M.Tech Thesis

SecPostgreSQL: A System for Flow-Secure View, Transaction, Sanitization and Declassification on MLS Database

Submitted in partial fulfillment of
the requirements for the award of the degree of

Master of Technology
in
Computer Science and Engineering

Submitted by
153050038
Pratiksha Chaudhary

Under the guidance of
Prof. R.K. Shyamasundar

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Approval Sheet

This thesis/dissertation/report entitled SeeInstgreSQL : A system for flow Secure View, Transaction, Sanitization and declassification on MLS database by Pratiksha Chaudhary is approved for the degree of M. Tech.

Examiners

PROF. ASHWIN GUMASTE
CSE DEPT, IIT BOMBAY

PROF. VIRENDRA SINGH
EE DEPT, IIT BOMBAY

Supervisor (s)

PROF. R.K. SHYAMASUNDAR
CSE DEPT, IIT BOMBAY

Chairman

VIRENDRA SINGH
EE DEPT, IIT BOMBAY

Date : 26-06-2017

Place : Mumbai

Declaration

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.

Pratiksha

(Signature)

Pratiksha Chaudhary

(Name of the student)

153050038

(Roll No.)

Date: 30-06-2017

Acknowledgments

I would sincerely like to thank my guide **Prof. R.K. Shyamasundar** for providing his valuable time and guidance throughout this project. I would also like to thank **N.V. Narendra Kumar** for his indispensable continual advice, which helped me immensely throughout this project.

Abstract

Security and integrity have been a rising concern in today's technology savvy world as storing and processing electronic data are in high demand. The classical *PostgreSQL* has various vulnerabilities that can easily be exploited by attackers. *SQLInjection* attacks have been on top-10 attacks of *OWASP* consistently since past decade. These attacks exploit the various vulnerabilities of a database. The Discretionary Access Control model used in a database system is insufficient to prevent such breaches as they allow us to apply security at the granularity level of a relation or an attribute. Hence, a need for much more secure database system arises which is highly secure as well as scalable to accommodate a large number of users. In this dissertation, we provide a design and implementation of *SecPosgreSQL* which extends the existing system, classical *PostgreSQL* for flow-secure multilevel database. *SecPostgreSQL* on top of all the features available in the classical *PostgreSQL* has incorporated a decentralized information flow control for protecting data from privacy breach. The semantics of complex SQL queries like, views, transactions, etc. has been transformed to incorporate information flow rules. *SecPostgreSQL* integrates a novel purpose-based policy model called Readers Writers Flow Model [5], in order to realize information flow control in databases. One of the important characteristic features of our model is the robust sanitization and declassification which is defined carefully to inhibit information flow leakage.

SecPostgreSQL looks database applications from two different views: one having single subject performing all SQL operations on the database on behalf of other users, which is addressed by tuple level labeling approach; while the other having a proper set of subjects and objects, which is addressed by Readers Writers Flow Model. The performance overhead has been compared with classical *PostgreSQL* for all basic SQL operations and the results obtained are satisfactory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional database systems are used to store data efficiently and fetch them when required. They are optimized for storage, fast retrieval, and access. The primary concern they address is the performance of the database system. Many organizations use databases for various applications. Some of them are storing user credentials, which may be encrypted, storing a medical history of a patient in a medical database, storing purchase history and billing information in a supermarket database and so on. These database systems do what they are supposed to do in an optimized and efficient manner. But all of them lack one common thing: security. Database systems do not incorporate stringent security checks when the data is being accessed. More specifically, the checks can be done only at the table level, but not at the finer level such as an attribute, or even a tuple. Traditional database systems have this shortcoming. To overcome this drawback Multi-Level Secure database [19], more generally MLS database can be used. MLS databases go beyond the traditional DAC policies adopted by the classical database systems. They enforce information flow control in the system at such a finer granularity such as an attribute or a tuple. Such enforcement is nearly impossible or very difficult to incorporate. The reason being that they were designed for efficiency, and not for information flow control. Hence a need arises for an MLS-enabled secure database system.

SecPostgreSQL, an MLS database system, built upon the classical *PostgreSQL*, overcomes such shortcomings of the traditional database system. It enforces the security checks at the tuple level and controls the information flow control in the system using a novel decentralized information flow control model RWFM. We will discuss more about RWFM in the later sections. Effectiveness of *SecPostgreSQL* has been demonstrated using various case studies and examples. Performance evaluation of *SecPostgreSQL* in comparison to classical *PostgreSQL* reveals that there is only a performance overhead

of approximately 15% for various queries such as *Select, Insert, Delete, Update* etc. The overhead has been measured using the benchmarking tool TPC-H [15].

The related work is done in the similar field, Decentralized Information Flow for Databases [2], provides support for Multi-Level Secure databases by assigning security classes to all the subjects and objects in the system and restricting all the invalid information flows in the system. Information flow has been tracked at two levels: database level and application level. Database accepts connection from only those applications which also implements information flow control. The paper has transformed application logic of Python and PHP to integrate information flow rules. Two popular applications, *CarTel* [17] and *HotCRP* [16], has been modified to incorporate the changes and performance have been evaluated using TPC-H benchmark. The semantics of all basic SQL operations, such as SELECT, INSERT, DELETE, UPDATE has been modified and new SQL constructs are also introduced to implement concepts like *declassification* and *sanitization*. The drawback with this approach is that the declassification rule is not so robust as it can make information available to anyone. Also, in the case of IFDB, an application also has to work in a trusted environment which keeps track of information flow.

The above-mentioned shortcomings have been addressed by our approach, in which *PostgreSQL* handles applications using two different security policies, depending on the role of subjects available in the system. If a system has subjects that operate on behalf of other users, which implies that the system does not have a proper set of subjects defined, then **Tuple-Level Labelling** approach is used and if a set of subjects is defined in the system, then **Readers Writers Flow Model** (RWFM) [5] is used. Both of them labels data but at two different levels. Tuple level assigns labels to entire tuples; while RWFM assigns labels to each of the data cells. Both of them have their own advantages and disadvantages.

This report is arranged in the following fashion. Chapter 2 gives the reader an insight into the required background knowledge for better understanding of this dissertation. It describes the various security models including RWFM and describes the working of *PostgreSQL* in detail. Chapter 3 provides the implementation details as to how *SecPostgreSQL* has been implemented. Modification done to the classical *PostgreSQL* for each query as well as the syntax and semantics of the newly introduced queries are described in this section. It also answers the *why* of this dissertation through general theoretical flaws in the existing database systems and compelling examples. Chapter 4 describes the RWFM based approach for securing information flow control in *PostgreSQL* using *HotCRP* as an example. Chapter

5 gives the performance analysis of *SecPostgreSQL* in comparison to *PostgreSQL* and species the measured performance overhead in various scenarios. Chapter 6 and Chapter 7 describe the conclusion of this dissertation and future directions.

Chapter 2

Background and Related Work

This chapter aims to give the reader a brief background required to understand the purpose, goal, and achievement of this thesis. First, we introduce the reader to various multilevel security models, followed by a detailed description of *PostgreSQL*, its architecture, and its relevant applications.

2.1 Classical Security Models

There are a plethora of security models available, but the discussion will be limited to a main few which need to be understood to understand various nuances of this thesis. A few of them are:

2.1.1 Bell-LaPadula Model

Bell-LaPadula model [9] was developed in the 1970s for securing information in the Military domain. It restricts the information flow in the system. This is achieved by assigning levels to various subjects and objects. Subjects, i.e., the users have clearance level whereas objects, i.e. mainly files, have security classifications. Now, based on these clearance levels and classification, accesses are allowed or denied. BLP model follows state transition to denote the security of a system at any point of time. A system makes a transition through several states when subjects perform operations such as read, write, etc. For each transition t, it is secure only if it follows the following three properties at any given point of time:

- **Simple security condition:** This property enforces the "no read up" constraint of BLP. It states that a subject which is at a lower clearance level than that of a subject cannot read an object which has a higher classification level than the subject. Thus a subject can read an object

only if the following constraint is satisfied: $L(s) \geq L(o)$ Where $L(s)$ is the clearance level of the subject s whereas $L(o)$ is the classification of an object o. The subject s can read that object o only if it has a higher clearance level than the object.

- **Star property:** This property enforces the "no write down" constraint. It states that a subject cannot write to an object which has classification level lower than the subject's clearance level. In other words, a subject can write only to objects which are at a level at least as that of the subject. $L(s) \leq L(o)$

- **Discretionary Access Control:** This property states that every access should also be constrained by the discretionary access control policy of the system. Just because the access is allowed by the BLP doesn't mean that the access should be granted if it violates the DAC policy of the system and vice-versa.

2.1.2 Biba Integrity Model

Biba-integrity model [9], in contrast to Bell-LaPadula model, deals with the integrity of the system, rather than its confidentiality. This is very useful in scenarios where the integrity of the data, i.e., preventing data corruption, is much more important than preventing leakage of data. Same as in BLP model, subjects have clearance level and objects have classification level. Based on their levels, subjects are granted access to objects. Biba also follows state transition framework for maintaining system integrity. Each state at any point of time should satisfy all the following properties:

1. **Simple Security Property:** A subject s cannot write to an object if level of the object dominates the level of the subject. That is: $L(s) \geq L(o)$, should be satisfied so that a subject can write to an object.

2. **The Star property:** A subject s cannot read an object o if the level of the subject dominates the level of the object. That is: $L(o) \geq L(s)$, should be satisfied so that a subject can read an object.

2.1.3 Denning's Information Flow Model

Denning et al in [1], proposes an information control model using lattice-based approach. It guarantees information flow control in the system using axioms that the system must follow. It defines a set o security classes SC, and the permissible flow controls using the *can-flow* operator $\rightarrow$. A class

combining operator $\oplus$ defines the Least Upper Bound (LUB) of two security classes. Let N be the set of all objects in a system. The objects can be file or variables or other information containers. And let, P be the set of processes in the system. These processes are active entities which cause the information flow from one object to another, or to another process. Together, these constitute an Information Flow Model, formally denoted as a five tuple,

$$<\text{N, P, SC, } \rightarrow, \oplus>$$

whose definition of each symbol was described above.

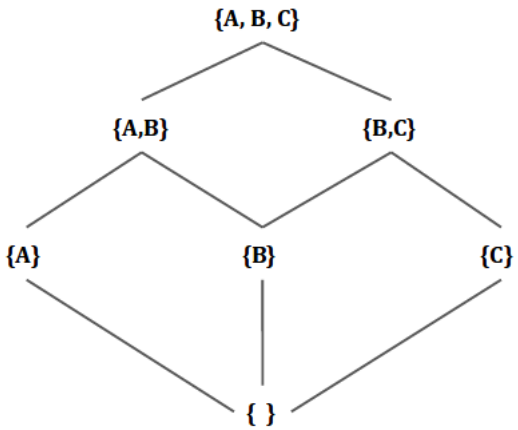

Figure 2.1: Lattice Hierarchy in Denning's Model. *A, B, C* are security classes

Information flow control in a system can be controlled by specifying what a subject at a particular level in the lattice can read or can not read. Each level in the lattice denotes a Security Class. A security level A is said to dominate a security level B, i.e., A$\geq$B, iff, B$\rightarrow$A, meaning information can flow from B to A.

In each of the above-described classical security model, the definition of a level in a lattice is not clearly and formally defined. Hence, RWFM model has been used for implementation of *SecPostgreSQL*. We will now discuss about the RWFM model in the next section.

2.1.4 The SeaView Security Model

This model is used for databases that contain data belonging to different classes of sensitivities and not all the users have the highest clearance to access the entire dataset. It follows lattice model with the nodes containing access classes, which further contains security level (for instance, TOP

SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED etc.), and categories. The security levels follow totally ordered relation; while categories follow partially ordered relation.

The SeaView Security Model follows Mandatory Access Control model, which prevents information leakage from covert channels. Within the lattice, no information can flow from a higher level of security to lower level and also, information can only be written to lower levels. The concept of *reference monitor* is introduced, which actually guards all access to the objects by subjects. All the access requests for objects pass through the reference monitor, which checks whether it conforms to the information flow rules or not. All the security-critical operations must pass through reference monitor.

The design approach of SeaView model implements multilevel relation using views at different security levels. Subjects interact with the relation using views, which fetches data from a single level relation stored physically in a file. This transition from single level to multi-level relation is transparent to users. Reference monitor lies just above the single level relation. Since for all the requests data is ultimately fetched from single level relation, which passes through reference monitor in order to perform label comparison to determine if access should be granted or not.

2.2 Models For Database Security

Several works have been done in this field. Here we discuss a few prominent works briefly. Comparison of their work with *SecPostgreSQL* is also discussed in the following sections.

2.2.1 Information Flow in Databases

[2] The classical *PostgreSQL* does not keep track of information propagation which might lead to information leakages and even if the data is protected by access control, it can be inferred by performing multiple SQL queries on dataset. GRANT and REVOKE statement controls access of data objects; but once a user gets access to information, it can always pass on the information to other users, which can not be prevented in the existing system.

The following are the main challenges that IFDB [2] addresses:

- IFDB uses *Query by Label* model, which modifies the semantics of basic SQL queries so that they conform to information flow rules.

- It also introduces new SQL constructs for *declassifying views* and *stored authority closures*.
- For preventing transactions and constraints from revealing information, concepts such as *transaction commit labels* and *declassifying classes* are introduced.

Information flow has been tracked at two levels: database level and application level. Database accepts connection from only those applications which also implement information flow control. The paper has transformed application logic of Python and PHP to integrate information flow rules. Two popular applications, *CarTel* and *HotCRP*, has been modified to incorporate the changes and performance has been evaluated using TPC-H benchmark. The semantics of all basic SQL operations, such as SELECT, INSERT, DELETE, UPDATE has been modified and new SQL constructs are also introduced to implement concepts like *declassification* and *sanitization*.

One of the major drawbacks that IFDB faces is that it does not have a robust declassification and sanitization feature. Any user can declassify data to any other user. This leads to information leakage, if not done carefully. However, in RWFM, only the influencers can be added to the readers set, i.e., data cannot be declassified to any user. Also, application has to work in a trusted environment which keeps track of information flow.

2.2.2 Fine Grained Access Control

The classical *PostgreSQL* follows Discretionary Access Control Model, i.e. user itself defines the access control policies, using GRANT and REVOKE statements. The drawback of this approach is that it does not provide access at a finer level of granularity, i.e. it can only operate at a relation or an attribute level.

The above shortcomings has been addressed by using **predicated GRANT** statements. Unlike the existing GRANT statements, *predicated GRANT* includes WHERE clause as well so as to introduce access control at tuple level or cell level. Also, they have introduced the notion of authorized views so that only the authorized user gets access to that view. The syntax and semantics of *predicated GRANT* has been changed to include qualifications, like other SQL queries. The *predicated GRANTs* also provide authorization on functions, procedures and aggregates. The authorized views can handle complex queries and large number of users as well. All the users share

the same authorized view with a parameter representing user identity. Such views are instantiated every time user tries to access it. Users can define authorization for a user group as well.

The only drawback with the *predicated GRANT* is that the authorized views has to be initiated every time a user tries to access it. This might result in increase in access time of the view.

2.3 Shortcomings of PostgreSQL

The classical *PostgreSQL* does not support Mandatory Access Control model, which implies all security policies enforced are controlled by users herself. In *PostgreSQL*, the following are the ways using which privacy of users can be ensured:

- One of the ways of dealing with privacy of data is by using GRANT and REVOKE statements. These statements follow Discretionary Access Control model as users themselves grants access to a relation or a few of the attributes of a relation to other users. This is one of the most common ways of enforcing security at table or attribute level. The syntax for granting and revoking access to data objects is as follows:

  ```
  GRANT privilege_name ON object_name TO user_name [WITH GRANT
  OPTION];

  REVOKE privilege_name ON object_name FROM user_name;
  ```

- In addition to GRANT and REVOKE statements, Row level Policies deal with preserving privacy of data at tuple level in a relation on a per user basis. Owner of the table defines security policies which will be abided by all users trying to access or modify that relation. This is again following a Discretionary Access Control model where in, a user defines the policy which returns boolean value for a particular user; the only difference being user can do so for only those relations that are owned by her. After defining the policy, if a user tries to access the relation, then prior to performing any check on qualifications provided in query, *PostgreSQL* first checks whether that tuple is visible to the user or not.

2.4 RWFM: Readers Writers Flow Model

RWFM [6] model is motivated by Denning's Information flow model. Its novelty is purpose based privacy policy which follows lattice model to restrict invalid information flow within the system. It recasts Denning's information flow model and brings a notion of implementing labels as a triplet $<$*Owner*, *Permissible readers*, *Influencers*$>$. Denning's information flow model assigns security classes to each of the entities in the system and controls information propagation by restricting invalid flows. In the Denning's model, information can not flow from higher security level to a lower level. The existing security models such as Biba Model, Bell La Padula Model etc. have several flaws:

- Labels are not defined properly in any of the security models with an implementation perspective,
- Dynamic labelling has not been incorporated correctly,
- Confidentiality of data takes only readers set into account,
- Declassification rules are not robust, leading to information leak.

RWFM model realizes the purpose-based privacy policies by assigning security labels to all subjects and objects in the system. According to this model, the *purpose of access* can be captured by looking at the state of a subject requesting access to the information. Another important aspect of this model is to control how a subject uses the information accessed, and in particular, to whom the subject discloses the results of processed information.

Definition:
Reader Writer flow model is defined using five tuples:

$$(\mathrm{S},\mathrm{O},2^S \times 2^S,(\cup,\cap),(\supseteq,\subseteq))$$

where;
S denotes a set of subjects.
O denotes a set of objects.
Labelling function is defined as λ: $\mathrm{S} \cup \mathrm{O} \rightarrow \mathrm{S} \times 2^S \times 2^S$
The labelling function assigns labels to all of the entities present in the system which basically contains three components and are described below :
RWFM is denoted using a three tuple set (s,R,W)

- s: owner
- R: permissible readers

- W: subjects who have influenced the information

Initial label of object is $\{s, S, \emptyset\}$ while subjects have the initial label $\{s, s, S\}$. The initial label depicts the state of subjects and objects at the starting state of the system. The initial label of the object is such as it denotes that the object has not yet been influenced by any of the subjects in the system. The subjects label gets modified once it starts gaining information, which will be described in the coming sections (Read rule). The initial label positions the subject at the lowest point in the RWFM lattice, and thus it *climbs* the lattice as it starts gaining information.

Confidentiality and integrity of the entities are expressed using readers set and writer set respectively. Readers set contains subjects that are allowed to read information; whereas writers set contains subjects that have influenced information so far.

2.4.1 Access Rules

- A subject s is granted read access on an object o iff,
  $$\mathrm{s} \in \mathrm{R(o)} \wedge \mathrm{R(o)} \supseteq \mathrm{R(s)} \wedge \mathrm{W(o)} \subseteq \mathrm{W(s)}$$
  This means that the subject should be present in the reader's list of that object, and that the subject is at equal or higher level that the object that it wants to read. Thus guaranteeing information confidentiality.

- A subject s can influence, that is, write to an object o iff,
  $$\mathrm{s} \in \mathrm{W(o)} \wedge \mathrm{R(s)} \supseteq \mathrm{R(o)} \wedge \mathrm{W(s)} \subseteq \mathrm{W(o)}$$
  This means that the object should be allowed to influence the object and that the subject should reside at a level equal or lower than the object.

Lowest point in lattice is {s, S, $\emptyset$ }, while the highest point is {s, $\emptyset$, S }. This is because the reader sets get *intersected* while moving up the lattice, while *union* happens for writer set.

2.4.2 Information Flow Diagram

RWFM uses labelling to denote the state of the system at any point of time. The state of the system can be inferred by the labels of the all the subjects and objects in the system. Subjects are active entities which are responsible for the information flow in the system. They read information from an object and influence/write another object. Object here act as a passive entity. They are passive because they themselves do not cause a state change in the system. The labels of a subject change over a period of time as it goes on gaining

information from various objects in the system, and thus have dynamic label. Objects on the other hand have static label, unless modified by the owner of that object. State change in RWFM happens by various events such as,

- Reading of an object by a subject
- Downgrading of an object
- Creation of an object.

2.4.3 RWFM rules

RWFM follows the Dennings lattice model for information flow control. But the rules are bit different than from Dennings. It is required as the lattice should now incorporate the readers and writers also, and then define the rules based upon them. The lattice points are defined by the set of readers and witers and thus they play a pivotal role in the rules of RWFM.

We will now discuss the various rules defined in RWFM and give a brief description of each.

- **Read rule**

  Read access request to an object o with label (s_1,R_1,W_1) is granted to a subject s with label (s_2,R_2,W_2) iff

  - s is present in readers list of that object, i.e., s $\in R_1$
  - If allowed to read, this means that there has been an information flow, thus the subject s will move up the lattice. The new updated label of the subject will be : (s_2,$R_1 \cup R_2$,$W_1 \cup W_2$)

- **Write rule**

  Write access request to an object o with label (s_2,R_2,W_2) is granted to a subject $s$with label (s_1,R_1,W_1) iff,

  - s should be present in the writers/influencers list of the object, i.e., s $\in W_2$
  - The level of the object in the lattice should be at least as much as that of the subject, meaning that the level of the object should dominate the level of the subject. Thus, $R_1 \supseteq R_2 \wedge W_1 \subseteq W_2$

  Here, like in read, the label of the subject doesn't change as there was no information gained by the subject. Also since the object label is not downgraded or created, hence object's label also do not change.

- **Create rule**

  When a subject s having label (s_1,R_1,W_1) creates a new object, the object label is set same as that of the subject. This denotes that the newly created object has the same level in the lattice as that of the subject and thus is information flow preserving.

- **Downgrade rule**

  If a subject s with label (s_1,R_1,W_1) requests to downgrade object o with label (s_2,R_2,W_2) to label (s_3,R_3,W_3)
  begin
  if ($s_1 = s_2 = s_3$) and ($s_1 \in R_2$)and(W_1= W_2 =W_3) and ($R_2 = R_1$)and($R_3 \supseteq R_2$)then
  if (W_1 = s) or (R_3 - $R_2 \subseteq W_2$) then
  λ(o) = (s_3,R_3,W_3)
  allow
  else
  deny
  else
  deny
  end

  - Only the owner of the object is allowed to downgrade.
  - Subject and object must be at the same level.
  - Only the subjects that have influenced the object before can only be added to the readers set.
  - Ownership and writer set of object remain intact.

- **Upgrade rule**

  If a subject s with label (s_1,R_1,W_1) requests to upgrade object o with label (s_2,R_2,W_2) to label (s_3,R_3,W_3)
  begin
  if ($s_1 = s_2 = s_3$) and ($s_1 \in R_2$) and ($W_1 \subseteq W_2$) and ($W_3 = (W_1 \cup$ s)) and ($R_3 \subseteq R_1 \subseteq R_2$)then
  λ(o) = (s_3,R_3;W_3)
  allow
  else
  deny
  end

- Subject must be in the readers set of object to be upgraded.
- Only owner of the object must be allowed to upgrade.
- Readers set of object after upgradation must be a subset of readers set of subject.
- Ownership and writer set of object remain intact.

2.4.4 Advantages of Readers Writers Flow Model

- Security policies are enforced stringently as every data access request always passes through the reference monitor. Even the data after processing is assigned labels, which prevents information leakage indirectly using covert channels.
- Information flow rules take into account the current label of the subject which has been influenced by all the information that has been read by the subject so far.
- Confidentiality of data takes into account both readers set and writers set, unlike other security models.
- Labels are defined properly and are easy to understand. They also provide a clearer view of information flow.

Chapter 3

SecPostgreSQL

In this chapter, first we will discuss about the motivation and the goal of this thesis. Several convincing scenarios are illustrated to convince the reader of the relevance of this thesis. In the further section, details about the implementation of *SecPostgreSQL* is given. Discussion of implementation of both tuple-based approach as well as RWFM-based approach will be done. SQL semantics of both the approaches will be described.

3.1 Motivation and Requirement Gathering

Privacy has become a crucial aspect in all of the modern day applications. In this section, we shall introduce the reader to modern day complex application scenarios and list out privacy requirements from databases of such applications with the help of few scenarios.

Scenario 1 - Medical Data: Consider the case of a medical database in a hospital management system. Such a database contains information about patients, doctors, nurses, medicines, pharmacies, etc. It will contain non-sensitive data such as name, age, etc., and sensitive data such as patient's medical history, their current medications, their contact details, etc. A patient's medical information is sensitive due to many prevalent reasons like `(i)` it determines the patient's insurance premium. `(ii)` a hereditary condition may reveal impending medical conditions of patient's off-springs and in turn their insurance coverage and premiums, for example, it is important that patient's data should be accessible to different roles of employees on need basis. For example, a nurse should not be allowed to access information of a patient with respect to medication etc.; whereas, the doctor may have increased access to her patients and possibly other doctor's patients too

if it helps to improve health of patients in general. Consider spread of an epidemic like swine flu etc. Doctors need more access to understand context and learn from other patient's response to treatments. To enforce these access policies in traditional database systems is not an easy task. One major reason is that they provide only table level or utmost an attribute level security policy. These do not suffice to enforce the required security policies in required manner. Thus hindering the database design process too.

Assume a medical database of *Patients* containing a table of information of all the patients who have visited or consulted the hospital. This information should be made available selectively on a need basis, and should not be made public. Hospitals aim to protect the privacy of patient's information, which they collect for treatment purposes. If a patient, John, visits doctor, Bob, then the information of that patient should be visible only to Bob, hospital management, and the patient. Similarly, Bob should not be usually allowed to view details of patients of other doctors. The nurse, pharmacist, other hospital staff should be allowed only information that they need-to-know about these patients.

Database applications are extensively used by a large number of users these days, which might contain data of different levels of sensitivities. All the users are not supposed to get the same view of data if it contains information belonging to different classes of secrecy. Multilevel secure databases play an important role in such scenarios. However, the existing *PostgreSQL* does not support multilevel databases, i.e., all the users get the same view of data if they execute same query. There is no sensitivity level defined for information in a database. Also, there is no fine grained authorization at the level of data cells. The existing *PostgreSQL* does provide SQL authorization at the level of tables or columns only. The queries such as GRANT/REVOKE give permission to access specified tables/columns for a user. Also, there is no such mechanism to keep track of information flow in the system. All of the information as well as users are treated equally, there are no security levels defined.

The following are a few scenarios where fine grained access control is required, but the current systems do not provide:

- patients must be able to see their own records,
- doctors can see records of their patients,
- students can see their own grades,
- faculty has full access to grades of courses they taught,
- public can see average grades for every course.

There could be several engineering approaches to tackle the above mentioned scenarios. One of the solution is using *Multi-Level Secure* databases, which categorizes all the users according to different levels of sensitivities so that information flow can be controlled in the above scenarios.

In addition to GRANT and REVOKE statements, *Row level Policies* deal with preserving privacy of data at tuple level in a relation on a per user basis. Owner of the table defines security policies which will be abided by all users trying to access or modify that relation. This is again following a Discretionary Access Control model where in a user defines the policy which returns boolean value for a particular user; the only difference being user can do so for only those relation that are owned by it. After defining the policy, if a user now tries to access the relation, then a check is performed to verify if the tuple is available access or modification by that user prior to any user defined condition or function. Usually, relations do not have *row level security* feature enabled. However, it can be enabled by the owner of the table by using the following command:

```
ALTER TABLE table_name ENABLE ROW LEVEL SECURITY;
```

Once the *row level security* is enabled, owner can now define policies on per user basis, which requires an extra attribute for storing the owner of each tuple. The policy definition requires a condition which needs to be satisfied in order to make a tuple visible to a user. The syntax for defining the definition is as follows:

```
CREATE POLICY policy_name ON table_name TO user_role
USING (user_name = current_user);
```

A relation may have such multiple policies defined for different users. Also, a user might make changes in policy using ALTER POLICY command and it may also delete policy using DROP POLICY command.

Scenario 2 - Operating System: A simple example of authentication in Linux is illustrated to depict how row level securities works. Consider a relation named *passwd* which contains all the authentication related information, such as *user_name*, *pwhash*, *uid*, *gid*, etc. The structure of the table is depicted below:

```
CREATE TABLE passwd (
```

```
 user_name   text   UNIQUE NOT NULL,
 pwhash      text,
 uid         int    PRIMARY KEY,
 gid         int    NOT NULL,
 real_name   text   NOT NULL,
 home_phone  text,
 xtra_info   text,
 home_dir    text   NOT NULL,
 shell       text   NOT NULL
);
```

Assuming there are three roles existing in the system: *Admin*, *Bob*, *Alice*. The *passwd* relation is populated with the following tuples:

```
INSERT INTO passwd
VALUES ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');

INSERT INTO passwd
VALUES ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');

INSERT INTO passwd
VALUES ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh
```

Since, relations do not support *row level security* by default, so the following command enables the *row level security* for that particular relation:

```
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;
```

Once the *row level security* is enabled, owner of the table creates policies for different users using the following commands:

- *Admin* can access or modify all the tuples.
  `CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);`
- Normal users, Alice and Bob, can access all the tuples.
  `CREATE POLICY all_view ON passwd FOR SELECT USING (true);`
- Normal users, Alice and Bob, can update their own records.
  `CREATE POLICY user_mod ON passwd FOR UPDATE USING (current_user = user_name) WITH CHECK ( current_user = user_name );`

Now, these security policies will be applicable every time user tries to access or modify this relation.

Statistical databases [18] might also give rise to covert channels. If a statistical query produces result which is contributed by a few tuples only, then a user might infer values of the individual tuples, which is not supposed to happen in case of statistical queries. Statistical queries are not supposed to reveal individual values of the tuples; but, they are only supposed to return aggregate results. The problem has been described in detail using an example illustrated below.

Scenario 3 - Student Information System: Suppose a *Student* relation contains the following attributes: *name, domain, marks* as shown in Table 3.1, in which marks of the students are sensitive and are not supposed to be revealed. This implies users cannot execute usual SELECT queries that fetch marks of the individual students. However, a user can always perform aggregation functions on sensitive attribute, *marks*, since the relation is a part of statistical database. But, performing statistical operations on a column might lead to marks of an individual student getting revealed. The same is illustrated with the help of an example.

Name	**Domain**	**Marks**
Alice	Physics	89
Bob	Biology	74
Cathy	Physics	88
Harry	Chemistry	91
John	Chemistry	90

Table 3.1: *Students* relation

Suppose a user issues a query to fetch the count of all students studying in *physics* department, which is a statistical query. The query is as follows:

```
SELECT COUNT(*) FROM Students WHERE Domain='Physics';
```

If this query returns a very small count as result, then the user might end up inferring the marks of individual students; which is a covert channel revealing the information indirectly. The output of the SELECT query on *Students* relation returns 1.

Since the user now knows that Physics department has only one student, it can further execute query to fetch sum of marks of students studying in Physics department; which would be the marks of an individual student.

The query for fetching sum of marks of all the students studying in *physics* department is as follows:

```
SELECT SUM(Marks) FROM Students WHERE Domain='Physics';
```

The output returned by this query will be 89, which shows that the marks of *Alice* has been revealed. This way a user can infer values of the individual tuples.

In addition to the above problems, existing systems have several other limitations as listed below:

- The standard query model does not provide a good basis for reasoning about flows of sensitive information. The classic *PostgreSQL* does not restrict information to be passed on to other unauthorized users once an authorized user gets access to sensitive data.
- DBMS must provide ways to manage the information flows that arise through mechanisms such as complex queries, stored procedures and views. For instance, executing complex queries such as JOIN, on two or more tables with different security levels, the resultant table should have a level at least as much as the highest level of all tables used for the JOIN operation.
- Transactions and constraints can lead to information leaks. For instance, a user within the same transaction may read tuples from a private database and write in onto a public database.

Scenario 4 - Employee Management System: Consider an employee database containing information of all the employees in an organization. The table contains information such as name, address, designation, salary, contact details etc. Now what will happen if all the employees are given unrestricted access to such a table? The privacy of many individuals will be compromised. One way to prevent this is to bifurcate the table into multiple tables and grant access to them accordingly. but is this solution viable in all the cases? Here since the stakeholders of this single table are restricted to a employees of an organization, and the information is not sensitive, what should be solution in case if the database is much more complex and contains hundreds of tables? Splitting the tables into multiple smaller tables and granting access accordingly will be a much cumbersome task. Thus the need of a much better, cleaner solution arises. Using MLS database is the solution. Assigning users a clearance level and the tuples in the database tables a classification

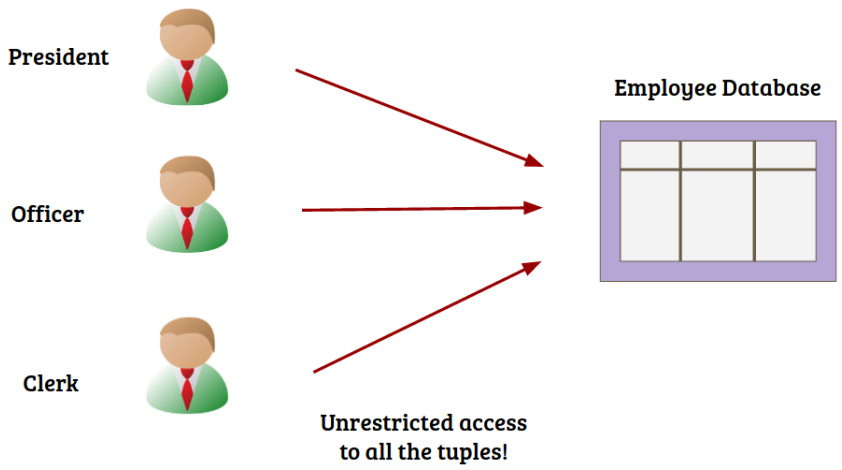

Figure 3.1: Different levels of employees accessing a common database

level, the access to the tuples can be granted depending on the security level. This solution avoids the need of modifying the database design in order to incorporate the various security requirements.

Summary of Requirements:

- In the first scenario of medical data, data is available to all the users without any restrictions. Also, there is no mechanism to declare data as private.
- In the second scenario of authentication in operating system, users themselves create security policies on row level on per user basis, which follows Discretionary Access Control.
- In the third scenario of Statistical queries that are not supposed to reveal individual tuple information, can do so indirectly using covert channels if the result of aggregation function is contributed by a very few tuples.
- In the last scenario, different sensitivity levels of tuples as well as users are not captured properly by the existing system, which leads to access of data objects belonging to a higher sensitive class by a user belonging to lower sensitive class.

3.2 Implementation Details

Various approaches have been implemented in order to secure multilevel databases. Primarily, the approaches vary in terms of security policies used. The reason behind this variation is because of the applications using the database can either have a finite set of subjects in terms of their roles, as in the case of *HotCRP* or can have general users with no specific roles defined. Since, the Readers Writers Flow model demands set of subjects to be constant within the system, so databases like *HotCRP* can be implemented using the same. However, there are some general purpose databases whose subjects are not predefined. Such cases are implemented using tuple based labeling approach. The following sections describes both the approaches in detail.

3.2.1 Utilizing Readers Writers Flow Model

Readers Writers Flow Model is one of the Lattice based models to introduce information flow control in the systems. It supports multilevel security system and has the ability to secure to control information flow in the system. Many security policies and models were proposed as a *Biba integrity model* [20], *Bell La Padula model* [10], *Chinese Wall Security model* [21]. But these models cannot be implemented on different types of system. Some models are also proposed specifically for database system: Sea View security model [4] and Information Flow Control for Databases [2].

3.2.1.1 Labelling the data

Readers Writers Flow Model follows a field based labelling approach, which means that all the cells in the database has a label assigned. It works at a much finer granularity level in terms of labelling. Each data cell acts as a separate object in the database. A separate column has been introduced for each attribute for storing these labels in the relation. A label is a triplet containing owner, set of readers and set of writers. All the subjects and objects in the system are assigned labels to depict their level of sensitivity in the lattice. The subject labels are dynamic in nature while the object labels remain static. Also, all the labels forms a partially ordered set, i.e., there exists some pairs that are not comparable. Since partially ordered sets follows reflexive, transitive and antisymmetric properties, all RWFM labels together forms a lattice.

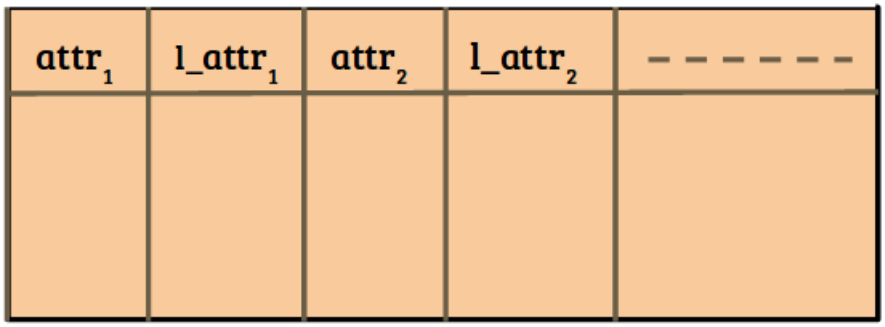

Figure 3.2: Table schema in RWFM based approach

3.2.1.2 Information-flow semantics of SQL Queries

Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements are changed to incorporate the label checking before reading from and writing to data. In DDL, CREATE TABLE statement is modified to support labels per column basis without making user aware of it. In DML, SELECT, INSERT and UPDATE statements are modified so far in order to perform label checking before permitting any information flow. The following sections describes the implementation details of changes done in both DML and DDL statements.

3.2.1.3 CREATE TABLE query

This statement creates a new table in the database with the structure defined by user in the query. When a user executes this query, the *PostgreSQL* creates label columns automatically for each of the user defined columns without letting user aware of this. The addition of label columns is done at parsing level in */backend/parser/gram.y* file. The grammar of CREATE TABLE is modified to support the change. For each *ColumnDefinition*, a *LabelDefinition* is also added, which is of type 'text', for storing the values of labels. The column name for the labels is *l_* followed by the corresponding column name whose label value will be stored in this column. This new label column is then added to the element list maintained by the *CreateStmt*, which is a list of Column definitions. Further processing of this new column appended will be same as that of other user defined columns. The assumption taken here is that all the columns whose names start with *l_* are treated as labels and the *PostgreSQL* will throw an error if any user tries to create a column name starting with *l_*. The table schema is shown in Figure 3.2.

Algorithm 1 CREATE TABLE procedure

```
1: procedure CREATE TABLE(query)
2:     Parse the TableElementList of the CreateStmt rule.
3:     if any column name starts with "l_" then
4:         throw an error
5:     else
6:         for each TableElement do
7:             create a new LabelDefinition called LabelDef
8:             set LabelDef → colname="l_" + ColDef → colname
9:             set LabelDef → typeName="text"
10:            Append LabelDef to TableElementList
11:    return
```

3.2.1.4 Select Query

Select statement is used to retrieve data from table in database. In the output, only those rows must be shown whose readers set contains the user on behalf of whom query is being executed. The check must also be performed on all the columns present in the where clause condition, if it is also a part of query. To perform this label checking, a few changes needed to be done while creating the query tree and also, when the plan is executed, i.e., when the actual values are fetched from the database. Just before creating the query tree, a function called *transformSelectStmt()* in */backend/parser/analyze.c* transforms the *targetlist* as well as the *whereclauselist*. In the same function, before the transformation, parse all the columns in the *targetlist* and add the corresponding label columns to the *targetlist*. Do the same for all the columns in the *wherelistclause*. Now when the plan is being executed (*executePlan()* defined in */backend/executor/execMain.c*) and the tuples are fetched one by one to be shown as output, perform the check on all the label columns in the *targetlist*. If current_user is present in the readers set of all the label columns, send the tuple to the destination to be shown as output, else ignore the tuple.

Algorithm 2 SELECT Procedure

```
1: procedure SELECT(query, user)
2:     Parse the targetlist and whereclauselist
3:     for each column in the targetlist and whereclauselist do
4:         Append the corresponding label column to the targetlist
5:     When the plan is being executed, fetch the tuples one by one.
6:     for each tuple do
7:         if current_user is in the readers set of all target label columns
   then
8:             send tuple to the destination
9:         else
10:            ignore the tuple
11:    return
```

3.2.1.5 Insert Query

Insert statement is used to insert new rows into a table. Insert statement is considered as creating new object in database. So according to create rule of Reader Writer Flow Model, each new inserted data label is updated with current subject's label. If target table has primary key and the key value to be inserted is already in the table, then a check need to be performed to see if the current subject is in the readers set of present label of the key column. If it is not, then the row should be inserted without throwing any *duplicate key value* error as according to RWFM model, this won't be considered as violation of data integrity constraint. Otherwise, *PostgreSQL* must report an error to the user. Label values for the columns values to be inserted are initialized with the label of subject at the parsing level, i.e., in the rule for *InsertStmt* defined in */backend/parser/gram.y*. Now, the *PostgreSQL* looks for the index of primary key, if it exists and scans the B-tree to check if the key value already exists. The check is performed in *_bt_check_unique()* defined in */backend/access/nbtree/nbtinsert.c*. If the value already exists, fetch the whole tuple by de-referencing the record pointer value stored in index. After fetching the row, check if readers set in label of key column contains the current subject. If yes, then throw *duplicate key value exists* error to the user; else, insert the row in the table and update the index as well.

Algorithm 3 INSERT Procedure

```
1: procedure INSERT(query, user)
2:     for each label columns do
3:         set the label values same as the current subject's label
4:     if index for the relation exists then
5:         if key value already exists then
6:             Fetch the tuple using the index
7:             if current subject exists in the readers set in label of key column then
8:                 throw "duplicate value exists" error
9:                 return
10:            Insert the row in the table
11:            Update the index
12:            return
13:    Insert the row in the table
14:    return
```

3.2.1.6 Update Query

Update statement changes the values of specified columns in all rows that satisfy the given condition. Only the columns to be modified need to be mentioned in the SET clause; columns not explicitly modified retain their previous values. According to the RWFM model, the labels of the columns to be updated must have current subject in their writers set and if the query has where clause, then the labels of all the columns on which condition is applied in *whereclause* must have current subject in their readers set.

Before performing the check, labels of all the columns present in *update-targetlist* and *whereclause* list should be added to update *targetlist*. This is performed just before the query tree creation and before transforming *udate-targetlist* and *whereclauselist* in a function called *transformUpdateStmt()* defined in */backend/parser/analyze.c* file. Now when the plan is being executed and the tuples to be updated are being fetched one by one, perform the label

checking. The UPDATE statement's execution is done by a function called *execUtils()* defined in */backend/executor/nodeModifyTable.c* file.

Algorithm 4 UPDATE Procedure

1: **procedure** UPDATE(*query*, *user*)
2: Parse the *updatetargetlist* and *whereclauselist*
3: **for** each column in the *updatetargetlist* and whereclauselist **do**
4: Append the corresponding label column to the updatetargetlist
5: When the plan is being executed, fetch the tuples one by one.
6: **for** each tuple **do**
7: **if** current subject is not present in the readers set of columns present in whereclauselist **then**
8: Do not update the tuple and continue fetching the next tuple.
9: **if** current subject is not present in the writers set of columns present in updatetargetlist **then**
10: Do not update the tuple and continue fetching the next tuple.
11: **return**

3.2.1.7 Transactions

A transactions is a set of multiple SQL queries forming a single logical operation. A transaction either gets executed completely or not executed at all; i.e., it is atomic in nature. Each of the query executed by *PostgreSQL* is a part of transaction. Individual queries create a new session for a transaction and once the execution is done, transaction is committed or aborted. The creation of a new session initializes the subject label with the initial value; i.e., owner is the subject itself, readers set contains all the subjects in the system and writers set contains again the subject itself. In case of multiple statements getting executed as part of a transaction, subject label might climb up in lattice as it is dynamic and can only be changed if subject reads information. Further, if the subject inserts a tuple in a relation, the label of the attributes of the tuple would reflect the updated label of the subject. That is, subject label has to be maintained in some data structure in main memory. It need not be stored in a persistent storage as once the transaction gets committed or aborted, the subject label will not be valid and need to be initialized again for execution of other queries.

When a subject begins a transaction, a flag is set to indicate if the queries

going to be executed are part of transaction or not. If yes, then the subject label is fetched from the data structure and relevant checks are performed using that label. The semantics of the basic SQL queries remain the same as discussed above; the only difference being subject label is fetched from the data structure as the label of the subject might change during the course of the transaction. When user execute commit or abort the flag is again unset and the label of the subject becomes invalid.

3.2.2 Tuple Based Labelling Approach

This approach is adopted in order to incorporate information flow control in systems where subjects are not predefined as in the case of RWFM. This drawbacks of RWFM has been handled by this approach. However, RWFM also plays a role in case of complex queries, stored procedures and transactions . The RWFM labels are a bit different from the one defined in the previous approach. The primary key value of each tuple is treated as a separate subject here in order to generalize it for different systems.

3.2.2.1 Labelling of data

This security policy follows the tuple level labelling approach unlike RWFM which follows cell level labelling. Assigning labels at row level do saves space but the granularity of classification of sensitivity level of data is finer in RWFM as compared to this policy. Also, here the labels are not like RWFM labels which are a triplet containing owner, set of readers and set of writers but is a combination of confidentiality level and integrity level. The confidentiality level defines the highest level a subject can read; while the integrity level defines the lowest level a subject can write. The levels are totally ordered set, i.e., any two of the levels from the universal set of levels are comparable. In other words, if all the levels satisfy the following condition, then the levels are totally ordered.

For any L_1, L_2 in L, either $L_1 \geq L_2$ or $L_1 \leq L_2$ must be defined, where L is a Universal set of all the security levels.

For instance, if L $\in$ {TOP SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED}, then L is totally ordered because all the security levels are comparable to one another, i.e.,

TOP SECRET > SECRET > CONFIDENTIAL > UNCLASSIFIED

In the implementation, security levels are treated as integers so that a system can then have as many security levels as possible, 0 being the lowest security level.

All the tuples are assigned confidentiality as well as integrity levels. Along with these, users can also insert private column value which is not supposed to be shown to any user. Even if a user has security level which is in compliance with the security level of tuple, then the user must not be allowed to view the value of that particular column. The value of that column is set as NULL just before displaying.

3.2.2.2 Information-flow semantics of SQL Queries

The functionalities of all the SQL queries, both DML as well as DDL statements are modified in order to incorporate the security policy within the *PostgreSQL*. However, the syntax and semantics of the queries remain the same which is one of the advantage of using this policy. Changing the syntax of the queries might yield inefficiency. The query execution follow *the best plan* approach. This section gives details of the changes made for each of the SQL query.

- **CREATE TABLE Query**
  This query creates a relation containing the attributes provided by the user. The attributes can have constraints like primary key, not null, check constraints, etc. The existing implementation of CREATE TABLE query is augmented to incorporate private column values as well. A user has an option to hide its private information, if required, from all other users. However, it can grant permission to some specific users to view the values. For instance, in the case of Hospital Management System, a patient might require to provide private information to doctor. In that case, she must have an option to make that value public just for her doctor. This is similar to Discretionary Access Control, where a user has control over the application of security policy.

  A flag, 0 or 1, is used to indicate whether a cell value is private or not and user provides the value of this flag at the time of tuple insertion. These private value flags are stored in a separate column created for each of the attributes. The attribute is declared as integer type and is created automatically when the user fires CREATE TABLE query, i.e., user need not define the column in the query explicitly. Along with the private columns, a label - confidentiality and integrity levels, is also maintained per tuple. So, two extra attributes are also added to

store the security levels of tuples. Both the columns are integer type and again user need not define it explicitly. It is automatically created when the query fires the query. Finally, the augmented relation contains:

- all the user defined columns,
- private column for each of the attributes,
- 1 attribute each for confidentiality and integrity levels.

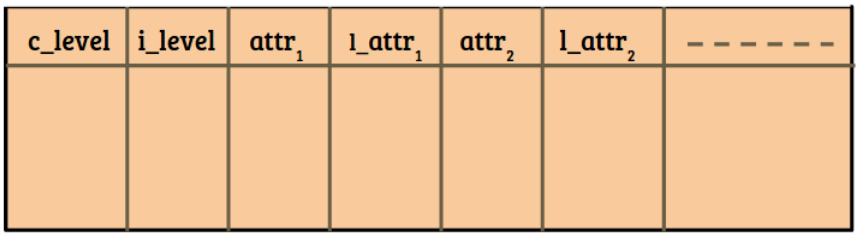

Figure 3.3: Schema of a table in *SecPostgreSQL*

The details of the implementation are as described below:

In the parsing phase, i.e. in *gram.y* file, a non terminal rule, *PrivateColumns*, is created which will be invoked for all the user defined attributes. This non terminal rule is also declared in *gram.y* file. So, even though the user has not defined the column in the query, the columns are created automatically. Also, for creating a separate attribute for accomodating confidentiality and integrity values for each tuple, another rule called *c_level* and *i_level* are created. Both of these non terminals are declared as tokens before defining the rules. Along with defining a few additional non terminal rules, the node structure for the respective statement is modified so that the changes get reflected in the further stages as well. CREATE TABLE is processed and executed by *utility processor* as DDL statements are straight forward and does not require any optimization and planning.

- **GRANT MAX_SECURITY_LEVEL**
  There are primarily, two types of subjects involved in the system - one is database admin and the other one is normal user. *PostgreSQL* behaves differently with both types of users. For database administrator, *PostgreSQL* bypasses all the security rules as it is assigned the highest

level of clearance and manages data of all the users. For normal users, all the SQL queries passes through reference monitors that performs the check before query execution.

Initially, database administrator assigns the maximum security level a subject can request to execute with. All the users in the system are assigned this level. Once the maximum level is granted, users can now request to execute queries with some specified levels. If the level requested is lower or equal to the maximum level, *PostgreSQL* grants it to the user. Otherwise an error is thrown stating *Requested Security level is higher than the maximum security level that can be assigned!.*

In order to grant maximum security levels, which contains both confidentiality and integrity levels, database administrator executes the following queries:

- BEGIN GRANT
  This query indicates that database admin is about to grant maximum security levels to the subjects. This query is like an indication that *PostgreSQL* can start storing values for all the subjects. The existing *PostgreSQL* does not include this query in their implementation. This is an additional query created. The syntax of the query is as follows:

  `BEGIN GRANT;`
- GRANT MAX_SECURITY_LEVEL
  This query grants maximum security level to the users specified. The full syntax of the query is as follows:

  `GRANT MAX_SECURITY_LEVEL <conf_level><integrity_level>TO <user_name>;`

  This is also an additional query created in *PostgreSQL*. The security levels are stored in information schema in the description column.
- COMMIT GRANT
  This query signals to store all the security levels assigned to all users in information schema. Earlier, while executing GRANT MAX_SECURITY_LEVEL, all the levels are still in main memory store in a data structure. Executing this query signals *PostgreSQL*

to make the changes permanent in the system. These changes retain in the database till the users are not granted different security levels to the users again.

- **GET SECURITY_LEVEL**
  This query is executed by normal users. It requests *PostgreSQL* to grant security levels if it is lower than or equal to the maximum security level a subject can get. After the query is executed, reference monitor performs a check the security level can be granted or not. If yes, all the queries executed after this would run with the assigned security level. Otherwise an error is thrown and default security level is granted, which is lowest confidentiality level and the highest integrity level. The syntax for requesting for a security level is as follows:

  `GET SECURITY_LEVEL <conf_level><integrity_level>;`

- **SELECT Query**
  This query retrieves tuples from the relations that satisfies the conditions provided by the user. The query contains the following main parts:

  – a target list, which contains the columns that are requested to be projected,
  – A table list, which contains list of relations from which tuples are to be fetched. This may contain a single relation as well as a JOIN of multiple relations,
  – A qualification clause, this contains the conditions of the WHERE clause and HAVING clause, if aggregate functions are being projected in the query.

  There are other clauses as well, like GROUP BY, ORDER BY etc., which do not require modifications in their implementations in order to adapt to our security policy.

  SELECT queries behave differently for database admin and other users in our case. Database admin gets all the tuples requested irrespective of the confidentiality level and integrity level of each row and also database admin can view private values as well. However this is not the case with other users.

  The overview of how the SELECT queries should work is as follows: The relation contains tuples that are assigned confidentiality and integrity levels. Also, the tuples can have private values for some attributes.

When a user logs in, it first requests for the confidentiality and integrity level to start with. In default case, it would be the lowest confidentiality level and the highest integrity level a subject can get. Now, when user fires the SELECT query, *PostgreSQL* perform check on the confidentiality levels of the tuples that satisfy the qualification provided by the user. The confidentiality levels of all the tuples to be displayed must be equal or lower than that of the user, i.e., the user must be at equal or higher confidentiality level in order to view the tuples. This way different users can get different views of database on the basis of their current level.

The detailed implementation is as follows:

After the SELECT query is executed by the user, *PostgreSQL* first checks if the SELECT query is a part if transaction or not. If it is a standalone query, then a new transaction session is created and a new memory context is also created for the query execution. But, if the SELECT query is a part of a transaction, then it shares the transaction session with the previous queries which have been part of the same transaction. Once the session has been created, the query now passes through the parsing stage.

The query string is passed to lexical analyzer and parser which is implemented using *flex* and *bison* in *PostgreSQL*. Syntax errors are reported by the parser if encountered any and the prompt on the client side starts asking for the next query. If no error found, the query tree is constructed and passed on to the analysis phase.

- **UPDATE Query**
  This query updates the values of data cells specified in the query for the tuples that satisfies the qualification specified. Update involves both reading as well as writing data. Tuples are first read from the tables to check if they are satisfying all the conditions specified in the qualification. If yes, then targeted columns are updated, else next tuple is fetched for checking. Since this involves both simultaneous reading and writing and our security model does not allow writing information to a lower level and reading data from upper level, the label of the subject must be the same as that of the tuple; i.e., c_level and i_level values of the tuple must be exactly same as that of subject's current c_level and i_level values. Also, if qualification of UPDATE query is performing a check on columns that are declared as private by the owner then that tuple must not be available for updation.

  All the changes made for UPDATE queries are done during the exe-

cution phase, which means the query plan has been optimized and the *best plan* is getting executed. This is one of the main advantage of our security model that it does not involve changing the syntax of SQL queries which might lead to inefficient query plan execution.

During the execution phase, in the *execScan()*, when the tuple slots are being retrieved for checking and updating, the execution flow is directed towards the *reference monitor*, which performs check on the security levels of the tuple. It also checks for the private column values in the where clause; if found, it discards that tuple and proceed with the next ones. A flag is maintained which is set to true when reference monitors finds that the tuple should be updated; otherwise it is set to false. This flag is used in *execUpdate()*, where the actual update is performed.

- **DELETE Query**
  This query deletes entire tuple that satisfies the qualification mentioned in it. Like Update, deleting a tuple also involves both reading and writing the data. Tuples are first read from the tables to check if they are satisfying all the conditions specified in the qualification. If yes, then that entire tuple is deleted, else the current tuple is discarded and next tuple is fetched for checking. Since this involves both simultaneous reading and writing and our security model does not allow writing information to a lower level and reading data from upper level, the label of the subject must be the same as that of the tuple; i.e., c_level and i_level values of the tuple must be exactly same as that of subject's current c_level and i_level values. Also, if qualification of DELETE query is performing a check on columns that are declared as private by the owner then that tuple must not be available for deletion.

  All the changes made for DELETE queries are done during the execution phase, which means the query plan has been optimized and the *best plan* is getting executed. This is one of the main advantages of our security model that it does not involve changing the syntax of SQL queries, which might lead to inefficient query plan execution.

  During the execution phase, in the *execScan()*, when the tuple slots are being retrieved for checking and updating, the execution flow is directed towards the *reference monitor*, which performs check on the security levels of the tuple. It also checks for the private column values in the where clause; if found, it discards that tuple and proceed with the next

ones. A flag is maintained which is set to true when reference monitors finds that the tuple should be deleted; otherwise it is set to false. This flag is used in *execDelete()*, where the actual delete is performed.

The execution flow for SELECT, UPDATE, and DELETE has been shown in the following figure.

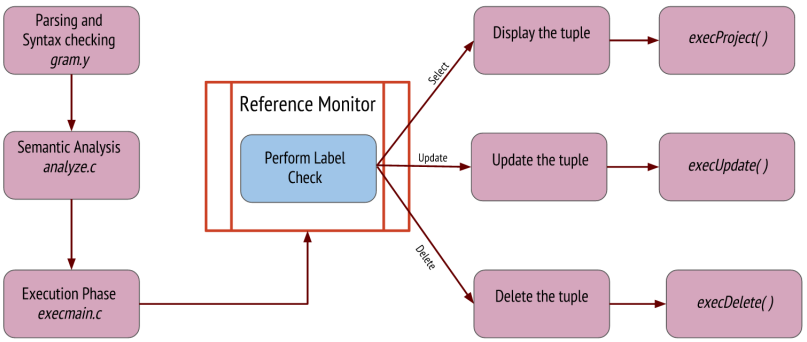

Figure 3.4: Integration of RWFM Monitor for Select, Update and Delete queries

- **INSERT Query**
  This query inserts a new tuple in a relation which is similar to creating objects in the systems. The tuple inserted has the same label (confidentiality and integrity levels) as that of the subject inserting it, like the case of creating new objects. Along with data values user also provides private flag values, which is either 0 or 1, to indicate if a column value is private or not. Each table has a primary key attribute. A user may try to insert duplicate key values. The existing *PostgreSQL* throws error for the same. However, since we have introduced security levels, not all the tuples are visible to a user. So, three cases may arise:

  – The primary key value is already present in the relation and is visible to the user, i.e., user is at a higher or equal confidentiality level as that of the tuple. In such cases, user must be denied to insert another tuple with the same primary key value.

  – The primary key value is already present in the relation and is not visible to the user, i.e., user is at a lower confidentiality level as that of the tuple. In such cases, user should be allowed to insert the tuple with the same confidentiality and integrity level as that of subject.

– The primary key is not present in the relation. In such cases, user must be able insert the tuple with the same confidentiality and integrity level as that of subject.

The execution flow for INSERT query has been shown in the following figure.

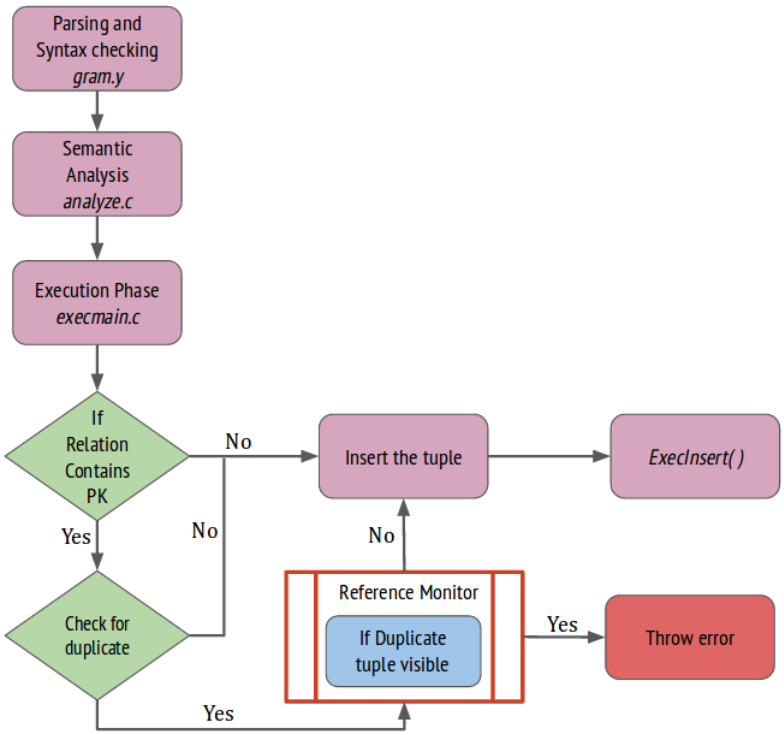

Figure 3.5: Integration of RWFM Monitor for Insert query

- **Transactions**
  Transactions are set of multiple SQL DML statements treated as a single logical operation. Transaction may contain SELECT, INSERT, UPDATE, DELETE queries. A single transaction is considered as a atomic operation which is either executed successfully or none of the statements get executed. So, they either commits or gets rolled back or aborted.

  When a subject executes a transaction, it is assigned a label which comprises of security level as well as RWFM label. The initial label of the subject contains security level, subject itself in the set of readers and empty set of writers. As subject reads tuples during a transaction,

primary key values are added to writers set and thus, its RWFM label moves up in the lattice i.e. it is now at a higher level in the lattice. This label is modified every time subject reads tuples. This label is valid till the lifetime of transaction. Hence, it need not be stored in a persistent storage. Now, when subject executes query other than SELECT, i.e., INSERT, DELETE and UPDATE, the following conditions arise:

  - The tuple to be inserted/updated/deleted is already been read by subject before, i.e. it is in the writers set of subject. If so, then subject must not be allowed to insert/update/delete the tuple.
  - The tuple to be inserted/updated/deleted is not read before, i.e. it is not in the writers set of subject. If so, subject is allowed to read/write/delete the tuple successfully, if the security rules also allow to do so.

- **CREATE VIEW**
  All the relations in the database are created by database administrator and normal users do not have direct access to the tables. This restriction is because all the users should not be permitted to view all the attribute values. Therefore, for normal users, database administrator creates views which are more like Access Views to the users. Users can insert, read, write and delete via access views only. Each view also has a label associated with it which is a combination of security levels and RWFM label. Security levels of the view is same as that with which the current query is being executed. Here, RWFM label is not the same as the one implemented in the previous approach. It obviously contains the set of readers and a set of writers but the subjects here is not the same as before. The readers set contains subject who is currently creating the view; while the writers set contains all the primary key values that are written in the view. Each primary key value is treated as a separate subject here.

  The existing implementation of CREATE VIEW query is changed to incorporate building labels for the entire view. This subject label is then stored in information schema along with other information of the view created. Later when the view is accessed, this label would be fetched by reference monitor in order to perform security rule checks.

  A subject can create a view of a relation if it wants to access only a few specific attributes of the relation. Creating a view is similar to executing a SELECT statement. The information flow rules are also

same as SELECT statement.All those tuples whose confidentiality level is not higher than the subject's confidentiality level and integrity level not lower than the subject's integrity level must be a part of the view; i.e., all those tuples which are at lower or equal level in the lattice can be read by the subject and hence, can be a part of the view. When it is getting computed, label of the view is also created and stored in information schema. The label contains the following components:

- Security level of the view, which is the same as that of owner's current security level.
- RWFM Label:

  Owner of the view

  Readers set containing only the subject creating the view.

  Writers set containing all the primary key values along with their security levels that are part of the view.

The security level of the view is stored so that it can be retrieved later when the view is accessed and is assigned to the subject accessing it, if it is in the readers set of the view. This implies, the view can be accessed by a subject which is in the readers set and all the tuples that are accessible with the security level of the view are available to the subject irrespective of its current security level.

The syntax for creating a view is same as that of creating a virtual view in SQL, which is as follows:

```
CREATE VIEW <view_name> AS <select_statement>;
```

During the execution phase, before scanning all the tuples, label of the view is allocated memory and is populated with subject's current security level, owner and readers set. Now, in *execScan()*, when the tuples are being checked for their qualification, execution flow is directed towards *reference monitor* to check if tuple can be read by the subject with the current security levels. If the tuple is visible to the subject, then the corresponding primary key value of that tuple is added to the writes set of view label; otherwise the tuple is discarded. Later, when the server is done with scanning all the tuples,i.e., in *executePlan()*, the label of the view is updated in information schema.

- **Declassifying Views**
  These views are created with an intention of making some of the at-

tributes of a tuple belonging to a higher security level to be made available to subjects that are at lower security level. The declassification is achieved by adding a user or multiple users to the readers set of the label of the view. When a subject, which is present in the readers set of a view tries to access it, the subject start operating at the level of the view irrespective of its current level and gets all the tuples that were a part of the view at the time of its creation. Declassification is a safe way of releasing information to subjects having lower security level.

The syntax of creating declassifying views is as follows:

```
CREATE VIEW <view_name> AS <select statement> WITH RELABEL
FOR '<subject name>';
```

The above statement creates a view with the same security level as that of subject's and the readers set contains the owner of the view and the subject for which relabelling has been done. The writers set as usual contains all the primary key values with their security levels of those tuples that are part of the view.

During the start of execution phase, label of the view is allocated memory and is populated with populated with subject's current security level, owner and readers set. Now, in *execScan()*, when the tuples are being checked for their qualification, execution flow is directed towards *reference monitor* to check if tuple can be read by the subject with the current security levels. If the tuple is visible to the subject, then the corresponding primary key value of that tuple is added to the writes set of view label; otherwise the tuple is discarded. Later, when the server is done with scanning all the tuples,i.e. in *executePlan()*, the label of the view is updated in information schema.

- **Sanitized views**
  Sanitized views are same as Declassifying views, the only difference being view is made available to all the subjects in the system i.e. it is made public. The implementation approach is similar to that of declassifying well. Like declassifying views, subject trying to access the view is first checked in its readers set; if present, it is assigned the same security level as that of the view and then all the tuples having same primary key values are fetched and returned. The syntax for creating sanitized view is as follows:

```
CREATE SANITIZED VIEW <view_name> AS <select statement>;
```

The above query first retrieves all the subjects present in the system and then append them to the readers set. The label of the view is then stored in information schema so that it can be retrieved later when the view is accessed.

A brief summary describing the semantics of SQL queries is presented below.

- **CREATE TABLE Query**

  CREATE TABLE query creates a new user defined relation with extra attributes to accommodate confidentiality level, integrity level, and private or non private values for each of the user defined attributes. These extra columns are not explicitly defined by user in query, which means that syntax of CREATE TABLE remains intact for users.

- **GRANT / GET SECURITY_LEVEL Query**

  GRANT MAX_SECURITY_LEVEL SQL construct has been created for database administrators to assign the highest confidentiality and the lowest integrity level at which a user can execute queries.

- **GET SECURITY_LEVEL Query**

  This query has been created for users to request for a certain level of confidentiality and integrity. The levels requested are then validated to check they are not higher than the maximum levels assigned by database administrator.

- **SELECT Query**

  SELECT queries retrieve tuples whose confidentiality and integrity pair is dominated by subject's label pair. Formally, if D is the set of tuples in the database, then the result of the query is equivalent to that of a standard SQL query on database D' , where $D' = \{\tau | \tau \epsilon D \wedge (\tau.L_C, \tau.L_I) \leq (S.L_C, S.L_I)\}$.

  All complex SELECT statements including JOIN, aggregate functions follow the same rule; however the semantics of the statements remain the same.

- **INSERT Query**

  INSERT query adds new tuples in a relation specified with values for all the user defined attributes as well as for their corresponding private/non private columns. INSERT query fired on a relation containing primary key first validate for the uniqueness constraint only on the

subset of tuples whose security levels are dominated by that of subject's level. If the key value exists in that subset, then an error is thrown; else the tuple is inserted irrespective of the existence of same key value outside the subset. *Polyinstantiation* prevents insertion of duplicate tuple at a higher security level.

- **UPDATE Query**

  UPDATE query requires both reading the tuples for finding the subset that matches the qualification provided by user as well as writing to the matching tuples found while reading. Therefore, UPDATE query only updates the tuples that have same level as that of the subject as tuples with more restrictive labels are not visible to the subject and tuples with less restrictive labels cannot be written by the subject without violating the information flow rules.

- **DELETE Query**

  DELETE query also requires reading the relation for finding qualifying tuples for deletion. Therefore, this query can only delete tuples having same level as that of the subject because like UPDATE, tuples with more restrictive levels can not be read and tuples with less restrictive levels can not be written without violating information flow rules.

- **CREATE VIEW Query**

  CREATE VIEW query creates an access view with a label comprising of security level pair and a Readers Writers Flow Model label. The security level is same as that of the subject creating the view, readers set of RWFM label contains the current subject itself and writers set contains primary key values of all the tuples that are part of the view. Access views are later retrieved with the same security level as that of view irrespective of the subject's current security level.

- **SANITIZATION AND DECLASSIFYING VIEWS**

  DECLASSIFYING VIEWS downgrades a view at higher security level to be visible to some specific subjects that are at lower security level. Declassifying the views adds other subjects to the readers set of view label in order to make it available to them.

SANITIZATION is same as declassification with the only difference being that the view is made public to all the subjects present in the system i.e. all the subjects are added to the readers set of view label.

In this approach, unlike the approach we adopted in previous section (RWFM), transactions are transparent to the users. All the changes made to the semantics of individual queries would be sufficient to make transactions run the desired way, i.e., information flow rules will be enforced on all query executions transparently.

Chapter 4

Case Study - Conference Management System HotCRP

HotCRP is a widely used conference management system, which is rich in security policies for sharing information among users. The existing implementation of HotCRP is vulnerable to information leakage and has various bugs and covert channels, that helped in exposing contact information of all the participants. HotCRP has authors, reviewers and program chair as users. Authors submits their papers, reviewers review papers that are allotted to them for review and program makes the final decision and announces it to users. Program Chair also defines conflict of interest so that other reviewers can not access reviews of their own paper or papers of their close colleagues. Other features are also included, such as, making all the reviews submitted to be available to top few authors, who got maximum score in the review process. This can be achieved when program chair declassifies reviews for all those authors. Program can also sanitize information to make it available to all of the subjects in the system.

The conference management system has a predefined set of subjects and objects, which is why RWFM can be used to restrict invalid information flow in the system. The only difference is that of the additional security rules that are defined by program chair to provide a fair review system should also be taken care of. All these rules are provided to the system as input in the form of files. Program chair assigning papers to reviewers follows Discretionary Access Model. So, this case-study is a blend of both DAC as well as MAC models.

4.1 System Specification

The system constitutes a well defined set of subjects and a set of objects. There are primarily three major roles having different privileges, which are as follows:

- Program chair, denoted by PC.
- A set of reviewers, denoted by R.
- A set of authors, denoted by A.

The objects in the system are as follows:

- A set of all papers, denoted by P.
- A set of all reviews submitted, denoted by Re.

Some of the basic security rules are defined which are to be conformed to in order to provide a fair review system. They are as follows:

- Authors can not review a paper.
  i.e. $A \cap R = \phi$

- Program chair is not allowed to review any paper. i.e. $PC \cap R = \phi$
- Reviewers are not allowed to access reviews of reviewers that are present in the conflict set. The conflict set is defined by the Program Chair before assigning the papers for reviews to all reviewers. The set is a mapping between a paper and a set of reviewers. All the reviewers belonging to the set are not permitted to access reviews of that particular paper even if Program Chair declassifies reviews to be available to all the reviewers. The conflict set is defines as follows:

  Conflict (C) : $P -> 2^R$

  For a paper p_i, Conflict set can be defined as follows:

  $C(p_i) = \{\text{r1, r2}\}$ where r1, r2 are the reviewers.

- Program chair assigns paper to be reviewed to all reviewers present in the system. No other reviewer is allowed to review that particular paper other than the ones present in assigned set. This assignment function is defined as follows:

  Assignment(AS) : $P -> 2^R$

  and $\forall p_i \in P$, $AS(p_i) \cap C(p_i) = \phi$

- A reviewer is permitted to see reviews submitted by other reviewers only after he/she has submitted review for the papers assigned.

4.2 Database Schema

A part of schema of conference management system has been taken to demonstrate how information flow is tracked and how the application has been made more secure while also abiding by the security policy. The schema contains information about the papers submitted by authors, reviews submitted by reviewers and the final decision taken by Program Chair for each of the papers submitted.

The database contains following relations:

- ***Paper*** (*paper_id*, *paper_title*, *author*, *score*)

- ***Paper_Review*** (*paper_id*, *review_id*, *reviewer*, *review*)

- ***Paper_Decision*** (*paper_id*, *decision*)

The above relations are user defined relations; however the actual relation stored in database contains the label columns as well for each of the user defined attributes. The exact structure of each of the relations is depicted below.

paper_id	*l_paper_id*	*paper_title*	*l_paper_title*	*author*	*l_author*	*score*	*l_score*

Table 4.1: Structure of *Paper* table

paper_id	*l_paper_id*	*review_id*	*l_review_id*	*reviewer*	*l_reviewer*	*review*	*l_review*

Table 4.2: Structure of *Paper_Review* table

paper_id	*l_paper_id*	*decision*	*l_decision*

Table 4.3: Structure of *Paper_Decision* table

Papers submitted by authors are stored in *Paper* table and they can be accessed by their respective authors, assigned reviewers and Program Chair. Reviews submitted are stored in *Paper_Review* table and are accessible to reviewers and Program Chair; but, they have been influenced by their respective authors and reviewers. Ultimately, the decision is taken on the basis of by the Program Chair on the basis of the reviews submitted and is stored in *Paper_Decision* table. This decision is accessible only to Program Chair, which has to be made available to other reviewers and authors by declassifying them.

4.3 Example

All the above discussed security policies have been implemented and tested to check whether information flow is successfully getting tracked or not. Some of the bugs in the classic HotCRP are also prevented by enabling this information flow control. This example demo also focuses on the same relations as discussed above, i.e. *Paper*, *Paper_Review* and *Paper_Decision* with the similar structure.

4.3.1 Roles in the system

Role	Persons
Program Chair	Alice
Reviewers	Bob, John, Harry
Authors	Cathy Sam, James

Table 4.4: Various principals in the example

Reviewers submit the assigned reviews. The content of *paper_review* relation after submitting is shown in Table 4.5

Paper ID	Review ID	Reviewer name	Review
1	1	Bob	1
3	2	Bob	-1
1	1	John	1
2	2	John	-1

Table 4.5: Paper Review Table content

Program chair provides Reviewer Assignment after authors submits their papers, on the basis of which the initial labels of paper and reviews are computed. The papers submitted can be read by Program Chair, assigned reviewers and the author itself and has been influenced by author itself. The submitted review can be read by Program Chair and the reviewer itself and has been influenced by the reviewer who submitted that review and the author of that paper. Initially, paper decisions can only be read by Program Chair and has been influenced by all the authors who submitted the paper, all the reviewers who reviewed the paper and Program Chair. All these information can be declassified later at any point of time to make it available to other subjects, who have influenced it before. For instance, paper decision is later declassified by Program Chair to authors and reviewers once the decision is made. Also after releasing the decision, Program Chair can also declassify the reviews submitted to authors of top few papers who got the highest scores.

First, authors submit their papers. In our example, Cathy, Sam and James submit their respective papers and after submitting, content of *paper* relation is shown in Table 4.4

Paper ID	Name	Author	Score
1	Securing PostgreSQL	Cathy	0
2	Decentralized Information Flow Control	Sam	0
3	Views in MLS databases	James	0

Table 4.6: Paper Table content

Reviewers submit the assigned reviews. The content of *paper_review* relation after submitting is shown in Table 4.5

Once the reviews of a paper are submitted, Program Chair updates scores

Paper ID	Review ID	Reviewer name	Review
1	1	Bob	1
3	2	Bob	-1
1	1	John	1
2	2	John	-1

Table 4.7: Paper Review Table content

obtained by each paper from all the reviewers. As per the RWFM rule, Program Chair can update score if their labels are exactly the same. The syntax of updating scores is:

```
UPDATE Paper SET score = (SELECT SUM(review) FROM Paper_review
WHERE paper_id = <paper_id>) WHERE Paper_id = <paper_id>;
```

Program Chair can declassify the reviews submitted to only those reviewers who have submitted at least one review. This is achieved by executing transaction as for declassification, subject has to be in the influencers list. Therefore, Program Chair first accesses all the reviews submitted which adds all the reviewers in its writers list. Now, it can perform declassification to make the reviews available to other reviewers. The syntax for declassification is as follows:

```
BEGIN;
SELECT review FROM paper_review;
DECLASSIFY_ALL REVIEW;
COMMIT;
```

Chapter 5

Performance Analysis

Both *PostgreSQL* and *SecPostgreSQL* have been run over a number of queries using TPC-H benchmarking tool for measuring the performance overhead. Benchmarking were done for the general DML queries such as Select, Insert, Delete and Update. Time taken was plotted for doing the specified operation on a discretely step-wise increasing number of tuples. SecPostgreSQL demonstrated a slight degradation in performance in each type of query. This can be attributed to the extensive computations for the various checks that are being done **for each tuple**. For example, to display contents of a table, only those tuples should be displayed which satisfies the MLS constraints. This check is done for each of the tuples that is being fetched. Thus the performance degrades slightly.

5.1 System Specification

The following are the system specification on which the benchmarks were run:

- **Processor**: Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz
- **Memory**: 8 GB
- ***PostgreSQL* version**: 9.4.4
- **Database specification**: TPC-H benchmarking database

The database schema was used as it is for measuring the performance of the existing *PostgreSQL* implementation. But *SecPostgreSQL* uses a slight *variant* of table schema, i.e., it stores extra columns for each attribute to specify whether it is private or non-private. Thus the Insert query needed to

be modified to accommodate the same. The rest of the queries are used as they are used in classic *PostgreSQL*. Now the performance for each query is illustrated in the further sections.

5.2 SELECT Query

A complex query has been used to encompass the various flavors of SELECT queries. The database schema used is same as in TPC-H. Hence the details regarding the semantic of each attribute is omitted here. The query is:

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

An average overhead of 33.6% has been measured as compared to classic *PostgreSQL*, in terms of time taken with respect to number of tuples in the tables.

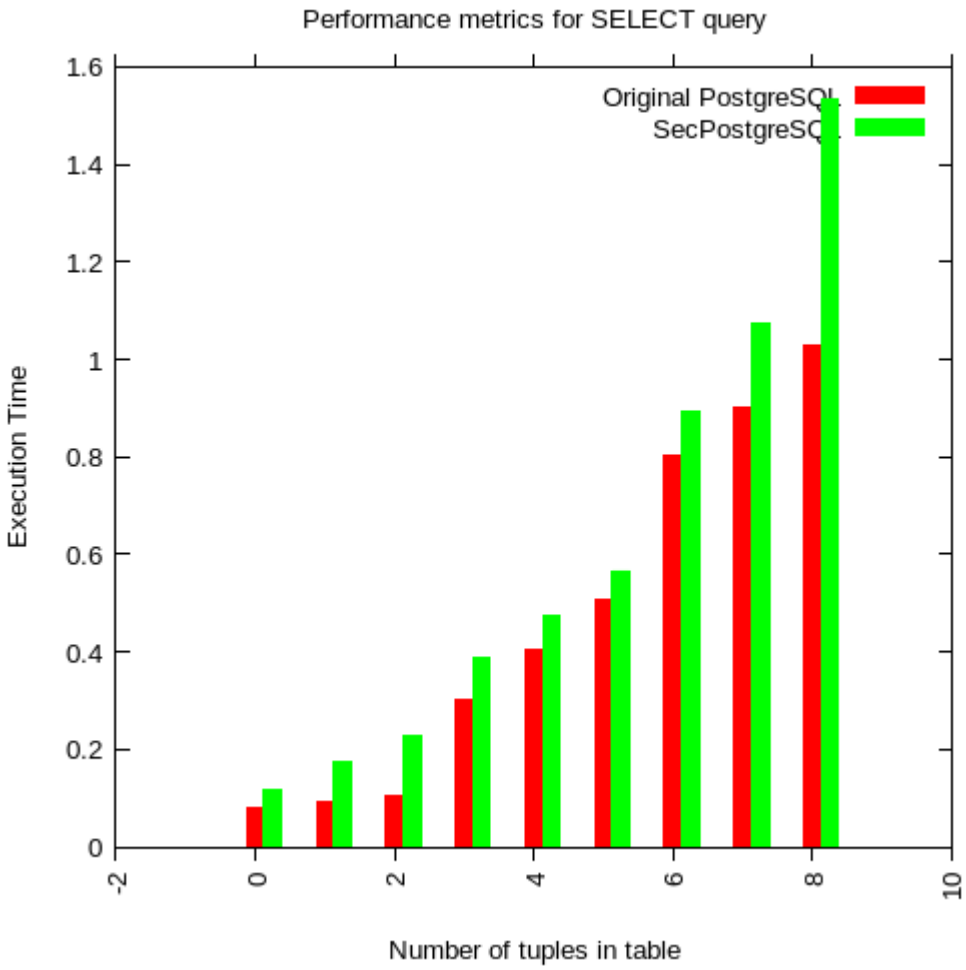

Figure 5.1: Performance of SecPostgreSQL with Select queries

5.3 INSERT Query

Here, the number of insert queries has been gradually increased to measure the performance overhead *SecPostgreSQL* in comparison to classical *PostgreSQL*. The performance overhead found to be 18.76% on an average.

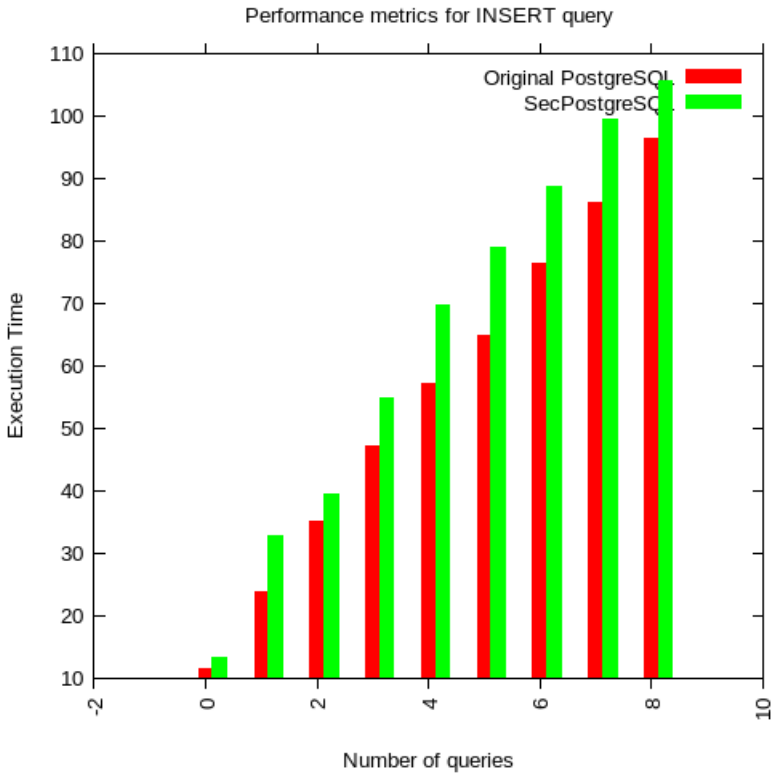

Figure 5.2: Performance of SecPostgreSQL with Insert queries

5.4 UPDATE Query

The performance overhead measurement was done in a similar fashion to that done for Insert. Several numbers of gradually increasing update queries were ran and performance overhead was observed and was found to be, on an average, 16.95%.

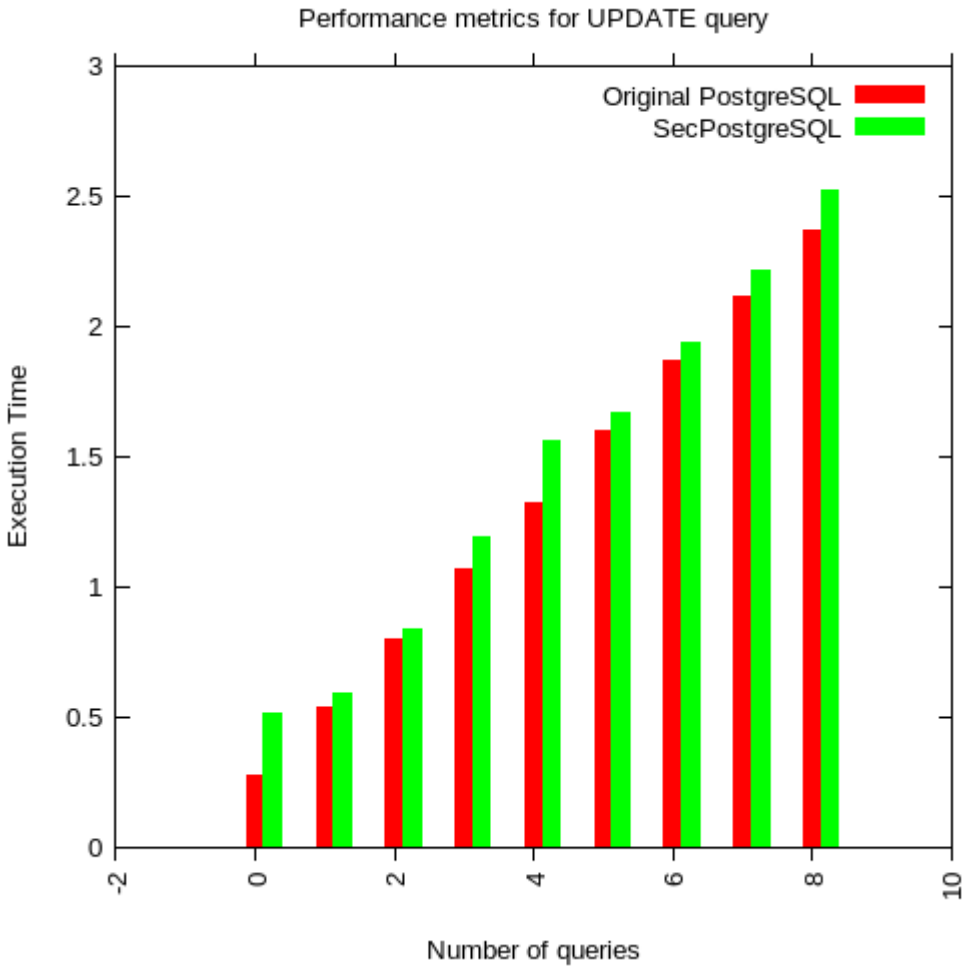

Figure 5.3: Performance of SecPostgreSQL with Update queries

5.5 DELETE Query

The delete queries were executed on an already populated database table. On an average, a performance overhead of 23.7% was measured as compared to classical *PostgreSQL*.

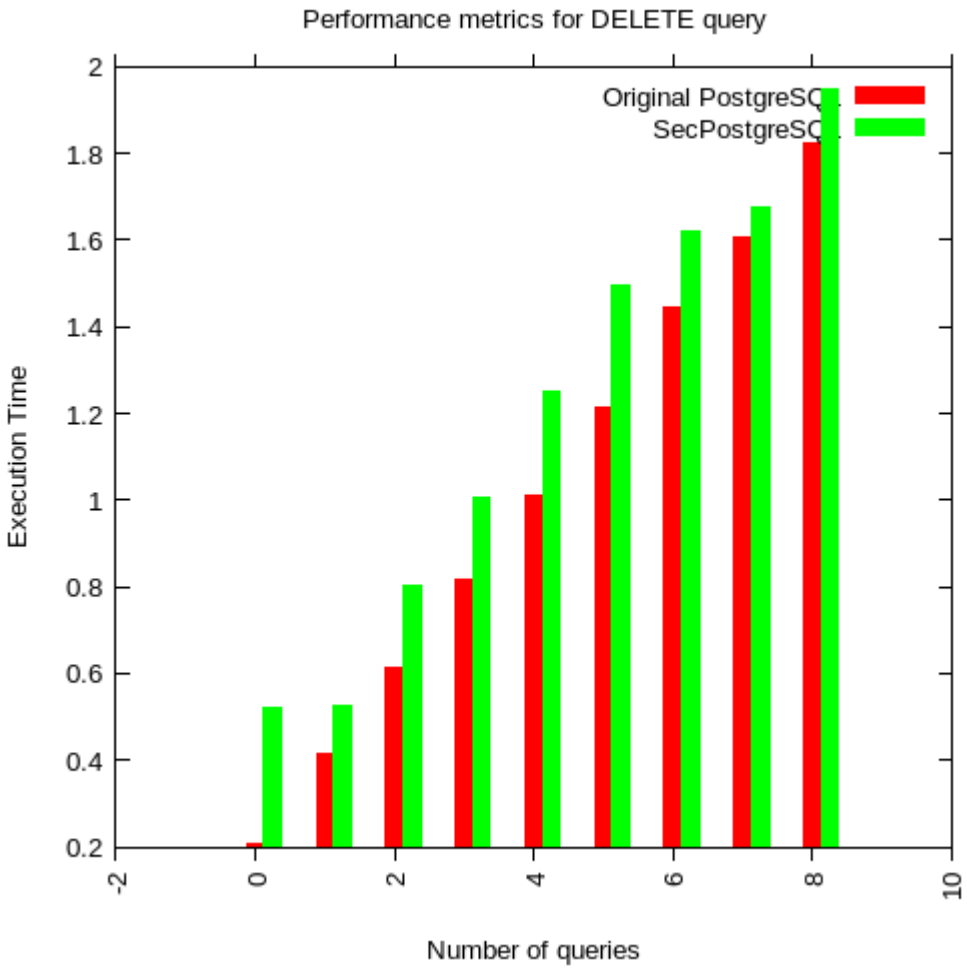

Figure 5.4: Performance of SecPostgreSQL with Delete queries

Chapter 6

Conclusion

SecPostgreSQL has been successfully designed and developed to inhibit all invalid information flows. It provides support for realizing multi-level secure databases. It handles applications using two different security policies, depending on the role of subjects available in the system: RWFM based approach and tuple-level labeling approach. Both of these approaches have been implemented with decentralized information flow control model RWFM and tuple-level security to provide support for MLS databases. The performance is measured with the classical *PostgreSQL* using the benchmark tool TPC-H [15] and the results are found satisfactory; i.e. the overhead computed is not much significant against the privacy benifits *SecPostgreSQL* introduces.

The correctness has been proved in a generalized manner through various case studies, for both tuple-level labeling approach as well the RWFM based approach. Classic *PostgreSQL* has been extended to incorporate RWFM model and is tested on a popular application for conference management system: *HotCRP* [16]. All the access to data objects is made to pass through reference monitor, which prevents bugs from leaking information after executing multiple queries in succession.

Chapter 7

Future Work

Current implementation of *SecPostgreSQL* encompasses the universe of CREATE, SELECT, INSERT, UPDATE, DELETE, VIEW and declassification. Further works that can further enhance *SecPostgreSQL* are:

- Handling the complex SQL queries such as triggers and stored procedures. This could be quite helpful in commercial database systems such as in case of banks, where many integrity checks are performed at a trigger level, such as the maximum withdrawal limit of an account may be, say, ₹20,000. Extending *SecPostgreSQL* to handle triggers and stored procedures will make information flow control at that level, and may also help to prevent information leak via covert channels.
- Another area where further work can be done is in the case of variants of SQL, such as PL/SQL. Support for such variants can also help make *SecPostgreSQL* more generalized and flexible to use.

Bibliography

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

[2] 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

[3] Myers, Andrew C and Liskov, Barbara, *A decentralized model for information flow control*, ACM, 31.5, 1997.

[4] Denning, Dorothy E and Lunt, Teresa F and Schell, Roger R and Shockley, William R and Heckman, Mark, *The SeaView security model*, IEEE, 218-233, 1988.

[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] NV Narendra Kumar, R.K.Shyamasundar, *RWFM Model: Soundness and Completeness*, manuscript, 2015.

[7] S Susheel, NV Narendra Kumar and R.K.Shyamasundar, *11 Int Conf on Information System Security (ICISS 2015)*, 2015.

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

[9] Robling Denning, Dorothy Elizabeth, *Cryptography and data security*, Addison-Wesley Longman Publishing Co., Inc., 1982.

[10] Bell, D Elliott and LaPadula, Leonard J, *Secure computer systems: Mathematical foundations*, DTIC Document, 1973.

[11] Jajodia, Sushil and Sandhu, Ravi, *Toward a multilevel secure relational data model*, ACM SIGMOD Record, 20(2):50-59, 1991.

[12] Cheng, Winnie and Ports, Dan RK and Schultz, David A and Popic, Victoria and Blankstein, Aaron and Cowling, James A and Curtis, Dorothy and Shrira, Liuba and Liskov, Barbara, *Abstractions for Usable Information Flow Control in Aeolus*, USENIX Annual Technical Conference, 139-151, 2012.

[13] PostgreSQL *https://www.postgresql.org/* Accessed June 2017

[14] PostgreSQL Developer Tool *https://www.postgresql.org/files/developer/tour.pdf* Accessed June 2017

[15] TPC-H Benchmark *http://www.tpc.org/tpch/* Accessed June 2017

[16] HotCRP: Conference Management System *https://hotcrp.com/* Accessed June 2017

[17] Cartel: Car Drives Monitoring Application *http://cartel.csail.mit.edu/doku.php* Accessed June 2017

[18] Statistical Databases: *https://en.wikipedia.org/wiki/Statistical_database* Accessed June 2017.

[19] Multi-level Security *https://en.wikipedia.org/wiki/Multilevel_security* Accessed June 2017.

[20] Biba, K.J., *Integrity Considerations for Secure Computer Systems*, MTR-3153, The Mitre Corporation, June 1975.

[21] Dr. David F.C. Brewer and Dr. Michael J. Nash, *The Chinese Wall Security Policy* IEEE, 1989.