attr columns from Main_table are also added .

4. In the where clause for each column say col_name corresponding r_col_name is added and is checked whether it contains the subject currently accessing in it or not using LIKE operator (r_col_name LIKE current_user). Also join Condition for the two tables (main_table and the table in the select statement) is added which is (user_id = current_user);

5. Now each tuple is received in the **ExecutePlan** method of **execMain.c**. Each tuple is received one by one. In each tuple we have the values of each column mentioned in the select statement along with their corresponding reader and writer set. Also the o_attr and w_attr value of the subject accessing is also present. Now the read rules as mentioned in RWFM model is checked. If the Tuple values follow the rules then the tuple is printed else not.

Algorithm:

Algorithm 3 SELECT statement algorithm

Inputs: Q : query

Ensure: desired data

```
1: SELECT STATEMENT(Q)
2: Parse table element list of SelectStmt rule
3: Add main_table to from_clause
4: for each column definition c in Select element list do
5:    if c starts with o_ or r_ or w_ then
6:       throw error
7:       break
8:    else
9:       Add column r_ append c to opt_target_list
10:      Add column w_ append c to opt_target_list
11:      Add condition r_ append c LIKE "current_user" to where_clause
12:   end if
13: end for
14: Add column r_attr of main_table to opt_target_list
15: Add column w_attr of main_ table to opt_target_list
16: Add condition o_attr LIKE "current_user" to where_clause
17: Flag=0
18: for each column definition c in Select element list do
19:   if r_attr ⊆ r_ append c and w_ append_ c ⊆ w_attr then
20:      continue
21:   else
22:      Flag=1
23:      break
24:   end if
25: end for
26: if Flag == 0 then
27:   print all columns
28: else
29:   Do nothing
30:   Flag=0
31: end if
```

3.5.4 Change in `UPDATE` query

Implementation of update statement is same as select Statement except that the writer set for each column is checked in where clause to Check whether the current subject is present in the writer set of the data to be Updated or not. Corresponding reader and writer sets for each attribute can then be checked to see whether the update rules are being followed or not according to RWFM model.

Algorithm :

Algorithm 4 Update statement algorithm

Inputs: Q : query

Ensure: Update data

```
UPDATE STATEMENT(Q)
Parse table element list of UpdateStmt rule
Add main_table to from_clause
for each column definition c in UpdateStmt list do
    if c starts with o_ or r_ or w_ then
        throw error
        break
    else
        Add condition w_ append c LIKE "current_user" to where_clause
    end if
end for
Add column r_attr of main-table
Add column w_attr of main_ table
Add condition o_attr LIKE 'current_user' to where_clause
Flag=0
for each column definition c in Select element list do
    if r_ attr ⊇ r_ append c and w_ append c ⊇ w_attr then
        continue
    else
        Flag=1
        break
    end if
end for
if Flag == 0 then
    print all columns
else
    Do nothing
    Flag=0
end if
```

3.5.5 Upgrade/downgrade of labels

The levels for each column say column_name as r_column_name and w_column_name can be updated according to the RWFM rules mentioned below. All the conditions are checked in the where clause list.

Downgrade Rule

If a subject s with label (s1,R1,W1) requests to downgrade object o with label (s2,R2,W2) to label (s3,R3,W3)

Algorithm 5 Downgrade Level

Inputs: Q : query

Ensure: change in label

```
1: Downgrade Level(Q)
2: if (s1 = s2 = s3) and (s1 ϵ R2 )and( W1 = W2 = W3) and (R2 = R1)and(R3 ⊇ R2)
   then
3:    if (W1 = s) or (R3 - R2 ⊆ W2) then
4:    λ(o) = (s3,R3,W3) then
5:       Allow
6:    else
7:       deny
8:    end if
9: else
10:   deny
11: end if
```

Upgrade Rule

If a subject s with label (s1,R1,W1) requests to upgrade object o with label (s2,R2,W2) to label (s3,R3,W3)

Algorithm 6 Upgrade Level

Inputs: Q : query

Ensure: Upgrade or not

1: Upgrade Level(Q)
2: **if** s1 = s2 = s3 and s1 ϵ R2 and W1 $\subseteq$ W2 and W3 = W1 $\cup$ s and R3 $\subseteq$ R1 $\subseteq$ R2 **then**
3: λ (o) = (s3;R3;W3)
4: Allow
5: **else**
6: deny
7: **end if**

New rules have been defined for upgrade / downgrade of levels. The rules are then converted to original rules that comply with PostGreSQL statements. Restriction is that level of only one cell data can be changed at a time.

Downgrade rule :

```
Downgrade <column-name> from <table-name> add <subject-name>
```

This is modified as :

```
Update <table-name> set <column-name> = <column-name> ||
<subject-name>  where <column-name> = (select o_attr from
main_table where user_id = current_user ) and w_<column-name>
= (select w_attr from main_table where user_id = current_user )
and w_<column-name> like '<subject-name>';
```

Upgrade rule :

```
Upgrade <column-name> from <table-name> remove <subject-name> ;
```

This is modified as :

```
Update table1 set r_<column-name> = regexp_replace( r_<column-name>,
<subject-name> ,'') where o_<column-name> = (select o_attr from main_table
```

```
where user_id = current_user) and w_<column-name> = (select w_attr from
main_table where user_id = current_user  ) and r_<column-name>
like '<subject-name>';
```

3.6 Discussion

After understanding the concept of RWFM we realize that the model is efficient regarding privacy. Also after understanding its implementation we realize that it can be easily used in PostgreSQL by modifying its source code. Many security can be converted into RWFM model. We will look into one such case in the next chapter. It turns out that RWFM forms a better model than the original one.

Chapter 4

Hippocratic Databases

In present world most of us share out personal information online for various reasons like shopping, ticket booking, communication, etc. Personal information includes name, address, e-mail ID, phone number, etc. The data can be used for various purposes. For example by knowing a type of items person bye online one can understand the likes and dislikes of people of a particular region. Also personal data can be used for research purposes, various statistical analysis, recommendation purposes, etc. These information are very personal to individuals and would like to control the way it is used. An individual may not like his data being used for some particular purpose. Every individual has the right to know how their personal information is being used. So if a particular organization wants to use an individuals data for some purpose it must be authenticated by the individual first. This leads us to the concept of purpose based privacy control which we will discuss subsequently. There are a special kind of databases called hippocratic databases that deal with purpose based privacy.

4.1 Introduction to Hippocratic Databases

In general purpose based privacy refers to giving access to a person by the owner of data based on the purpose defined with which the person wants to get access to the data. The concept is based on Hippocratic databases. So as the name suggests Hippocratic Databases are the databases where each data owner has control on their own data. The owner can specify whom to give access to data based on a table called privacy policy table.

So a user's access to a data is determined by the privacy tables. Also each data owner specifies for what purpose a user can access the data. Let's understand it in details with the help of an example.

Let us consider the following database :

Table name	**Attributes**
Customer	Customer-Id, Name, Shipping Address, e-mail, Credit-card info
Order	Customer-Id , Transaction-id, product info, Status

Let this database be maintained by a company that sells products. In doing so the company stores data of the users such as the name, address, email and credit-card information. Now the company can use the data collected for various purposes. Let us look into some of them.

- Company can give e-mail address to third party company for promotion of their products via mailing.
- Using shipping address company can make a statistical analysis of the products being sold at a particular region. In one place a particular product may be sold more than some other place. These data would be very useful in maximizing the profit by giving relevant product details to each region to new users.
- Giving name a customer to some other company is also possible. We know already that name of a person is a very sensitive information. It defines what a particular person is buying online. Many customers would not like to disclose their identity.

These are few examples of how data can be used. Now some customers may like their data to be shared to others while some may not. So to maintain trust with the customers the company must ensure that they use the data of those customers who have given permission to use their data.

A company has many division as follows:

- **Customer-Service** : Their job is to help customers in need. For that they need the name, e-mail and address of the customers.

- **Shipping** : They have the job to deliver the product to the customer. For that they need at least the delivery address of the customer. The Shipping department can be any third party , other than the company, as well.
- **Charge** : They have the job to ensure that the product payment has been done by the customer or not. For that they need the credit-card info of the person.
- **Purchase-Circle** : They make a statistical analysis of the selling of products. They find regions where a particular product is popular and so give recommendation of that product to new users in that particular region.

Suppose their is a customer Bob who does not want any data to be retained by the company once his purchase his over. Also let us consider another customer Alice who is happy with the fact that the company can trace her purchase history to refer new and relevant products to her. But she does not want to share her information for purchase circles. So in case of Alice we can see that she is willing to share data to the company but for specific **purposes**. As already said in order to ensure trust of Alice the company has to follow her will. Here is where the notion of Purpose Based Privacy Control comes into picture. For the current example let us define some purposes :

- **PURCHASE**
- **REGISTRATION**
- **RECOMMENDATION**
- **PURCHASE-CIRCLES**

From the list above we can easily say that Alice wants to share data for purpose 'PURCHASE' & 'RECOMMENDATION' but not for 'PURCHASE-CIRCLE'.

In Hippocratic Databases, there are two database tables defined to achieve the desired privacy control.

Table name	Attributes
Privacy-policies	Customer-Id,Purpose, table, Attribute, external-recipients, retention
Privacy-authorization	Customer-Id,Purpose , table, attribute, authorized-users

Privacy-Policies table stores the list of external users who can get a particular data of a particular customer. The list is stored in the triplet :(table,attribute,purpose). Thus for each table, for each attribute in that table and for each purpose to use that attribute the customer defines who can get access to that data from outside the company.

An example of Privacy policy table :

Purpose	Table	Attribute	External-Recipient	Retention
Purchase	Customer	Name	deliver-company, credit-card company	6 months

Privacy-Authorization table stores the list of users(we can say them as departments or subjects) within the company who can get access to a particular data of a particular customer. This list is again stored in the triplet :(table-name, attribute, purpose).

An example of Privacy policy table :

Purpose	Table	Attribute	External-Recipient	Retention
Purchase	Customer	Name	deliver-company, credit-card company	6 months

Here we will consider only the privacy authorization table. Let us consider that the privacy authorization table belongs to user Alice. From the table we can see that Alice has given access to Shipping Address attribute only to subject Shipping for purpose Purchase. So Alice can get shipping address data by stating the purpose Purchase only and not for any other purpose. Also we can see that olap cannot access to shipping address data for purpose-circle. Thus it ensures that data of Alice is not shared for statistical analysis which complies with her will. In this way a privacy authorization table is defined.

Let us consider the following table :

Customer (name,shipping-address,e-mail,credit-card info)

Name	Shipping-address	e-mail	Credit-card info
Alex	Washington	alex@gmail.com	7895-4578-4512-7896
Jim	Texas	jim@hotmail.com	8974-6524-1452-3254
Olly	Boston	olly@ymail.com	7896-2541-3256-2541
Brian	New York	brain@gmail.com	2145-9874-6589-4785

Here the data-owners are Alex, Jim, Olly and Brian who have added their data into the database Customer. Each of the data owners will have their own privacy authorization table that defines the access to different attributes of its data given to different subjects (users) for different purposes. Let us define a privacy authorization table for a data owner (Alex in this case)

Purpose	Attribute	Authorized users
purchase	name	shipping,charge,customer-service
purchase	shipping-address	shipping,customer-service
purchase	e-mail	customer-service
purchase	credit-card info	charge
registration	name	register,customer-service
registration	shipping-address	
registration	e-mail	register,customer-service
purchase-circle	shipping-address	olap

Suppose user shipping wants to get access to name attribute in table customer.
Query :

```
Select name from customer for purpose purchase.
```

First in the privacy authorization table of Alex it is checked whether Alex has given access to shipping for purpose purchase. As he has given access (from the privacy authorization table it is clear), shipping will get access to the attribute name.

Name
Alex

Now suppose olap wants to get access to email attribute for purpose purchase-circle

```
Query:    Select email from customer for purpose purchase-circle;
```

In privacy authorization table Alex has not given access to attribute email for purpose purchase-circle. So he will not get access to the data.
In this way privacy is controlled using the privacy authorization table.

4.2 Policy Enforcement in Hippocratic Databases

A policy is a rule to give access (read/write) to a database element of a table to a person (subject in this case). Read access can be given to a subject by giving access to select statements in SQL. Write access can be given to a subject by giving access to insert,update and delete statements. Syntax of a policy is as follows:

```
Create restriction <policy-name>
On        <table-name>
For user  <subject-name>
To cell   <attribute-name>
For purpose <purpose-name>
Restricting access to <access-type>
```

Access-type can be either select (for read), or insert-update (for write) operations. Here we will consider only read operations, so access type will be only select.

Let us consider the table **Customer** used before and a new privacy authorization table of subject(data-owner) Brian.

Purpose	**Attribute**	**Authorized users**
purchase	name	shipping
purchase	shipping-address	shipping
purchase	e-mail	shipping,customer-service
purchase	credit-card info	charge
registration	name	register,customer-service
registration	e-mail	register,customer-service

Let us apply the following policy:

```
Create restriction r1 on customer
for user customer-service
to cells name
for purpose purchase
restricting access to select
```

Privacy authorization table after implementation of the policy:

Purpose	**Attribute**	**Authorized users**
purchase	name	shipping,customer-service
purchase	shipping-address	shipping
purchase	e-mail	shipping,customer-service
purchase	credit-card info	charge
registration	name	register,customer-service
registration	e-mail	register,customer-service

So we see here that after implementing the policy customer-service is added in the Authorized users column where purpose is purchase and Attribute is name.

There can be situation where a data-owner has 2 policies and needs to apply one of them. So it may be needed to determine whether the 2 policies are contradictory or not. To determine whether the 2 policies are contradictory we model the Hippocratic database into RWFM model.

4.2.1 Converting a policy into labels

So in RWFM model each of the data will have reader-writer label in them. As we consider only select operations here, so we need to keep only the reader sets of objects and subjects.

Name	**R_Name**	**Shipping-address**	**R_Shipping-address**	**e-mail**	**R_e-mail**
-	-	-	-	-	-

As data is accessed in the table based on purpose so let us see whether keeping purpose in the reader set is consistent with the original model or not.

Let us consider the following privacy authorization table:

Purpose	**Attribute**	**Authorized users**
purchase	name	shipping,charge,customer-service
purchase	shipping-address	shipping
purchase	e-mail	shipping,customer-service
purchase	credit-card info	charge
registration	name	register,customer-service
registration	e-mail	register,customer-service
purchase-circle	shipping-address	olap

Reader-set for attributes:

- **Name:** purchase,registration
- **Shipping-address:** purchase,registration,purchase-circle
- **E-mail:** purchase, registration
- **Credit-card info:** purchase

Reader set for subjects:

- **Shipping:** purchase
- **Charge:** purchase
- **Customer-service :** purchase,registration
- **Olap:** purchase-circle

According to RWFM model subject **charge** will get access to all attributes, which is not desirable according to the privacy authorization table. So purpose alone cannot form the reader-set, we need to specify which users are given the authority to access the data along

with the purpose.

Reader-set for attributes:

- **Name:** {purchase:shipping,charge,customer-service}, {registration:register,customer-service}
- **Shipping-address:** {purchase:shipping},{registration:register},{purchase-circle:olap}
- **E-mail:** {purchase:shipping,customer-service},{ registration:register,customer-service}
- **Credit-card info:** {purchase:charge}

subjects list would be:

- **Shipping-purchase**
- **Charge-purchase**
- **Customer-service-purchase**
- **Customer-service-registration**
- **Olap-purchase_circle**

4.2.2 Proof of consistency

In the original model for each subject we were checking the purpose for which the subject is accessing an attribute and whether he is authorized to get access or not. Here in the reader set both the purpose and authorization are clubbed together so we can check directly whether a particular user has access to the data or not. Subjects will also have the purposes clubbed with the subject name, example customer-service-purchase where customer-service is subject name and purchase is purpose. The reason for adding the purpose is that there can be many purposes for which access may not be granted. For example in the reader-set of attributes mentioned above subject Olap cannot access any attribute for purpose purchase. So new subject created as olap-purchase. So now it is taken into consideration according to the RWFM access rules that olap-purchase (if defined

as a subject) cannot access any attribute. So in the new model which is a conversion of the purpose-based model based on privacy authorization table to RWFM model all access are consistent.

4.3 Implementation Details

Changes are made in the source code of PostgreSQL in order to implement the RWFM model in database. 1 additional database table is required that for the implementation of the model. The structure of the table is:

- **main_table(user_id type(text), reader-set type(text))**

main_table contains the reader-set of each of the subjects that has access to the database. The reader-set set of subjects are added to the read labels of the attributes in a table when the subject gets access to that particular table.

Structure of main_table :

user_id	**reader-set**
-	-

The implementation of RWFM model should be such that users of the database should be unaware of the fact that the model is deployed. So subjects should not know that the reader-set of each of the attributes are also present in the database and should not be able to access them.

So CREATE statement of PostgreSQL must be modified such that for each attribute defined an additional attribute is added to the database table that contains the reader-set for that attribute. Here we define a standard way of defining the new attributes created. Suppose the name of the attribute defined by the user be **X** then reader-set of the corresponding attribute is **R_X**.

On inserting a row into a table by a subject S, the reader-set of S is also added to each of the reader-set attribute labels with purpose OWNER. So the initial labels of each of the data elements in a table are always the reader-set of the data-owners with purpose OWNER. After that policies are applied to extend the read-set of the attributes to give access to the data to other users.

SELECT statement is also modified to ensure access to the data is according to the RWFM rules. For each of the attributes defined in the select statement corresponding reader-set of the attributes is also checked. For each row being accessed from a table, the reader-set of the attributes defined in the SELECT statement are checked for whether the reader-set of the subject accessing the attributes is a subset to the reader-set of the attributes. If any if the condition fails for a row the row is not accessed by the subject.

Let us consider the following data is present in **main_table** :

user_id	**reader-set**
Shipping	Shipping
Purchase-circle	Purchase-circle
Customer-Service	Customer-service
Register	Register
Bob	Bob
Olap	olap

Step 1 : to create a table as follows:

```
Create table customer(name text,shipping_address text,
email text,credit_card text)
```

As source code of PostgreSQl is modified to add the reader-set column of each attribute, so the new table structure would be as:

(name , r_name,shipping_address,r_shipping_address, email, r_email, credit_card, r_credit_card)

Step 2: to create subjects(roles in terms of PostgreSQL):
(for data owners)

```
Create role shipping-purchase;
Create role customer_service-purchase;
```

Same would be for other purposes like registration,purchase-circle,etc. and for other subjects like register and olap with their own set of purposes

(for subjects who want to access data)

```
Create role jim;
Create role alex;
```

Step 3: to give access to insert and select data on table customer to data owners.

```
Grant insert on customer to alex;
Grant insert on customer to jim;
Grant select on customer to alex;
Grant select on customer to jim;
```

Also data-owners have to be given access to main_table and subject_details to enter their reader-set and add their attribute-list and purpose-list respectively.

```
Grant select on main_table to alex;
Grant select on main_table to jim;
Grant select on subject_details to alex;
Grant select on subject_details to jim;
```

Step 4: inserting data into table by data-owners:

```
Set role alex;
Insert into customer (name, shipping_address, email, credit_card
values(alex, washington,alex@gmail.com,9874-5698-7895-1254

Set role jim;
Insert into customer (name, shipping_address, email, credit_card)
values (jim,texas,jim@ymail.com,9814-5758-7843-1874);
```

If we use the above insert statements then only the data of the actual columns will only get inserted, not the reader-sets of the columns. According to RWFM model we need the reader-sets of the columns for access and initially the reader-sets must be the reader-sets of the data owners. So the insert SQL statement must be modified as below.
Change of the sql code:

```
Insert into customer (name, r_name, shipping_address,
r_shipping_address , email, r_email, credit_card, r_credit_card)
values(jim,( select reader-set from main_table where
user_id = current_user), texas, (select reader-set from main_
table where user_id = current_user), jim@ymail.com, (select
reader-set from main_table where user_id = current_user),
9814-5758-7843-1874, (select reader-set from main_table where
user_id = current_user));
```

Step 5: Selecting(or accessing) data by data-owners:

```
Set role jim;
Select name,shipping_address from customer;
```

Current situation of the table

name	R_name	shipping-address	R_shipping-address	email	r_email	credit-card	R_credit-card
alex	{own:alex}	washington	{own:alex}	alex@gmail.com	{own:alex}	9874-5698-7895-1254	{own:alex}
jim	{own:jim}	texas	{own:jim}	jim@ymail.com	{own:jim}	9474-5148-1795-6554	{own:jim}

Table 4.1: Table customer

Step 6: Creating policies : Let us apply the following policies on table customer.

```
Create restriction r1 on customer for user shipping to
cells name for purpose purchase restricting access to select;

Create restriction r2 on customer for user register to cells
shipping_address for purpose registration restricting access
to select;
```

After execution of the policies the table would look like as follows:

name	R_name	shipping-address	R_shipping-address	email	r_email	credit-card	R_credit-card
alex -	{own:alex} {purchase:shipping}	washington -	{own:alex}- {registration:register}	alex@gmail.com -	{own:alex} -	9874-5698-7895-1254 -	{own:alex} -
jim -	{own:jim} {purchase:shipping}	texas -	{own:jim}- {registration:register}	jim@ymail.com -	{own:jim} -	9474-5148-1795-6554 -	{own:jim} -

Table 4.2: Table customer

So now **shipping** can access **name** attribute and **register** can access **shipping** attribute

Algorithm 7 CREATE statement algorithm

Inputs: Q : query

Ensure: CREATE table

1: CREATE STATEMENT(Q)
2: Syntax : Create table <table_name> <attribute_list>
3: New_attribute_list = <attribute_list>
4: **for** each attribute att in < attribute_list > **do**
5: Create new attribute p_att of type int /* p_att is set to 1 if data owner wants does not want to share data with anyone else 0 */
6: Create new attribute r_att of type int /* r_att stores reader-set who can access the data */
7: Append p_att and r_att to new_attribute_list
8: **end for**
9: Apply new statement to database
10: Create table <table_name> new_attribute_list>

Algorithm 8 INSERT statement algorithm

Inputs: Q : query

Ensure: INSERT data into table

1: INSERT STATEMENT(Q)
2: Syntax : Insert into <table_name> <attribute_list> Values <values_of_attribute_list>
3: new_attribute_list = NULL
4: New_value_of_attribute_list = NULL
5: **for** each attribute att in < attribute_list > **do**
6: Add att in new_attribute_list
7: Add p_att_in new_attribute_list
8: Add r_att in new_attribute_list
9: **end for**
10: **for** each value V1 in value_of_attribute_list **do**
11: Add V1 in new_value_of_attribute_list
12: Add 0 in new_value_of_attribute_list
13: Add Select reader_set from main_table where user_id = current_user in new_value_of_attribute_list
14: **end for**
15: Apply new statement to database
16: Insert into <table_name> <new_attribute_list> values <new_values_of_attribute_list>

Algorithm 9 SELECT statement algorithm

Inputs: Q : query

Ensure: SELECT data from table

1: STATEMENT STATEMENT(Q)
2: Syntax : Select <attribute_list> from <table> where <condition>
3: New_statement = Select <attribute_list> from <table> where condition and
4: **for** each attribute att in <attribute_list> **do**
5: New_statement = new_statement + p_att $\neq$ 1
6: New_statement = new_statement + r_att like select reader_set from main_table where user_id = current_user
7: **end for**
8: Apply new_statement to database

Policy structure :

Let us consider we want to apply a policy for user Alex.

Create restriction <**restriction-name**>

On <**table-name**>

For user <**subject-name**>

To cell <**attribute-name**>

For purpose <**purpose-name**>

Restricting access to <**access-type**> ;

String s=Policy

Array Arr = words of string S

Restriction-name=Arr[2]

Table-name=Arr[4]

Subject-name=Arr[7]

Attribute-name=Arr[10]

Purpose-name=Arr[13]

Access-type=Arr[17]

- Update <Table-name> set r_<Attribute-name> = r_<Attribute-name> || < Purpose-

name > : <subject-name> where name = 'Alex';

By understanding the algorithms we realize that implementing RWFM in hippocratic databases is easy and effective. It does not change the normal functioning of hippocratic databases and enhances the privacy. It also prevents data leaks which is extremely essential. We understood that privacy is controlled by use of policies in Hippocratic databases. There may be some situations where we need to compare a particular set of policies with another set of policies. In the next section we will discuss of it in details. We will try to understand it with the help of examples and finally describe an algorithm to do the same.

4.4 Detecting Policy Contradictions

There can be a policy P1 and another policy P2 that can be applied to a database table. We want to find out whether P1 is same as P2 or different. If they are different then there is a policy contradiction. To determine so we need to have two identical tables with same set of data and same reader-set for each of the data in the table initially. Let the two tables be table1 and table2. Now apply P1 on table1 and P2 on table2 to check whether the policies are contradictory or not.

let the data in table1 and table2 initially be as follows:

name	**R_name**	**shipping-address**	**R_shipping-address**	**email**	**r_email**	**credit-card**	**R_credit-card**
alex -	{own:alex }-	washington -	{own:alex }-	alex@ gmail.com	{own:alex }-	9874-5698 -7895-1254	{own:alex }-
jim -	{own:jim }-	texas -	{own:jim }-	jim@ ymail.com	{own:jim? }	9474-5148 -1795-6554	{own:jim }-
alice -	{own:alice} -	sydney -	{own:alice} -	alice@ hotmail.com	{own:alice} -	8745-1254 -2563	{own:alice} -
bob -	{own:bob} -	melbourne -	{own:bob} -	bob@ gmail.com	{own:bob} -	7894-5612 -7894	{own:bob} -

Table 4.3: Table1 and table2

Policy on table1 by data owner Alex:

```
Create restriction p1 on table1 for user register to
cells name for purpose registration restricting access to select ;
```

Policy on table2:

name	R_name	shipping-address	R_shipping-address	email	r_email	credit-card	R_credit-card
alex -	{own:alex} -{registration:register}	washington -	{own:alex }-	alex@ gmail.com	{own:alex }-	9874-5698 -7895-1254	{own:alex }-
jim -	{own:jim}- .	texas -	{own:jim }-	jim@ ymail.com	{own:jim }-	9474-5148 -1795-6554	{own:jim }-
alice -	{own:alice}- .	sydney -	{own:alice} -	alice@ hotmail.com	{own:alice} -	8745-1254 -2563	{own:alice} -
bob -	{own:bob}- .	melbourne -	{own:bob} -	bob@ gmail.com	{own:bob} -	7894-5612 -7894	{own:bob} -

Table 4.4: Table1

```
Create restriction r2 on table2 for user register to cells
shipping-address for purpose registration restricting access to select ;
```

table1 and table2 after applying policy P1 and P2:

name	R_name	shipping-address	R_shipping-address	email	r_email	credit-card	R_credit-card
alex -	{own:alex} .	washington -	{own:alex }-{registration:register}	alex@ gmail.com	{own:alex }	9874-5698 -7895-1254	{own:alex }-
jim -	{own:jim}- .	texas -	{own:jim -	jim@ ymail.com	{own:jim }-	9474-5148 -1795-6554	{own:jim }-
alice -	{own:alice}- .	sydney -	own:alice -	alice@ hotmail.com	own:alice -	8745-1254 -2563	{own:alice} -
bob -	{own:bob}- .	melbourne -	{own:bob} -	bob@ gmail.com	{own:bob} -	7894-5612 -7894	{own:bob} -

Table 4.5: Table2

In policy one register is getting access to name in table1 and in second policy register is getting access to shipping-address in table2. Clearly the two policies are different. It is to be noted here that we can say that the two policies are contradictory only when the subjects on which the policies are applied are same.

Implementation Implementation starts with creating a same copy of the table on which the two set of policies P1 and P2 are applied. Let the two tables be Table1 and Table2. Now the policies P1 and P2 are applied on Table1 and Table2 respectively. After that we check the access of each of the subjects on each of the the attributes of the two tables. If the output of each of the results in both the tables is same then the policies are same else contradictory. Let the subject on which P1 is applied be S1 and subject on which P2 is applied be S2.

Algorithm 10 Detecting policy contradiction

Inputs: $P1$: Policy 1

$P2$: Policy 2

Ensure: Policies are contradictory or not

```
1: Apply P1 to table1
2: Apply P2 to table2
3: Flag_contradictory = 0
4: for each attribute Ai do
5:    if S1 = S2 then
6:       Set role S1
7:       OP1 = Select Ai from table1
8:       OP2 = Select Ai from table2
9:       if (OP1 ≠ OP2) then
10:         Flag_contradictory =1
11:      end if
12:   end if
13: end for
14: if (Flag_contradictory == 1) then
15:    Print policies are contradictory
16: else
17:    Print Policies are equal
18: end if
```

4.5 Discussion

So we can summarize that RWFM is a good alternative to normal purpose-based privacy method in Hippocratic databases. Also it can be used to find out whether two set of policies are contradictory or not. In the next chapter we will get into details about a case study of Conference Management System, HotCRP.

Chapter 5

HotCRP: A case study

HotCRP is a widely used conference management system. It is used as a paper reviewing system and uses security policies to control information flow within the system. Authors add their papers and Reviewers review the papers and give score to them. The authors then get to know about the scores given to their papers by the reviewers without knowing the individual scores given by the reviewers. The whole process is controlled by an administrator known as Program Chair. After scores for all the papers are finalized Program Chair has then declassifies some of the top paper reviews to the author of the paper. Even after using the security policies HotCRP has is vulnerable to information leakage. The Program Chair uses Discretionary Access Control to control flow which is why it is vulnerable. The subjects and objects in HotCRP are predefined and so we can use RWFM model to control information leakage and make the system more secure.

The different set of subjects present in HotCRP are as follows:

- A central administrator known as Program Chair, PC that controls the overall system. It declassifies information to other subjects as and when required.
- Set of Authors Auth who enter details about their papers to be reviewed.
- A set of Reviewers Rev who review the papers entered by Authors.

The main objects in the system are as follows :

- Papers P

- Reviews R

Main security policies in the system are as follows :

1. Authors cannot review papers i.e., Auth $\cap$ Rev = ϕ
2. Program Chair cannot review any paper i.e, PC $\cap$ Rev = ϕ
3. A paper P when added can be viewed by all reviewers and also by the Program-Chair.
4. Each paper also has an Assigned Set (AS). An assigned set is also a mapping from a paper P to a set of Reviewers Rev. P -> $2^{|\text{Rev}|}$ Only the reviewers present in the AS are allowed to review the paper. So C $\cap$ AS = ϕ
5. A reviewer is allowed to access reviews of other reviewers for a particular paper only if the reviewer has submitted his own review.

Let us understand the work flow of HotCRP with the help of an example

- Let the program chair be C.
- Let us consider two authors as A1 and A2. Let there be 2 reviewers as RR1 and RR2. So the total subjects in the system are
  {A1,A2,RR1,RR2,C}
- Let there be 2 papers P1 and P2 of authors A1 and A2 respectively.
- Let R11 be the review of reviewer RR1 for paper P1 and R12 be the review of reviewer RR2 of paper P1.
- Hence the set of objects in the system are { P1,P2,R11,R12 }

Now we need to define RWFM labels for the objects and the subjects.

- For Paper P1 Author A1 is the author, hence A1 is the owner of P1. According to the HotCRP rule initially a paper can be read by the author, program chair and all the reviewers. So the reader set would be { A1,RR1,RR2,C }. As author A1 has

written it so the writer set would be A1 itself. The RWFM labels for the paper P1 would be

({A1}, {A1,RR1,RR2,C}, {A1})

- Similarly for P2 the RWFM labels would be: **({A2}, {A2,RR1,RR2,C}, {A2})**
- In HotCRP a review is submitted to the the Program Chair . Initially the reviewer is the owner of a review. As only the reviewer and program chair has access to the review the reader-set for a review would be the reviewer and program-chair. For the review author of the paper influences the review hence the author has to be in the writer set along with the reviewer. So RWFM labels for R11 (review of paper P1 by reviewer RR1):

  (RR1, {C,RR1}, {A1,RR1})

  RWFM labels for R12 (review of paper P1 by reviewer RR2):

  (RR2, {C,RR2}, {A1,RR2})

- For program chair, initially it can read and write only its data. hence the initial label would be

  (C, {C}, {C})

After a review is submitted by a reviewer the Program-Chair(C) accesses it using the RWFM rules. After accessing the review the Program-chair stores the review on a table. Let the table be Review-Decision. structure of the table is:

According to the RWFM rules the label for the review stores in the new table Review-

Paper	**Author**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
-	-	-	-	-	-	-

Table 5.1: Table Review-Decision

decision by program-chair would be:

(C, {C,RR1} ∩ {C }, {A1,RR1} ∪ { C}) which equals to

(C, {C} , {A1, RR1, C})

Similarly for review R12 label in Review-Decision would be:

(C, {C} , {A1, RR2, C})

The decision column decides the final label of the reviews. Suppose review R11 comes first

followed by R12. The label of decision would be least upper bound of label of R11 and R12 which is **(C, {C} , {A1, RR1,RR2, C})** .The label of decision column determines who can get access to the reviews finally. In declassification process a review is given access to other reviewers and the author. Declassification of an object to a subject is possible only when the subject is present in the writer-set of the object. Also there is a condition in HotCRP that a reviewer cannot get access to other reviews unless he submits his own review. The decision column of the paper-review table controls that. Whenever a reviewer submits his review the label of the reviewer and the current label of Decision and joined (as least upper bound) that includes the new reviewer in the writer set of Decision column. This ensures that the current reviewer has submitted his review and so now can get access to other reviews for the paper after the program-chair declassifies paper to him. Here after R11 is submitted it has to be checked whether RR2 has submitted his review. If only RR2 has submitted the review then he can access the the review of RR1 that is R11. Similarly it would be for reviewer RR1 to access R12. RR1 can access R12 only if he has submitted his own review that is R11. Hence the declassification process is restricted by the above conditions. This is ensured by the Decision column of the paper review table. It is to be noted that a reviewer enters into the writer set of Decision column only after his review is submitted.

Let us understand it with the example. First at time T1, R11 submits the review for paper P1. So the Review-Decision table would be:

Paper	**Auth**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1	A1	R11	({C}, {C},	-	-	({C}, {C},
.	.	.	{A1,RR1,C})	-	-	{A1,RR1,C})

Table 5.2: Review-Decision - after submitting review R11

The program chair cannot declassify the review R11 to RR2 as RR2 is not present in the writer-set of decision. This satisfies the condition in HotCRP that a reviewer cannot see other review unless he submits his own review. So next RR2 submits the review R12 for paper P1. So now the review decision table would be:

Now the program-chair executes the declassification command to declassify R11 to RR2. So from the Review-Submit table we find that RR2 has submitted the review for paper P1 so declassification process can be done. The declassify command would be:

Paper	**Auth**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C} {A1, RR1, RR2,C})

Table 5.3: Review-Decision - after submitting review R12

```
Declassify to RR1 for paper P1 ;
Declassify to RR2 for paper P1 ;
Declassify to A1 for paper P1 ;
```

Updated table Review-Decision would be:

Paper	**Author**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C,RR1,RR2} {A1, RR1, RR2,C})

Table 5.4: Review-Decision - after declassifying R11 and R12

Now after the declassification process RR2 gets access to the review R11, RR1 gets access to review R12 and author A1 gets access to reviews R11 and R12. Let P2 be now submitted by author A2. Label for P2 would be **(A2, {A2,RR1,RR2,C} , {A2})** . Then RR1 and RR2 submit their reviews as R21 and R22. Decision label will perform lub operation with the new labels. Similar to R11 label of R21 would be **(RR1, {C,RR1} , {A2,RR1})** and for R22 it would be **(RR2, {C,RR2} , {A2,RR2})** . In decision column least upper bound operation will be performed between the current label of decision and label of new review being submitted. After submitting R21 and R22 the table would look like as follows.

Paper	**Auth**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C} {A1, RR1, RR2,C})
P2 -	A2 -	R21 -	({C}, {C}, {A2, RR1, C})	R22 -	({C}, {C}, {A2, RR2,C})	({C}, {C} {A1,A2, RR1, RR2,C})

Table 5.5: Review-Decision - after adding reviews R21 and R22

Program-Chair can now declassify the reviews to RR1, RR2 and A2 so that they can get access to the reviews R21 and R22. The Review-Decision table would look like as

follows: Now as each of the papers P3, P4 .... P_n are submitted corresponding decision

Paper	**Auth**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1	A1	R11	({C}, {C},	R12	({C}, {C},	({C}, {C}
-	-	-	{A1, RR1, C})	-	{A1, RR2,C})	{A1, RR1, RR2,C})
P2	A2	R21	({C}, {C},	R22	({C}, {C},	({C}, {C,RR1,RR2,A2}
-	-	-	{A2, RR1, C})	-	{A2, RR2,C})	{A1,A2, RR1, RR2,C})

Table 5.6: Review-Decision - after declassifying R21 and R22

column label is changed using the Least upper bound operation. So after both the reviews are submitted for paper P_n label of decision column is updated as **({C}, {C}, {A1,A2,.... A_n, RR1, RR2,C})**. Declassification is then done to the desired authors and reviewers by the program chair.

Chapter 6

Deployment of secPostgreSQL in system

In this chapter we will explain how to implement RWFM model in PostgreSQL. Prerequisites for it are:

- Operating System: Ubuntu version 14 and above.
- PostgreSQL source code version 9 and above.

6.1 Installing and configuring PostgreSQL source code

To implement RWFM in PostGreSQL we need to modify its source code. After downloading the source code following commands need to be executed:

- Following variables must be set appropriately:

```
export POSTGRES_INSTALLDIR=<path to install dir>
export POSTGRES_SRCDIR=<path to postgre source dir>
```

- Then use command:

```
cd ${POSTGRES_SRCDIR}
```

- Configure for debugging using the following command:

```
./configure --prefix=
${POSTGRES_INSTALLDIR} --enable-debug

export enable_debug=yes
```

- Run the following 2 commands:

```
make | tee make.out

make install | tee make_install.out
```

- Set the following environment variables:

```
export LD_LIBRARY_PATH=${POSTGRES_INSTALLDIR}
/lib:${LD_LIBRARY_PATH}
export PATH=${POSTGRES_INSTALLDIR}/bin:${PATH}
export PGDATA=${POSTGRES_INSTALLDIR}/data
```

- Now a new database cluster needs to be created:

```
initdb -D ${PGDATA}
```

- Postgres Server is run using the following steps:

```
postmaster -D  $PGDATA > logfile 2>&1 &
```

- new database is created and run by the following 2 commands:

```
createdb -p5432 test
psql -p 5432 test
```

6.2 Changing source code of PostgreSQL

Parsing of CREATE, INSERT, SELECT statements are done in the file: **src/backend/-parser/gram.y**. Before making changes a table is created named **main_table** that stores the owner, reader and writer sets for each of the subjects in the database. The main_table has the columns as follows:

- user_id: stores the name of subject.
- owner: stores the owner label for the subject.
- reader-set: stores the reader-set of the subject.
- writer-set: stores the writer-set of the subject.

Now changes are made in the SQL statements as follows:

- For CREATE statement go to label **createstmt** that defines the rule for create statement query. From there parse through the following rules **OptTableElementList -> TableElementList -> TableElement -> columnDef -> ColId**. ColId defines the name for each column in the table. For each of the column name we get in ColID label create three different nodes one for owner, one for reader-set and another for writer-set for the column. The data type of the columns would be Text. Now add them into the table in the **createstmt** rule.

- For INSERT statement go to file: **src/backed/tcop/postgres.c**. In the file go to **PostgresMain** method variable query_string stores the SQL insert statement. Format of insert statement is:

```
Insert into <table-name> (column-names)
values (values-for-the columns) ;
```

Now (column-names) is parsed in the variable **query_string** and for each column-name c, o_c, r_c and w_c are added into (column-names). Corresponding to it in the (values-for-the-columns) the owner, reader and writer sets are added of the subject who is entering the data into the database. So select statements are added into the (value-for-the-columns) as:

```
Select owner from main_table where user_id = current_user ;
Select reader-set from main_table where user_id = current_user ;
Select writer-set from main_table where user_id = current_user ;
```

example of a modified insert statement:

```
insert into <table-name> (A,R_A,W_A) values
(<value-A>, select reader-set from main_table
where user_id = current_user, select writer-set
from main_table where user_id = current_user) ;
```

- For Select Statement we again do the changes in **src/backend/parser/gram.y**. Go to the label **simple_select** in **gram.y** and parse through all the columns defined in the select query in the label **OptTableElementList**. For each of the column c parsed add the reader-set and writer-set label of c in the column list. In the where condition add condition for each of the column c as **reader-set-c like current_user**. Now go to location: **src/backend/executor/execMain.c** where the

reader-set and writer set of all the columns in the select statement are visible. Now use RWFM read rules with the reader-writer sets of each column and the reader-writer set of the subject. If the conditions satisfy then display the record else do not show.

After each change made in the source code we need to rebuild the source code using the following statements:

```
make | tee make.out
```

```
make install | tee make_install.out
```

6.3 Implementing HotCRP

For implementing HotCrp we need 3 tables as follows:

- **main_table** to store the reader-writer sets of each object that is the papers and reviews.
- **review_decision** table that stores the reviews of each paper along with their labels and the decision label. The detailed list of columns of review_table has been mentioned previously.
- **paper_details** that stores the paper_id, author and description of each paper submitted by an author.

Access to each of the tables by the subjects are as follows:

- Program-Chair C has all access (insert, update, select, etc.) on table review_decision and main_table and paper_details. Corresponding SQl statement is:

```
Grant all on <table-name> to <program-chair> ;
```

- Each of the authors can perform only insert and select operations on the paper_details table. Corresponding SQl statements are:

```
Grant select on paper_details to <author> ;
Grant insert on paper_details to <reviewer> ;
```

- Each of the reviewers can perform only select operation in paper_details. Corresponding SQl statement is:

```
Grant select on paper_details to <reviewer> ;
```

Following are the steps of execution:

1. After a paper s submitted in paper_details table, Program-chair checks it using Select statement on paper_details. Then he adds the paper_id and author name in the review_decision table. SQL statements are:

```
Select paper_id,author from paper_details ;
insert into review_decision (paper_id,author)
values (<paper_id>,<author>) ;
```

2. Reviewers check the paper_details table to get details of papers. Then they submit their review.

```
Select paper_id,author from paper_details ;
Submit review <review-name> for paper <paper_id> ;
```

   The second statement is added to a php code connected to PostgreSQL. The input to the PHP code are 2 parameters, one the reviewer name and second the review submit command. When a review is submitted the php code is executed to find the lub (least upper bound) of the review with the RWFM label of Program-Chair. The RWFM label of program-chair is present in main_table. Now the lub currently calculated is added to the RWFM label of the review in review_decision. Simultaneously the

lub of the label of the review and the present RWFM label of decision column is calculated and updated in the decision column label the the PHP code.

3. Declassification is done only by the program-chair. To declassify the reviews for a particular paper the writer-set of the decision column is first accessed and then matched whether the subject to whom the review needs to be declassified is present in the writer-set or not. If present then the reader-set is updated by adding the subject-name in it. The declassification process is also done in a PHP code connected to PostgreSQL. The input to the PHP code is the declassification command:

```
declassify reviews to <subject-name> for
paper <paper_id> ;
```

The PHP code extracts the subject-name and paper_id and check the conditions as mentioned above. If conditions satisfy declassification is done else not. For declassification to the authors the PHP code checks whether the author is same as the author of the paper to be declassified. The check is done in the review decision table as:

```
select author from review_decision for
paper_id = <paper_id>;
```

If the author name from resulting from the select statement matches with the author name to whom the paper needs to be declassified then the paper is declassified to the author else not.

6.4 Demonstration of HotCRP

1. First Program-Chair C inserts paper_id and author in review_decision table.

```
insert into review_decision (paper_id,author)
values('p1','a1') ;
```

Paper	Author	RR1	l-RR1	RR2	l-RR2	Decision
P1	A1	.	.	.	.	.

Table 6.1: Review-Decision

2. Reviewer RR1 enters the review R11 using the statement:

```
submit review r11 for paper p1 ;
```

Paper	Author	RR1	l-RR1	RR2	l-RR2	Decision
P1	A1	R11	({C},{C},	.	.	({C},{C},
.	.	.	{C,RR1,A1})	.	.	{C,RR1,A1})

Table 6.2: Review-Decision

3. RR1 tries to access the review using the statement

```
set role rr1;
select rr1 from review_decision where paper_id = 'p1' ;
```

He does not get access to it as rr1 is not present in the reader-set of decision column label.

4. Then reviewer RR2 enters the review R12 using the statement:

```
submit review r12 for paper p1 ;
```

Paper	Auth	RR1	l-RR1	RR2	l-RR2	Decision
P1	A1	R11	({C}, {C},	R12	({C}, {C},	({C}, {C}
-	-	-	{A1, RR1, C})	-	{A1, RR2,C})	{A1, RR1, RR2,C})

Table 6.3: Review-Decision - after submitting review R12

5. Program-Chair C declassifies the reviews R12 and R11 to RR1 using the statement :

```
declassify reviews to rr1 for paper p1 ;
```

6. Now RR1 tries to access the reviews using the statement:

Paper	**Author**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C,RR1} {A1, RR1, RR2,C})

Table 6.4: Review-Decision - after declassifying R11 and R12

```
set role rr1;
select paper_id,rr1,rr2 from
review_decision where paper_id = 'p1' ;
```

Paper_id	**RR1**	**RR2**
P1	R11	R12

Table 6.5: Review-Decision

7. Still RR2 and author A1 cannot access the reviews as they are not in the reader-set of decision column label. So now the reviews are declassified for them as:

```
declassify reviews to rr2 for paper 'p1' ;
declassify reviews to a1 for paper 'p1'  ;
```

Updated review_table shown in table 6.6

Paper	**Author**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C,A1,RR1,RR2} {A1, RR1, RR2,C})

Table 6.6: Review-Decision

8. Now A1 accesses the reviews using the SQL query:

```
set role a1;
select rr1,rr2 from review_decision where paper_id = 'p1' ;
```

result will be same as RR1.

9. Now paper p2 is inserted into review-decision:

```
insert into review_decision (paper_id,author)
values('p2','a2') ;
```

Updated review_table shown in table 6.7

Paper	**Author**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C,A1,RR1,RR2} {A1, RR1, RR2,C})
P2	A2	.	.	.	.	.

Table 6.7: Review-Decision

10. Reviews R21 and R22 for paper p2 by reviewers RR1 and RR2 are submitted:

```
submit review r21 for paper p2 ;
submit review r22 for paper p2 ;
```

Updated review_table shown in table 6.8

Paper	**Auth**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C,A1,RR1,RR2} {A1, RR1, RR2,C})
P2 -	A2 -	R21 -	({C}, {C}, {A2, RR1, C})	R22 -	({C}, {C}, {A2, RR2,C})	({C}, {C} {A1,A2, RR1, RR2,C})

Table 6.8: Review-Decision - after adding reviews R21 and R22

11. Now reviews for P2 can be declassified to RR1:

```
declassify reviews to rr2 for paper p2 ;
```

Updated review_table shown in table 6.9

Paper	**Auth**	**RR1**	**l-RR1**	**RR2**	**l-RR2**	**Decision**
P1 -	A1 -	R11 -	({C}, {C}, {A1, RR1, C})	R12 -	({C}, {C}, {A1, RR2,C})	({C}, {C,A1,RR1,RR2} {A1, RR1, RR2,C})
P2 -	A2 -	R21 -	({C}, {C}, {A2, RR1, C})	R22 -	({C}, {C}, {A2, RR2,C})	({C}, {C,A2,RR1,RR2} {A1,A2, RR1, RR2,C})

Table 6.9: Review-Decision - after declassifying reviews R21 and R22

12. P2 is declassified to RR1 and not to RR2 or A2. So if RR2 or A2 tries to access the reviews for paper P2 they will not get. So the statement for RR2:

```
Select paper_id,author from review_decision ;
```

    RR2 will not get access to the details.

13. Now Program-Chair declassifies the data to RR2 and A2 as:

```
declassify reviews to rr2 for paper p2 ;
declassify reviews to a2 for paper p2 ;
```

    Now they can see the reviews R21 and R22.

14. Author A2 selects the reviews for his paper P2 as:

```
set role a2 ;
select paper_id,rr1,rr2 from review_decision
where paper_id = 'p2' ;
```

    Updated review_table shown in table 6.10

Paper_id	**RR1**	**RR2**
P2	R21	R22

Table 6.10: Review-Decision

The database can be extended to add more reviewers and authors. Corresponding to it the reader-writer sets of the reviewers and authors have to be added to the main_table.

Chapter 7

Conclusions and Future Work

In this report we focused on 3 things:

- Detecting whether 2 policies are contradictory in Hippocratic Databases.
- Comparison of classical databases and database implementing RWFM.
- Modelling HotCRP as RWFM.

We found out that Purpose based privacy control model has some privacy issues that can be rectified by converting it into RWFM model and that the conversion process is consistent with the original model. We also developed an algorithm to find out whether 2 set of policies acted upon a database are consistent with each other or not. We also found out that Modelling HotCRP with RWFM is consistent with the original model. It can be used to prevent data leaks in the system. Here we provide things that has to be done in future for implementing better security in multilevel database.

- Current Implementation deals with the basic SQL queries like CREATE, INSERT, DELETE, UPDATE, SELECT and VIEW. There are other functionality in SQL as transaction, concurrency, etc. where security needs to be implemented.
- Also functionality to handle triggers must be included as triggers play a vital role in database management.

Bibliography

[1] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, Walid Rjaibi. Extending Relational Database Systems to Automatically Enforce Privacy Policies. *21st International Conference on Data Engineering (ICDE'05)*

[2] Rakesh Agrawal, Ramakrishnan Srikant, Jerry Kiernan, Yirong Xu. Hippocratic Databases *IBM Almaden Research Center*

[3] Dorothy E Denning, Selim G. Akl, Mark Heckman, Teresa F. Lunt, Metthew Morgenstern, Peter G. Neumann and Roger R. Schell. Views for Multilevel Database Security. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-13, NO. 2, FEBRUARY 1987*

[4] David Schultz, Barbara Liskov IFDB: Decentralized Information Flow Control for Databases, *Proceeding EuroSys '13, Proceedings of the 8th ACM European Conference on Computer Systems, Pages 43-56*

[5] N.V.Narendra Kumar and R.K.Shyamasundar . Realizing Purpose-Based Privacy Policies Succinctly via Information-Flow Labels. *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on 3-5 Dec, 2014*

[6] Denning, Dorothy E, A lattice model of secure information flow, *Communications of the ACM, 19(5):236-243, 1976.*

[7] Bertino, Elisa and Sandhu, Ravi, Database security-concepts, approaches, and challenges, *IEEE, 2(1):2-19, 2005.*

[8] PostgreSQL https://www.postgresql.org/ *Accessed June 2018*

[9] Biba, K.J., Integrity Considerations for Secure Computer Systems, *MTR3153, The Mitre Corporation, June 1975.*

[10] HotCRP: Conference Management System https://hotcrp.com/ *Accessed June 2018*

Acknowledgements

I would sincerely thank my guide **Prof. R. K. Shyamasundar** for providing me valuable time and guide throughout this project. I would also sincerely like to thank my senior, **Pratiksha Chaudhary**, who has worked in this project and provided me valuable information.

Signature: .....

Aniket Kuiri

163050059

Date: ..... June 2018