CERIAS Tech Report 2010-32
Mechanisms for database intrusion detection and response
by Ashish Kamra
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Ashish Kamra

Entitled Mechanisms for Database Intrusion Detection and Response

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

A. Ghafoor, Co-Chair
Chair

E. Bertino, Co-Chair

A. Raghunathan

N. Li

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): E. Bertino, Co-Chair

Approved by: V. Balakrishnan 7/06/10
Head of the Graduate Program Date

Graduate School Form 20
(Revised 10/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation: Mechanisms for Database Intrusion Detection and Response

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Ashish Kamra
Signature of Candidate

7/06/10
Date

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

MECHANISMS FOR DATABASE INTRUSION DETECTION AND RESPONSE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ashish Kamra

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2010

Purdue University

West Lafayette, Indiana

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन ।
मा कर्मफलहेतुर्भू: मा ते संगोऽस्त्वकर्मणि ।।
(२—४७)

Your freedom is only in the field of action, and not in the field of bringing about the fruits of action. Never take yourself as the cause of bringing about a situation, and never resort to a life of inaction.

- Shrimad Bhagwad-Gita

ACKNOWLEDGMENTS

I would like to extend by sincere gratitude to my adviser, Prof. Elisa Bertino, for her unwavering support and guidance over the years. Very simply put, there would not have been a *Dr. Ashish Kamra* without her constant encouragement and guidance.

I would like to show my appreciation to my wife, Monalisa Hota, for bearing with me (wholeheartedly I suppose !) over the last few years since we have been married. I was close to leaving the PhD program once but stuck it out largely because of her encouragement(and insistence !).

I would also like to thank my academic committee members, Prof. Ninghui Li and Prof. Anand Raghunathan for their valuable inputs. I would like to thank Prof. Arif Ghafoor for agreeing to be my co-adviser from the ECE department and for providing very valuable inputs during the preliminary and the final exams that have gone a long way in improving the quality of this work. I would also like to thank Prof. Guy Lebanon, my former adviser, for having helped me out and providing the much needed advice whenever I required.

I would like to thank my family for standing firmly behind me during the PhD years and for never doubting my intent or ability to complete the degree.

Finally, I would like to thank my friends and colleagues who have helped me in one way or another over the course of this endeavor. They know who they are but I would like to specially acknowledge Evimaria Terzi, Rimma Nehme, Abhilasha Bhargav-Spantzel, Ji-Won Byun, Ashish Kundu, and Mohamed Shehab for their unconditional support, guidance, and encouragement.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

DBMS	DataBase Management System
DBA	DataBase Administrator
ID	Intrusion Detection
AD	Anomaly Detection
NBC	Naive Bayes Classifier
IDR	Intrusion Detection and Response
RBAC	Role Based Access Control
ECA	Event Condition Action
PSAC	Privilege State based Access Control
JTAM	Joint Threshold Administration Model

ABSTRACT

Kamra, Ashish. Ph.D., Purdue University, August 2010. Mechanisms For Database Intrusion Detection And Response. Major Professors: Elisa Bertino and Arif Ghafoor.

Data represent today a valuable asset for companies and organizations and must be protected. Most of an organization's sensitive and proprietary data resides in a Database Management System (DBMS). The focus of this thesis is to develop advanced security solutions for protecting the data residing in a DBMS. Our approach is to develop an Intrusion Detection and Response (IDR) system, integrated with the core DBMS functionality, that is capable of detecting and responding to anomalous SQL commands submitted to a DBMS. For the intrusion detection mechanism, the key idea is to learn profiles of database users from the SQL commands submitted by them to the DBMS. A SQL command that deviates from these profiles is then termed as anomalous. For responding to such anomalous and potentially malicious SQL commands, we introduce a policy-driven intrusion response mechanism that is capable of issuing an appropriate response based on the details of the anomalous request. Such response actions include fine-grained actions such as request suspension and request tainting; we introduce an access control system based on the notion of privilege states to support such fine-grained responses. For the management of the response policies, we introduce a joint threshold administration model that mitigates the risk of insider threats from malicious database administrators. A major component of the thesis involves prototype implementation of the IDR mechanism in the PostgreSQL DBMS. We discuss the implementation details on the same and report experimental results that show that our techniques are feasible and efficient.

1. INTRODUCTION

1.1 Database Intrusion Detection

Data represent today an important asset for companies and organizations. Some of these data are worth millions of dollars and organizations take great care at controlling access to these data, with respect to both internal users, within the organization, and external users, outside the organization. Data security is also crucial when addressing issues related to privacy of data pertaining to individuals; companies and organizations managing such data need to provide strong guarantees about the confidentiality of these data in order to comply with legal regulations and policies [1]. Overall, data security has a central role in the larger context of information systems security. Therefore, the development of Database Management Systems (DBMSs) with high-assurance security (in all its flavors) is a central research issue. The development of such DBMSs requires a revision of architectures and techniques adopted by traditional DBMS [2]. An important component of this new generation security-aware DBMS is an Intrusion Detection (ID) mechanism. Even though DBMSs provide access control mechanisms, these mechanisms alone are not enough to guarantee data security; they need to be complemented by suitable ID mechanisms. However, despite the fact that building ID systems for networks and operating systems has been an active area of research, few ID systems exist that are specifically tailored to DBMS. Why is it important to have an ID mechanism tailored for a DBMS? The main reason is that actions deemed malicious for a DBMS are not necessarily malicious for the underlying operating system or the network; thus ID systems designed for the latter may not be effective against database related attacks. For example, consider that a database user/application normally access data only from the human resources schema. Consider that such user/application submits a SQL command to the DBMS that accesses the financial records of the employees from the finance schema. Such anomalous access pattern of the SQL command

may be the result of a SQL Injection vulnerability or privilege abuse by an authorized user. The key observation is that an ID system designed for a network or an operating system is ineffective against such database specific malicious actions.

1.2 DBMS Integration

Organizations have stepped up data vigilance driven by various government regulations concerning data management such as SOX, PCI, GLBA, HIPAA and so forth [3, 4]. The compliance regulations have led the organizations to use various third-party database activity monitoring products that employ DBMS specific ID techniques [3]. Such products are useful for many reasons. One, they are DBMS technology independent thus they can work with multiple DBMS vendors. Two, they are mostly non-intrusive since their core functionality resides outside the DBMS. However, one of the goals of this thesis is to integrate our DBMS specific ID mechanism with the core DBMS functionality. There are three main advantages of such close integration of an ID mechanism with a DBMS. First, the intrusion detection is done as close to the target as possible (during query processing) thereby ruling out any chances of a backdoor entry to the DBMS that may bypass the ID mechanism. Second, since the ID mechanism is presented as one of the features of the DBMS, the physical location of the DBMS is not a constraint on obtaining the ID service. Such requirement is critical in the current age of cloud computing if the organizations want to move their databases to a cloud service provider. The problem with the third-party activity monitoring products is that the organizations may not be able to move them to the cloud since the network infrastructure is under the control of the cloud service provider. Third, integration with the DBMS allows the ID mechanism to issue more versatile response actions to an anomalous request. We expand on the response capabilities of our system in Chapter 4.

1.3 Insider Threats

The problem of insider threats to DBMSs is being recognized as a major security threat by the organizations; in a 2004 E-crime watch survey conducted by CERT and US Secret Service, insider threat was identified as the second biggest threat after hackers. The solution to the insider threat problem requires among other techniques the adoption of mechanisms able to detect and respond to access anomalies by users internal to the organization owning the data. For our IDR system to provide stronger security guarantees, it needs to ensure that the activities of even the database administrators (DBAs) be monitored, and responded to if deemed malicious. This is a difficult problem to address since the policies that specify a response action need to be created for the DBAs who are, in turn, responsible for managing the same policies. We describe our approach in Chapter 3 to address this problem.

1.4 Overview of Our Approach

1.4.1 Intrusion Detection

Our approach to the intrusion detection and response (IDR) mechanism consists of two main elements, specifically tailored to a DBMS: an anomaly detection (AD) system, and an anomaly response system. The first element is based on the construction of access profiles of users, roles and applications, and on the use of such profiles for AD. The first application of our AD system is detection of anomalous database access patterns of users/roles (Chapter 2). The AD system in this case considers *two* different scenario. In the first scenario, it is assumed that the database has a Role Based Access Control (RBAC) model in place. The AD system is able to determine role intruders, that is, individuals that while holding a specific role, behave differently than expected. When role information does exist, the problem is transformed into a supervised learning problem. The roles are treated as classes for the classification purpose. The AD task for this setting is as follows: For every user request under observation, its role is predicted by a trained classifier. If the predicted role is different from the original role associated with the query, an anomaly is detected. In the

second case, the same problem is addressed in the context of a DBMS without any role definitions. In such setting, every SQL command is associated with the user that issued it. We build user-group profiles (clusters of similar behaviors) based solely on the SQL commands users submit to the database. The specific methodology that is used for the AD task is as follows: the training data is partitioned into clusters using the standard clustering techniques. A mapping is maintained for every database user to its representative cluster. For every new query under observation, its representative cluster is determined by examining the user-cluster mapping. For the detection phase, two different approaches are followed. In the first approach, the classifier is applied in a manner similar to the supervised case, to determine whether the user associated with the query belongs to its representative cluster or not. In the second approach, a statistical test is used to identify if the query is an outlier in its representative cluster. If the result of the statistical test is positive, the query is marked as an anomaly and an alarm is raised. In order to build profiles, the log-file entries need to be pre-processed and converted into a format that can be analyzed by the detection algorithms. Therefore, each entry in the log file is represented by a basic data unit that contains five fields, and thus it is called a *quiplet*. The abstract form of a quiplet consists of five fields (*SQL Command, Projection Relation Information, Projection Attribute Information, Selection Relation Information and Selection Attribute Information*). Depending on the level of detail required in the profile construction phase and in the AD phase, the quiplets are captured from the log file entries using *three* different representation levels. Each level is characterized by a different amount of recorded information. For more details of this approach, we refer the reader to Chapter 2.

1.4.2 Anomaly Response

The second element of our approach is in charge of taking some actions once an anomaly is detected. There are three main types of response actions, that we refer to respectively as conservative actions, fine-grained actions, and aggressive actions. The conservative actions, such as sending an alert, allow the anomalous request to go through, whereas

the aggressive actions can effectively block the anomalous request. Fine-grained response actions, on the other hand, are neither conservative nor aggressive. Such actions may *suspend* or *taint* an anomalous request. A suspended request is simply put on hold, until some specific actions are executed by the user, such as the execution of further authentication steps. A tainted request is marked as a potential suspicious request resulting in further monitoring of the user and possibly in the suspension or dropping of subsequent requests by the same user. For more details on our approach towards supporting such fine-grained response actions in a DBMS, we refer the reader to Chapter 4.

With such different response options, the key issue to address is which response measure to take under a given situation. Note that it is not trivial to develop a response mechanism capable of automatically taking actions when abnormal database behavior is detected. Let us illustrate this with the following example. Consider a database monitoring system in place that builds database user profiles based on SQL queries submitted by the users. Suppose that a user U, who has rarely accessed table T, issues a query that accesses all columns in T. The detection mechanism flags such request as anomalous for U. The major question is what should the system do next once a request is marked as anomalous by the AD mechanism. Since the anomaly is detected based on the learned profiles, it may well be a false alarm. It is easy to see then there are no simple intuitive response measures that can be defined for such security-related events. If T contains sensitive data, a strong response action is to revoke the privileges corresponding to actions that are flagged as anomalous. In our example, such a response would translate into revoking the select privilege on table T from U. However, if the user action is a one-time action part of a bulk-load operation, when all objects are expected to be accessed by the request, no response action may be necessary. The key idea is to take different response actions depending on the details of the anomalous request, and the context surrounding the request (such as time of the day, origin of the request, and so forth). Therefore, a *response policy* is required by the database administrator to specify appropriate response actions for different circumstances. In Chapter 3, we propose a high-level language for the specification of such policies which makes it very easy to specify and modify them.

The two main issues that we address in the context of such response policies are that of *policy matching* and *policy administration*. Policy matching is the problem of searching for policies applicable to an anomalous request. When an anomaly is detected, the response system must search through the policy database and find policies that match the anomaly. Our ID mechanism is a real-time intrusion detection and response system; thus efficiency of the policy search procedure is crucial. In Chapter 3, we present two efficient algorithms that take as input the anomalous request details, and search through the policy database to find the matching policies. We implement our policy matching scheme in the PostgreSQL DBMS [5], and discuss relevant implementation issues. We also report experimental results that show that our techniques are very efficient.

The second issue that we address is that of administration of response policies. Intuitively, a response policy can be considered as a regular database object such as a table or a view. Privileges, such as *create policy* and *drop policy*, that are specific to a policy object type can be defined to administer the policies. However, a response policy object presents a different set of challenges than other database object types. Recall that a response policy is created to select a response action to be executed in the event of an anomalous request. Consider the case of an anomalous request from a user assigned to the database administrator (DBA) role. Since a DBA role is assigned all possible database privileges, it will also possess the privileges to modify a response policy object. Now consider a scenario, where organizational policies require auditing and detection of malicious activities from all database users including those holding the DBA role. Thus, response policies must be created to respond to anomalous requests from all users. But since a DBA role holds privileges to alter any response policy, it is easy to see that the protection offered by the response system against a malicious DBA can trivially be bypassed. The fundamental problem in such administration model is that of *conflict-of-interest*. The main issue is essentially that of *insider threats*, that is, how to protect a response policy object from *malicious* modifications made by a database user that has *legitimate* access rights to the policy object.

To address this issue, we propose an administration model that is based on the well-known security principle of separation of duties (SoD). SoD is a principle whereby multi-

ple users are required in order to complete a given task. As a security principle, the primary objective of SoD is prevention of fraud (insider threats), and user generated errors. Such objective is traditionally achieved by dividing the task and its associated privileges among multiple users. However, the approach of using privilege dissemination is not applicable to our case as we assume the DBAs to possess all possible privileges in the DBMS. Our approach instead applies the technique of *threshold cryptography signatures* to achieve SoD. A DBA authorizes a policy operation, such as *create* or *drop*, by submitting a signature share on the policy. At least k signature shares are required to form a valid final signature on a policy, where k is a threshold parameter defined for each policy at the time of policy creation. The final signature is then validated either periodically or upon policy usage to detect any malicious modifications to the policies. The key idea in our approach is that a policy operation is invalid unless it has been authorized by at least k DBAs. We thus refer to our administration model as the **Joint Threshold Administration Model** (JTAM) for managing response policy objects. To the best of our knowledge, ours is the only work proposing such administration model in the context of management of DBMS objects. The three main advantages of JTAM are as follows. First, it requires no changes to the existing access control mechanisms of a DBMS for achieving SoD. Second, the final signature on a policy is non-repudiable, thus making the DBAs accountable for authorizing a policy operation. Third, and probably the most important, JTAM allows an organization to utilize *existing man-power resources* to address the problem of insider threats since it is no longer required to employ additional users as policy administrators. For more JTAM details, we refer the reader to Chapter 3 of the thesis.

1.4.3 System Architecture

The system's architecture consists of three main components: the traditional DBMS that handles the query execution, the profile creator module for collecting the training data and creating/maintaining the profiles, and the detection and response mechanisms integrated with the core DBMS functionality. These components form the new extended

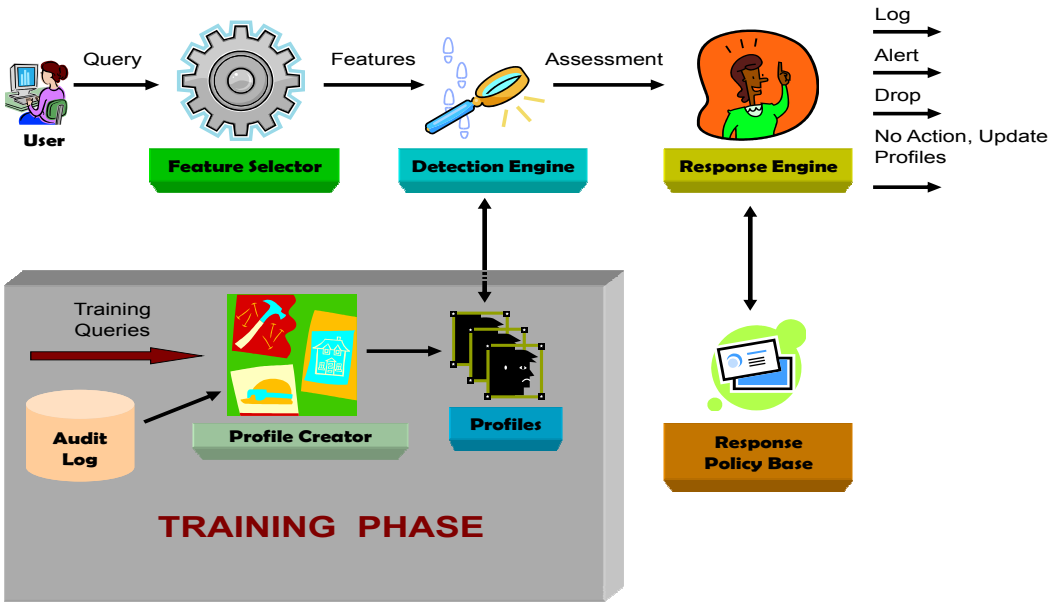


Fig. 1.1. System Architecture

DBMS that is enhanced with an independent ID system operating at the database level. The flow of interactions for the IDR process is shown in Figure 1.1. During the training phase, the SQL commands submitted to the DBMS (or read from the audit log) are analyzed by the profile creator module to create the initial profiles of the database users. For every SQL command under detection, the feature selector module extracts the features from the queries in the format expected by the detection engine. The detection engine then runs the extracted features through the detection algorithm. If an anomaly detected, the detection mechanism submits its assessment of the SQL command to the response engine according to a pre-defined interface; otherwise the command information is sent to the profile creator process for updating the profiles.

The response engine consults a policy base of existing response policies to issue a response depending on the assessment of the query submitted by the detection engine. Notice that the fact that a query is anomalous may not necessarily imply an intrusion. Other information and security policies must also be taken into account. For example, if the user logged under the role is performing some special activities to manage an emergency, the response mechanism may be instructed not to raise alarms in such circumstances. If the

response engine decides to raise an alarm, certain actions for handling the alarm can be taken. The most common action is to send an alert to the security administrator. However other actions are possible (Figure 1.1), such as log the alarm, drop the query, or even take no action at all. We have implemented a prototype of this system architecture in the PostgreSQL DBMS. We refer the reader to Chapter 5 for the implementation details and experimental results concerning the overhead of the system.

1.5 Thesis Statement and Contributions

The goal of the doctoral thesis is to develop architectures, mechanisms and algorithms for a DBMS equipped with activity monitoring, intrusion detection and response capabilities. Within this broad context, the research issues that we address are as follows:

1. Creating profiles that succinctly represent user/application-behavior interacting with a DBMS.

2. Developing efficient algorithms for online detection of anomalous database user and application behavior.

3. Developing strategies for responding to intrusions in the context of a DBMS.

4. Creating a system architecture for database intrusion detection and intrusion response as an integral component of a DBMS, and a prototype implementation of the same in the PostgreSQL DBMS [5].

1.6 Thesis Organization

The rest of the thesis document is as follows. Chapter 2 presents our approach towards detecting anomalous access patterns in a DBMS. Chapter 3 presents our approach towards the response mechanism in a DBMS. Chapter 4 presents the privilege state based access control mechanism that provides support for the fine-grained response actions. Chapter 5 presents the details of our prototype implementation of the detection and response mecha-

nism in the PostgreSQL DBMS. Chapter 6 presents an overview of related work in the area of database intrusion detection and response. We summarize the thesis in Chapter 7 with a brief overview of the future enhancements.

2. DETECTING ANOMALOUS ACCESS PATTERNS IN DATABASES

2.1 Introduction

In this chapter we present algorithms for detecting anomalous user/role access to a DBMS. The key idea underlying our approach is to build profiles of normal user behavior interacting with a database. We then use these profiles to detect *anomalous* behavior. In this context, our approach considers two different application scenarios. In the first case, we assume that the database has a Role Based Access Control (RBAC) model in place. Authorizations are specified with respect to roles and not with respect to individual users. One or more roles are assigned to each user and privileges are assigned to roles. Our ID system builds a profile for each role and is able to determine role intruders, that is, individuals that while holding a specific role deviate from the normal behavior of that role. The use of roles makes our approach usable even for databases with large a user population. Managing a few roles is much more efficient than managing many individual users. With respect to ID, using roles means that the number of profiles to build and maintain is much smaller than those one would need when considering individual users. Note that RBAC has been standardized (see the NIST model [6]) and has been adopted in various commercial DBMS products. This implies that an ID solution, based on RBAC, could be easily deployed in practice.

In the second case, we address the same problem in the context of a DBMS without any role definitions. This is a necessary case to consider because not all organizations are expected to follow a RBAC model for authorizing users of their databases. In such a setting, every SQL command is associated with the user that issued it. A naive approach for ID in this setting would be to build a different profile for every user. For systems with large user bases such an approach would be extremely inefficient. Moreover, many of the users in

those systems are not particularly active and they only occasionally submit queries to the database. In the case of highly active users, profiles would suffer from over-fitting, and in the case of inactive users, they would be too general. In the first case we would observe a high number of false alarms, while the second case would result in high number of missed alarms, that is, alarms that should have been raised. We overcome these difficulties by building user-group profiles (clusters of similar behaviors) based solely on the transactions users submit to the database. Given such profiles, we define an *anomaly* as an access pattern that deviates from the profiles.

The two problems that we address in the context of an intrusion detection mechanism specifically tailored for a DBMS are as follows: how to build and maintain profiles representing accurate and consistent user behavior; how to use these profiles for performing the ID task at hand. The solution to both problems relies on the use of 'intrusion free' database traces, that is, sequences of database audit log records representing normal user behavior[1]. However, the information contained in these traces differ depending on the application scenario in question. When role information does exist, the problem is transformed into a supervised learning problem. A classifier is trained using a set of intrusion-free training records. This classifier is then used for detecting anomalous behavior. For example, if a user claims to have a specific role while the classifier classifies her behavior as indicative of another role, then an alarm is raised. On the other hand, for the case in which no role information is available, we form our solution based on unsupervised learning techniques. We employ clustering algorithms to construct clusters of users that behave in a similar manner (with respect to their database access patterns). These clusters may also help the DBA in deciding which roles to define. For every user, we maintain the mapping to its representative cluster. For the ID phase, we specify two different approaches. In the first approach, we treat the problem in a manner similar to the supervised case with the clusters

[1]Guarantying the intrusion free nature of the training data is an issue often raised in the context of anomaly detection systems. The standard technique employed to address this concern is to use outlier detection algorithms to remove potential anomalies from the training data. Though this does not guarantee that all malicious SQL statements are removed from the training data or that every outlying point that is removed is malicious; in practice, this step has often been observed to increase the accuracy of anomaly detection systems. In this work, however, we do not employ such strategy for our experiments.

as the classifier classes. In the second approach, we treat the detection phase as an outlier detection problem. That is, an alarm is raised for a new query if it is marked as an outlier with respect to its representative cluster.

The main challenge in attacking our problem is to extract the right information from the database traces so that accurate profiles can be built. To address this problem, we propose several representations for the database log records, characterized by different granularity and, correspondingly, by different accuracy levels. By using those representations, we then address the first scenario as a classification problem and the second one as a clustering problem.

2.2 Data Representation

In order to identify user behavior, we use the database audit files for extracting information regarding users' actions. The audit records, after being processed, are used to form initial profiles representing acceptable actions. Each entry in the audit file is represented as a separate data unit; these units are then combined to form the desired profiles.

We assume that users interact with the database through commands, where each command is a different entry in the log file, structured according to the SQL language. For example, in the case of *select* queries such commands have the format:

```
SELECT [DISTINCT] {TARGET-LIST}
FROM              {RELATION-LIST}
WHERE             {QUALIFICATION}
```

In order to build profiles, we need to pre-process the log-file entries and convert them into a format that can be analyzed by our algorithms. Therefore, we represent each entry by a basic data unit that contains five fields, and thus it is called a *quiplet*.

Quiplets are our basic unit for viewing the log files and are the basic components for forming profiles. User actions are characterized using sets of such quiplets. Each quiplet contains the following information: the SQL command issued by the user, the set of relations accessed, and for each such relation, the set of referenced attributes. This in-

formation is available in the three basic components of the SQL command, namely, the `SQL COMMAND`, the `TARGET-LIST` and the `RELATION-LIST`. We also process the `QUALIFICATION` component of the query to extract information on relations and their corresponding attributes, that are used in the query predicate.[2] Therefore, the abstract form of such a quiplet consists of five fields (*SQL Command, Projection Relation Information, Projection Attribute Information, Selection Relation Information and Selection Attribute Information*)[3] . For the sake of simplicity we represent a generic quiplet using a 5-ary relation $Q(c, \mathcal{P}_\mathcal{R}, \mathcal{P}_\mathcal{A}, \mathcal{S}_\mathcal{R}, \mathcal{S}_\mathcal{A})$, where c corresponds to the command, $\mathcal{P}_\mathcal{R}$ to the projection relation information, $\mathcal{P}_\mathcal{A}$ to the projection attribute information, $\mathcal{S}_\mathcal{R}$ to the selection relation information, and $\mathcal{S}_\mathcal{A}$ to the selection attribute information. Depending on the type of quiplet the two arguments $\mathcal{P}_\mathcal{R}$(or $\mathcal{S}_\mathcal{R}$) and $\mathcal{P}_\mathcal{A}$(or $\mathcal{S}_\mathcal{A}$) can be of different types, but for simplicity and clarity we allow the symbols to be overloaded. Whenever the type of quiplet is vital, we will explicitly specify it. However, when it is not specified our claims hold for all types of quiplets.

Depending on the level of detail required in the profile construction phase and in the ID phase, we represent the quiplets from the log file entries using three different representation levels. Each level is characterized by a different amount of recorded information.

We call the most naive representation of an audit log-file record, *coarse quiplet* or *c-quiplet*. A c-quiplet records only the number of distinct relations and attributes projected and selected by the SQL query. Therefore, c-quiplets essentially model how many relations and how many attributes are accessed in total, rather than the specific elements that are accessed by the query. The c-quiplets are defined as follows:

Definition 2.2.1 *A* **coarse quiplet** *or* **c-quiplet** *is a representation of a log record of the database audit log file. Each c-quiplet consists of* 5 *fields (*`SQL-CMD, PROJ-REL-COUNTER, PROJ-ATTR-COUNTER, SEL-REL-COUNTER,`

[2]The relation and attribute information is assumed to be present in the join conditions of the predicate. We do not consider the cases of complex sub-queries that cannot be reduced to join conditions.

[3]For clarity, we only show the representation for the syntax of a *select* command. The representation is general enough to capture information from other SQL commands such as *insert, delete and update*. For example, for the *insert* command, the inserted into relation and columns are encoded as the projection relation and projection attributes.

SEL-ATTR-COUNTER*). The first field is symbolic and corresponds to the issued SQL command. The next two fields are numeric, and correspond to the number of relations and attributes involved in the projection clause of the SQL query, respectively. The last two fields are also numeric, and correspond to the number of relations and attributes involved in the selection clause of the SQL query.* □

In terms of the quiplet notation $Q()$, here both $\mathcal{P}_{\mathcal{R}}$(or $\mathcal{S}_{\mathcal{R}}$) and $\mathcal{P}_{\mathcal{A}}$(or $\mathcal{S}_{\mathcal{A}}$) correspond to the number of relations and attributes involved in the query respectively. Apparently, a large amount of valuable information in the database log is ignored by c-quiplets. It is however useful to consider such a primitive data representation since it is sufficient in the case of a small number of well-separated roles. Moreover, more sophisticated representations of log-file entries are based on the definition of c-quiplets.

The second representation scheme captures more information from the log file records. We call this representation, *medium-grain quiplet* or *m-quiplet*. These quiplets extend the coarse quiplets by further exploiting the information present in the log entries. Like a c-quiplet, a m-quiplet represents a single log entry of the database log file. In this case though, each relation is represented separately by the number of its attributes projected (or selected) by the SQL query. Thus, in terms of the quiplet notation $Q()$, $\mathcal{P}_{\mathcal{R}}$, $\mathcal{P}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{R}}$ and $\mathcal{S}_{\mathcal{A}}$ are vectors of the same size which is equal to the number of relations in the database.

The m-quiplets are defined as follows:

Definition 2.2.2 *A* **medium-grain quiplet** *or* **m-quiplet** *is a data object which corresponds to a single entry of the database log file and consists of* 5 *fields (*SQL-CMD, PROJ-REL-BIN[], PROJ-ATTR-COUNTER[],SEL-REL-BIN[], SEL-ATTR-COUNTER[]*). The first field is symbolic and corresponds to the issued SQL command, the second is a binary (bit) vector of size equal to the number of relations in the database. The bit at position* i *is set to* 1 *if the* i*-th relation is projected in the SQL query. The third field of the quiplet is a vector of size equal to the number of relations in the database. The* i*-th element of the* PROJ-ATTR-COUNTER[] *vector corresponds to the number of attributes of the* i*-th relation that are projected in the SQL query. The semantics*

of SEL-REL-BIN[] *and* SEL-ATTR-COUNTER[] *vectors are equivalent to those of* PROJ-REL-BIN[] *and* PROJ-ATTR-COUNTER[] *vectors, but the information kept in the former corresponds to the selections rather than to the projections of the SQL query.* □

Finally, we introduce a third representation level of log-file records which extracts the maximum information from the log files. We call this representation *fine quiplet* or *f-quiplet*. The structure of a f-quiplet is similar to that of a m-quiplet. In particular, the first, the second and the fourth fields of a f-quiplet are the same as the corresponding fields of the m-quiplets. The f-quiplets and m-quiplets differ only for the third and fifth fields. In the case of f-quiplets, these fields are vector of vectors and are called PROJ-BIN-ATTR[] [] and SEL-BIN-ATTR[] [] respectively. The i-th element of PROJ-BIN-ATTR[] [] is a vector corresponding to the i-th relation of the database and having size equal to the number of attributes of relation i. The i-th element of PROJ-BIN-ATTR[] [] has binary values indicating which specific attributes of relation i are projected in the SQL query. The semantics of SEL-BIN-ATTR[] [] are analogous. For f-triplets, $\mathcal{P}_{\mathcal{R}}$ and $\mathcal{S}_{\mathcal{R}}$ are vectors of size equal to the number of relations in the database while $\mathcal{P}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}}$ are vectors of the same size, but with each element i being a vector of size equal to the number of attributes in relation i. The formal definition of the f-quiplets is as follows:

Definition 2.2.3 *A* **fine quiplet** *or* **f-quiplet** *is a detailed representation of a log entry. It consists of* 5 *fields (*SQL-CMD, PROJ-REL-BIN[], PROJ-ATTR-BIN[] [], SEL-REL-BIN[], SEL-ATTR-BIN[] []*). The first field is symbolic and corresponds to the SQL command, the second is a binary vector that contains* 1 *in its* i*-th position if the* i*-th relation is projected in the SQL query. The third field is a vector of* n *vectors, where* n *is the number of relations in the database. Element* PROJ-ATTR-BIN[i] [j] *is equal to* 1 *if the SQL query projects the* j*-th attribute of the* i*-th relation; it is equal to* 0 *otherwise. Similarly, the fourth field is a binary vector that contains* 1 *in its* i*-th position if the* i*-th relation is used in the SQL query predicate. The fifth field is a vector of* n *vectors, where* n *is the number of relations in the database. Element* SEL-ATTR-BIN[i] [j] *is equal to* 1 *if the SQL query references the* j*-th attribute of the* i*-th relation in the query predicate; it is equal to* 0 *otherwise.* □

Table 2.1
Quiplet construction example

SQL Command	c-quiplet	m-quiplet	f-quiplet
Select $R_1.A_1, R_2.C_2$ From R_1, R_2 Where $R_1.B_1 = R_2.B_2$	select$< 2 >< 2 >$ $< 2 >< 2 >$	select $< 1, 1 >$ $< 1, 1 >$, $< 1, 1 >< 1, 1 >$	select $< 1, 1 >$ $< [1, 0, 0], [0, 0, 1] >$ $< 1, 1 > [0, 1, 0]$, $[0, 1, 0]$

Table 2.1 shows a SQL command corresponding to select statement and its representation according to the three different types of quiplets. In the example, a database schema consisting of two relations $R_1 = \{A_1, B_1, C_1\}$ and $R_2 = \{A_2, B_2, C_2\}$, is considered.

2.3 Role-Based Anomaly Detection

In this section, we describe our methodology when information related to the roles of users is available in the database traces. This role information allows us to address the problem at hand as a standard classification problem.

2.3.1 Classifier

We use the Naive Bayes Classifier (NBC) for the ID task in RBAC administered databases. Despite some modeling assumptions regarding attribute independence inherent to this classifier, our experiments demonstrate that it is surprisingly useful in practice. Moreover, NBC has proven to be effective in many practical applications such as text classification and medical diagnosis [7–9], and often competes with much more sophisticated learning techniques [10, 11]. The reason for the popularity of NBC is its low computational requirements for both the training and classification task. The small running time is mainly due to the attribute independence assumption. Moreover, like all probabilistic

classifiers under the Maximum Aposteriori Probability (MAP) decision rule, NBC arrives at the correct classification as long as the correct class is more probable than any other class. In other words, the overall classifier is robust to deficiencies of its underlying naive probability model. We refer the reader to the paper by Domingos et al. [7] that explains the optimality region for the NBC and discusses the reasons for its effective performance even when the attribute independence assumption does not completely hold.

We first describe the general principles of the NBC (for details see [8]) and then show how it can be applied to our setting. In supervised learning, each instance x of the data is described as a conjunction of attribute values, and the target function $f(x)$ can only take values from some finite set V. The attributes correspond to the set of observations and the elements of V are the distinct classes associated with those observations. In the classification problem, a set of training examples D_T is provided, and a new instance with attribute values $(a_1, ..., a_n)$ is given. The goal is to predict the target value, or the class, of this new instance.

The approach we describe here is to assign to this new instance the most probable class value v_{MAP}, given the attributes $(a_1, ..., a_n)$ that describe it. That is

$$v_{\text{MAP}} = \arg\max_{v_j \in V} P(v_j | a_1, a_2, ..., a_n).$$

Using *Bayes Theorem* we can rewrite the expression as

$$\begin{aligned} v_{\text{MAP}} &= \arg\max_{v_j \in V} P(v_j | a_1, a_2, ..., a_n) \\ &= \arg\max_{v_j \in V} \frac{P(a_1, a_2, ..., a_n | v_j) P(v_j)}{P(a_1, a_2, ..., a_n)} \\ &\propto \arg\max_{v_j \in V} P(a_1, a_2, ..., a_n | v_j) P(v_j). \end{aligned}$$

The last derivation is feasible because the denominator does not depend on the choice of v_j and thus, it can be omitted from the $\arg\max$ argument. Estimating $p(v_j)$ is simple since it requires just counting the frequency of v_j in the training data. However, calculating $P(a_1, a_2, ..., a_n | v_j)$ is hard when considering a large dataset and a reasonably large number

of attributes [12]. The NBC, however, is based on the simplifying assumption that the attribute values are *conditionally independent*, and thus

$$v_{\text{MAP}} \propto \arg\max_{v_j \in V} P(v_j) \prod_i P(a_i|v_j). \tag{2.1}$$

This reduces significantly the computational cost since calculating each one of the $P(a_i|v_j)$ requires only a frequency count over the tuples in the training data with class value equal to v_j.

Thus, the conditional independence assumption seems to solve the computational cost. However, there is another issue that needs to be discussed. Assume an event e occurring n_{e_j} number of times in the training dataset for a particular class v_j with size $|D_{v_j}|$. While the observed fraction ($\frac{n_{n_j}}{|D_{v_j}|}$) provides a good estimate of the probability in many cases, it provides poor estimates when n_{e_j} is very small. An obvious example is the case where $n_{e_j} = 0$. The corresponding zero probability will bias the classifier in an irreversible way, since according to equation 2.1, the zero probability when multiplied with the other probability terms will give zero as its result. To avoid this difficulty we adopt a standard Bayesian approach in estimating this probability, using the m-estimate [8]. The formal definition of m-estimate is as follows:

Definition 2.3.1 *Given a dataset D_T with size $|D_T|$ and an event e that appears n_{e_j} times in the dataset for a class v_j with size $|D_{v_j}|$ and n_e times in the entire dataset, then the* m-estimate of the probability $p_{e_j} = \frac{n_{e_j}}{|D_{v_j}|}$ *is defined to be*

$$p_{e_j}^m = \frac{n_{e_j} + m \cdot \frac{n_e}{|D_T|}}{|D_{v_j}| + m}. \tag{2.2}$$

The parameter m is a constant and is called equivalent sample size*, which determines how heavily to weight p_{e_j} relative to the observed data. If n_E is 0, then we assume that* $p_E^m = \frac{1}{|D_{v_j}|}$.

The NBC directly applies to our anomaly detection framework by considering the set of roles in the system as classes and the log-file quiplets as the observations. In what follows, we show how equation 2.1 can be applied for the three different types of quiplets.

For the case of c-quiplets the application is simple since there are five attributes $(c, \mathcal{P}_\mathcal{R}, \mathcal{P}_\mathcal{A}, \mathcal{S}_\mathcal{R}, \mathcal{S}_\mathcal{A})$ to consider namely the command, the projection relation count, the projection attribute count, the selection relation count and the selection attribute count. If $\mathcal{R}$ denotes the set of roles, the predicted role of a given observation $(c_i, \mathcal{P}_{\mathcal{R}i}, \mathcal{P}_{\mathcal{A}i}, \mathcal{S}_{\mathcal{R}i}, \mathcal{S}_{\mathcal{A}i})$ is

$$r_{\text{MAP}} = \arg\max_{r_j \in \mathcal{R}} \quad \big\{p(r_j)p(c_i|r_j)p(\mathcal{P}_{\mathcal{R}i}|r_j)p(\mathcal{P}_{\mathcal{A}i}|r_j) \; p(\mathcal{S}_{\mathcal{R}i}|r_j)p(\mathcal{S}_{\mathcal{A}i}|r_j)\big\}.$$

For m-quiplets, we again have five fields $(c, \mathcal{P}_\mathcal{R}, \mathcal{P}_\mathcal{A}, \mathcal{S}_\mathcal{R}, \mathcal{S}_\mathcal{A})$, where $\mathcal{P}_\mathcal{R}$, $\mathcal{P}_\mathcal{A}$, $\mathcal{S}_\mathcal{R}$ and $\mathcal{S}_\mathcal{A}$ are vectors of the same cardinality. Except for the command attribute c, the rest of the attributes considered in this case are from the product $\mathcal{P}_\mathcal{R}\mathcal{P}_\mathcal{A}^\mathcal{T}$ and $\mathcal{S}_\mathcal{R}\mathcal{S}_\mathcal{A}^\mathcal{T}$. Therefore there are $|\mathcal{P}_\mathcal{R} \cdot \mathcal{P}_\mathcal{A}^\mathcal{T}| + |\mathcal{S}_\mathcal{R} \cdot \mathcal{S}_\mathcal{A}^\mathcal{T}| + 1$ attributes, and Equation 2.1 can be rewritten as follows

$$r_{\text{MAP}} = \arg\max_{r_j \in \mathcal{R}} \quad \Big\{p(r_j)p(c_i|r_j) \prod_{i=1}^{N} \big\{p(\mathcal{P}_\mathcal{R}[i] \cdot {\mathcal{P}_\mathcal{A}}^T[i]|r_j) \; p(\mathcal{S}_\mathcal{R}[i] \cdot {\mathcal{S}_\mathcal{A}}^T[i]|r_j)\big\}\Big\},$$

where N is the number of relations in the DBMS.

Finally, for f-quiplets, where fields $\mathcal{P}_\mathcal{R}$,$\mathcal{S}_\mathcal{R}$ are vectors and $\mathcal{P}_\mathcal{A}$,$\mathcal{S}_\mathcal{A}$ are vectors of vectors, the corresponding equation is

$$r_{\text{MAP}} = \arg\max_{r_j \in \mathcal{R}} \quad \Big\{p(r_j)p(c_i|r_j) \prod_{i=1}^{N} \{p(\mathcal{P}_\mathcal{R}[i] \cdot \mathcal{P}_\mathcal{A}[i]|r_j) \; p(\mathcal{S}_\mathcal{R}[i] \cdot \mathcal{S}_\mathcal{A}[i]|r_j)\}\Big\}.$$

With the above definitions in place, the ID task is quite straightforward. For every new query, its r_{MAP} is predicted by the trained classifier. If this r_{MAP} is different from the original role associated with the query, an anomaly is detected. For benign queries, the classifier can be updated in a straightforward manner by increasing the frequency count of the relevant attributes.

The procedure for ID can easily be generalized for the case when a user is assigned more than one role at a time. This is because our method detects anomalies on a per query basis rather than per user basis. Hence, as long as the role associated with the query is consistent with the role predicted by the classifier, the system will not detect an anomaly.

2.3.2 Experimental Evaluation

In this section, we report results from an experimental evaluation of the proposed approach and illustrate its performance as an ID mechanism. Our experimental setting consists of experiments with both synthetic and real data sets. In our previous work [13], we had reported the performance of the three quiplet types under different modes of database access patterns. The objective of the current experimental evaluation is to assess the performance of our methods on databases deployed for real-world applications. For modeling the SQL query access patterns in a real-world deployed database, we use the general form of a *zipf* probability distribution function (pdf) that is frequently used to model non-uniform access. The zipf pdf, for a random variable X, is mathematically defined as follows:

$$Zipf(X, N, s) = \frac{1/x^s}{\Sigma_{i=1}^{N} 1/i^s},$$

where N is the number of elements and s is the parameter characterizing the distribution. Figure 2.1 shows the cumulative density function for a zipf distribution for $N = 10$ and different values of s. Suppose N here denotes the number of tables in a database schema ordered according to some criteria such as lexicographic order. Then figure 2.1 shows that, as we increase s, the probability mass accumulates towards the left half of the schema, thereby making the access pattern more and more skewed. For our experiments, we also

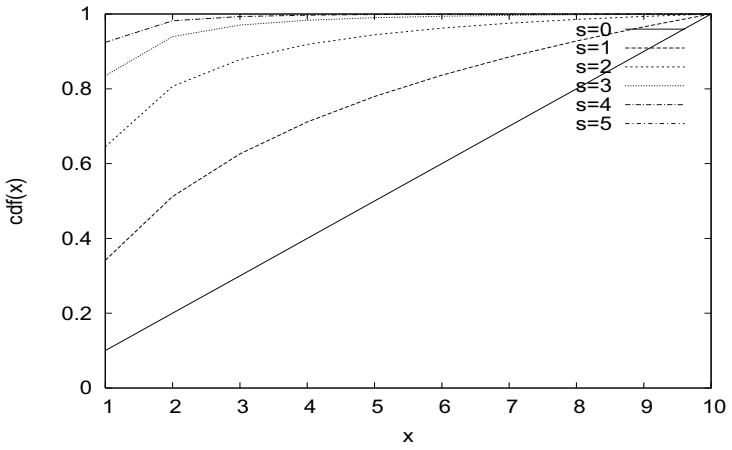


Fig. 2.1. A sample Zipf distribution for N=10

use a *reverse zipf* distribution which is a mirror image of the corresponding zipf plot with respect to a vertical axis.

Before describing our experimental findings, we give a brief outline of the generation procedure for our test datasets and anomalous queries.

Data Sets

Synthetic data sets: The synthetic data are generated according to the following model: Each role r has a probability, $p(r)$, of appearing in the log file. Additionally, for each role r the generator specifies the following five probabilities: (i) the probability of using a command c given the role, $p(c|r)$, (ii) the probability of projecting on a table t given the role and the command, $p(P_t|r, c)$, (iii) the probability of projecting an attribute within a table $a \in T$ given the role, the table and the command, $p(P_a|r, t, c)$, (iv) the probability of using a table t in the selection clause given the role and the command, $p(S_t|r, c)$ and finally, (v) the probability of using an attribute $a \in t$ in the query predicate given the role, the table and the command, $p(S_a|r, t, c)$. We use four different kinds of probability distribution functions for generating these probabilities namely, uniform, zipf, reverse zipf and multinomial.

Real data set: The real dataset used for evaluating our approach consists of 8368 SQL traces from eight different applications submitting queries to a MS SQL server database. The database schema consists of 130 tables and 1201 columns in all. The queries in this dataset consists of a mix of *select, insert and update* commands with precisely 7583 *select* commands, 213 *insert* commands and 572 *update* commands. There are no sub-queries present in any of the query predicates. Also, since role information is not available, we consider the applications themselves as our roles. For a more detailed description of the dataset we refer the reader to [14].

Anomalous query generation: We generate the anomalous query set keeping in mind the insider threat scenario. For this, we generate the anomalous queries from the same probability distribution as that of normal queries, but with role information negated. For example, if the role information associated with a normal query is 0, then we simply change the role to any role other than 0 to make the query anomalous.

2.3.3 Results

We now describe the the first synthetic dataset that we use for our experiments. The database schema consists of 100 tables and 20 columns in each tables. The number of roles for the database is 4. The SQL query submission pattern for the roles is governed by the pdf, $Zipf(N = 4, s = 1)$. The first two roles are read-only, such that they use the $select$ command with probability 1. The first role accesses the tables with a pdf, $Zipf(100, s)$, and the columns with a pdf, $Zipf(20, s)$. We vary the parameter s for our experiments. Similarly, the second role accesses the tables and columns with a pdf governed by $R\_Zipf(100, s)$ and $R\_Zipf(20, s)$, respectively. The third and the fourth roles are read-write such that they issue the *select*, *insert*, *delete* and *update* commands with probabilities $0.1, 0.1, 0.1$ and 0.7 respectively. For the *select*, *delete* and *insert* commands, these two role access all the tables and columns within each table with a uniform probability. The third role executes the *update* command with a pdf, $Zipf(100, s)$, and the fourth with a pdf, $R\_Zipf(100, s)$. We use a training data size of cardinality 5000 and set the m parameter (in equation 2.2) to

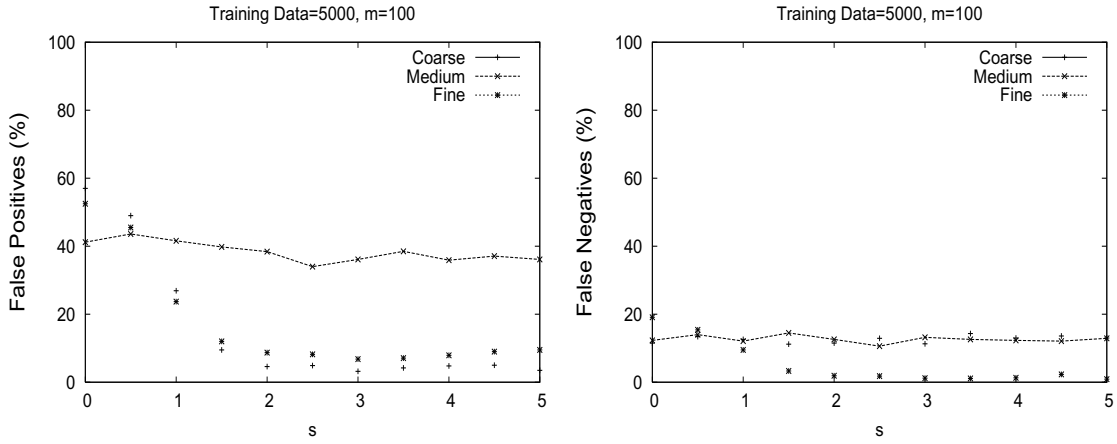


Fig. 2.2. Dataset 1: False Positive and False Negative rate

100. Figure 2.2 shows the False Positive (FP) and False Negative (FN) rates for increasing values of s. As expected, the FP and the FN rate for f-quiplet is the lowest among the three quiplet types. Also, as we make the database access becomes more skewed by increasing s, FP rate for the f-quiplet goes down.

We generate the second dataset as follows. The database schema is same as in the first dataset with 100 tables and 20 columns in each table. However, there are now 9 roles that access the database as shown in Figure 2.3. Roles 1 to 6 are read-only and roles $7, 8$ and 9 are read-write. Figure 2.4 shows the FP and FN rates for this dataset. An interesting observation is that the performance of m-quiplet is actually better than that of f-quiplet for lower values of s and comparable to f-quiplet for higher values of s. This suggests that m-quiplet may prove to be an effective replacement for f-quiplet for a DBMS with an access pattern similar to that of the second dataset.

Finally, we present experimental results for the real data set. The results are averaged over a 10-fold cross validation of the dataset. Anomalous queries are generated as described earlier. The parameter m in equation 2.2 is again set to be 100. Table 2.2 shows the performance of the three quiplet types. The FN rate for all three quiplet types is quite low. One matter of concern is the high FP rate for this dataset. This result could be due to the

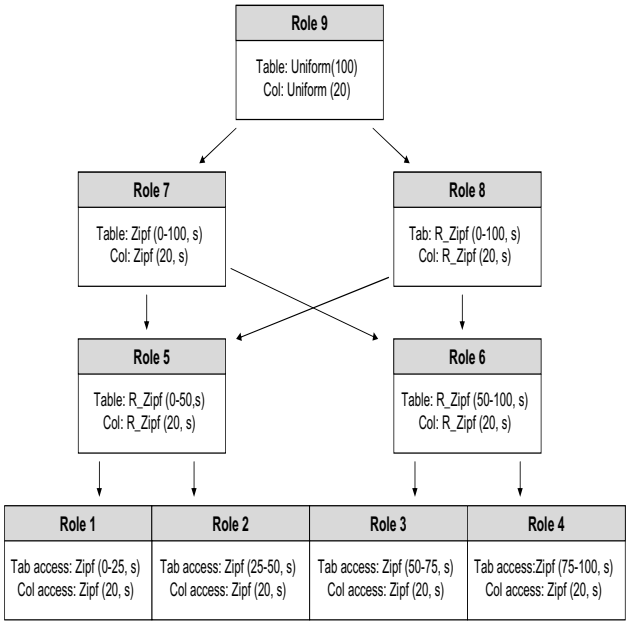


Fig. 2.3. Dataset 2: Description of Roles

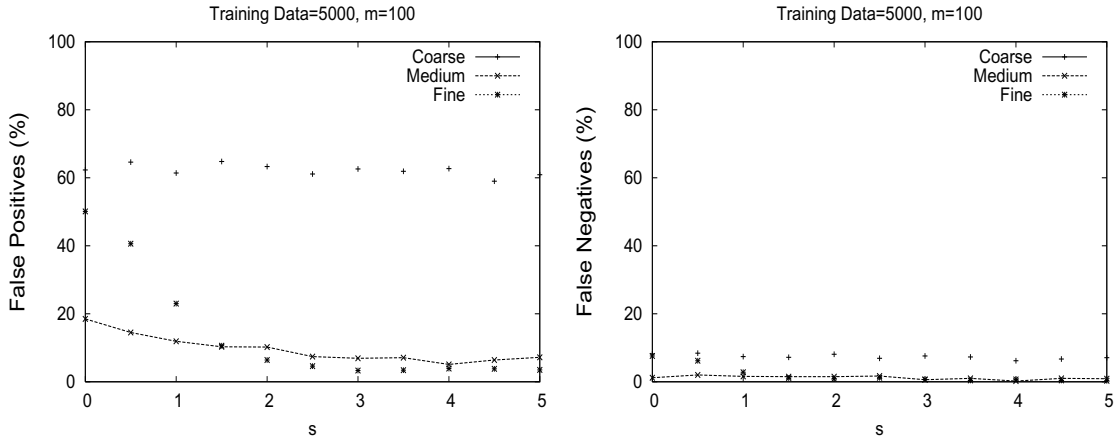


Fig. 2.4. Dataset 2: False Positive and False Negative rate

Table 2.2
Real data: False Positive and False Negative rate

Quiplet type	False Negative (%)	False Positive (%)
c	2.6	19.2
m	2.4	17.1
f	2.4	17.9

specific nature of the real dataset; or for m and f-quiplet the large number of attributes may trigger such behavior.

Overall, the experimental evaluation reveals that in most cases f-quiplet capture the access pattern of the users much better than either c or m-quiplet.

2.4 Unsupervised Anomaly Detection

We now turn our attention to the case where the role information is not available in the audit log files. In this case, the problem of forming user profiles is clearly unsupervised and thus it is treated as a clustering problem. The specific methodology that we use for the ID task is as follows: we partition the training data into clusters[4] using standard clustering techniques. We maintain a mapping for every user to its representative cluster. The representative cluster for a user is the cluster that contains the maximum number of training records for that user after the clustering phase. For every new query under observation, its representative cluster is determined by examining the user-cluster mapping. Note the assumption that every query is associated with a database user. For the detection phase, we outline two approaches. In the first approach, we apply the naive bayes classifier in a manner similar to the supervised case, to determine whether the user associated with the query belongs to its representative cluster or not. In the second approach, a statistical test is used to identify if the query is an outlier in its representative cluster. If the result of the statistical test is positive, the query is marked as an anomaly and an alarm is raised. The

[4]In the unsupervised setting, the clusters obtained after the clustering process represent the profiles.

methods we use for clustering include some standard techniques. The next section explains in detail the distance measures used for clustering. After that we briefly explain the clustering algorithms and the statistical test for detecting intruders and finally report experimental results on them.

2.4.1 Distance Functions

For clustering the quiplets into groups such that quiplets in the same group are close to each other, we need a measure to establish the "closeness" between the quiplets. For this we provide definitions of the necessary distance functions.

In order to introduce the distance functions, we first need to introduce a (generic and overloaded) function $\beta(\ )$ which will be used for evaluating and comparing quiplets of the same type. Let $Q = (c, \mathcal{P}_{\mathcal{R}}, \mathcal{P}_{\mathcal{A}}, \mathcal{S}_{\mathcal{R}}, \mathcal{S}_{\mathcal{A}})$ and $Q' = (c', \mathcal{P}'_{\mathcal{R}}, \mathcal{P}'_{\mathcal{A}}, \mathcal{S}'_{\mathcal{R}}, \mathcal{S}'_{\mathcal{A}})$ be two quiplets in general, and let $T = (\mathcal{P}_{\mathcal{R}}, \mathcal{P}_{\mathcal{A}}, \mathcal{S}_{\mathcal{R}}, \mathcal{S}_{\mathcal{A}})$ and $T' = (\mathcal{P}'_{\mathcal{R}}, \mathcal{P}'_{\mathcal{A}}, \mathcal{S}'_{\mathcal{R}}, \mathcal{S}'_{\mathcal{A}})$ denote information contained in Q and Q' respectively, minus the command c. We define, $\beta : T \times T \rightarrow \Re$ as a mapping from pairs of quiplets (minus the command c) to real numbers.

- For *c-quiplet*, function $\beta(\ )$ is calculated as follows:

$$\beta(T, T') = \sqrt{(P_R - P'_R)^2 + (P_A - P'_A)^2 + (S_R - S'_R)^2 + (S_A - S'_A)^2}$$

- For *m-quiplet*, we have:

$$\beta(T, T') = ||P_R P_A - P'_R P'_A||_2 + ||S_R S_A - S'_R S'_A||_2$$

Note that given two vectors $v_i = \{v_{i1}, v_{i2}, ..., v_{in}\}$ and $v_j = \{v_{j1}, v_{j2}, ..., v_{jn}\}$, their L_2 distance $||v_i - v_j||_2$ is defined to be $||v_i - v_j||_2 = \sqrt{\sum_{\ell=1}^{n}(v_{i\ell} - v_{j\ell})^2}$.

- For f-quiplet, function $\beta(\ )$ is calculated as follows:

$$\begin{aligned}\beta(T,T') = \\ \textstyle\sum_{i=1}^{N} \quad & \{||P_R[i]P_A[i] - P'_R[i]P'_A[i]||_2 + \\ & ||S_R[i]S_A[i] - S'_R[i]S'_A[i]||_2\}\end{aligned}$$

Observation 2.4.1 *All the above definitions of $\beta(\ )$ satisfy the triangle inequality.*

Definition 2.4.1 *The distance between two quiplets Q and Q' is defined as follows:*

$$d_Q(Q,Q') = \begin{cases} 1+\beta(T,T') \ \textit{if} \ c \neq c' \\ \beta(T,T') \ \textit{otherwise} \end{cases} \tag{2.3}$$

The following lemma states an important property of function $d_{\mathcal{T}}$.

Lemma 1 *If $\beta(\ )$ satisfies the triangle inequality, then $d_Q(\ )$ satisfies the triangle inequality.*

Proof Consider three quiplets T_1, T_2 and T_3, minus the command c. If $\beta(\ )$ satisfies the triangle inequality then the following inequality holds:

$$\beta(T_1,T_2) + \beta(T_2,T_3) \geq \beta(T_1,T_3).$$

This means that d_Q satisfies the triangle inequality as well for all the cases when $c_1 = c_2 = c_3$. Therefore we only have to re-examine the case when $c_1 \neq c_3$. Assume then that $c_1 \neq c_3$. If $c_1 = c_2$, then it should be that $c_3 \neq c_2$ and therefore the triangular inequality for d_Q is also preserved. ■

2.4.2 Clustering Algorithms

This section describes the algorithms that are used for forming the profiles in the unsupervised setting.

k-centers

The k-centers algorithm takes as input the set of data points and their distances and a parameter k, which is the desired number of clusters. The output is a flat k-clustering, that is, a single clustering consisting of k clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$. The clusters form a partitioning the input data points.

If we denote by x a data point, that is a quiplet in the log files, and by μ_j the point representing the j^{th} cluster, in the k-centers clustering algorithm we try to find the partition that optimizes the following function:

$$\max_j \max_{x \in C_j} d_{\mathcal{Q}}(x, \mu_j)$$

This problem is NP-Hard. For solving it we use the following approximate algorithm [15], also known as the *furthest-first traversal* technique. The idea is to pick any data point to start with, then choose the point furthest from it, then the point furthest from the first two[5] and so on until k points are obtained. These points are taken as cluster centers and each remaining point is then assigned to the closest center. This algorithm provides a 2-approximation guarantee for any distance function that is a metric. Given Lemma 1 that proves that $d_{\mathcal{Q}}$ is a metric, the above algorithm provides a 2-approximate solution in our setting as well.

This algorithm minimizes the largest radius of the clusters that are returned as output and uses as cluster centers, or representatives, points that are already in the data set. The advantages of this algorithm are expected to be revealed in cases in which the data set does not contain large number of outliers. That is, if the data we use for creating user profiles are free from intruders, this algorithm is expected to create profiles reasonably close to the real ones.

[5]The distance of a point x from a set S is the usual $\min\{d_{\mathcal{Q}}(x, y) : y \in S\}$.

k-means

In order to address the case where some outliers may exist in the data used to build the profiles, we also consider an alternative clustering heuristic. This is the widely used k-means algorithm. The k-means algorithm is also a member of the flat k-clustering algorithms, that output a single clustering consisting of k-clusters that partition the input points. Although, there is no proof of how good approximations we obtain using k-means, the algorithm has been widely adopted due to its low computational requirements, ease of implementation and mainly due to the fact that it works well in practice. The algorithm consists of a simple re-estimation procedure and works as follows. First, k points are chosen randomly, representing the initial cluster representatives. In this case, the representatives of the clusters correspond to the means of the data points in the cluster given the metric space. Then, the remaining data points are assigned to the closest cluster. The new representatives, subject to the last assignment, are re-computed for each cluster. The last two steps are alternated until a stopping criterion is met, that is, when there is no further change in the assignment of data points to clusters. The algorithm minimizes the following cost function:

$$\sum_{j} \sum_{x \in C_j} d_{\mathcal{Q}}(x, \mu_j)$$

where x again corresponds to a data point and μ_j is the representative of the j^{th} cluster; and in this case, it is the mean of the points in the cluster.

A significant advantage of the k-means algorithm when compared to the other clustering algorithms discussed in this section is that updates of the clusters can be executed in constant time. Consider the case in which we have already formed the k clusters on some initial set of normal input points. Now assume that new normal points arrive and we want to incorporate them into the existing clusters. Assume that x is a new point and we want to incorporate it in cluster C_i that has cardinality $|C_i|$ and is described by the mean μ_i. Then of course finding the new mean μ_i' of the cluster after the addition of point x is a trivial task, since $\mu_i' = \frac{x+\mu_i \cdot |C_i|}{|C_i|+1}$. Now our additional claim is that the error in the new cluster that

contains the points $C_i \cup \{x\}$ can also be computed in constant time. This can be executed by computing the error of each cluster by using the following formula:

$$\sum_{i=1}^{|C_i|}(x_i - \mu_i) = \frac{1}{|C_i|}\sum_{i=1}^{|C_i|} x_i^2 - (\frac{1}{|C_i|}\sum_{i=a}^{b} x_i)^2$$

Now, the error when the additional point x is added can be computed in constant time by keeping two pre-computed arrays for the cluster points: the sum of the values and the sum of squares of the values of the points appearing in the cluster.

2.4.3 Anomaly Detection Methodology

So far we have described two alternative ways of building the profiles given unclassified log quiplets. In this section, we describe our methodology in detail for identifying anomalous behavior given the set of constructed profiles.

Let z denote an issued SQL query for which our mechanism has to tell whether it is anomalous or not. The mechanism that decides whether a query is a potential intruder works as follows:

1. Find the *representative cluster* (C_z) for query z issued by user U. The representative cluster is obtained by simply looking up the user-cluster mapping created during the clustering phase.

2. We specify two different approaches for the detection phase. In the first approach, we use the naive bayes classifier in a manner similar to the supervised case by treating the clusters as the classifier classes. In the second approach, we determine if z is an outlier in cluster C_z with the help of a statistical test. If it is an outlier, we declare z as an anomaly.

In the second approach for the detection phase, we use a statistical test for deciding whether a query is an anomaly or not, but we do not specify the statistical test to use. In principle, any test appropriate for identifying outliers from a univariate data set which

cannot be mapped to a standard statistical distribution like Normal and Lognormal, is applicable. In our setting we use the *MAD* (Median of Absolute Deviations) test [16], which we describe below in brief.

MAD test: Assume to have n data points (log quiplets). Let d_i denote the distance of data point i from the cluster center it belongs to. Also, let $\overline{d}$ denote the median value of the d_i's for $i = 1, 2, \ldots, n$. Then first, we calculate the MAD as

$$\text{MAD} = \text{median}_i(|d_i - \overline{d}|).$$

Additionally, for each point i we calculate

$$Z_i = \frac{0.6745(d_i - \overline{d})}{\text{MAD}}.$$

Now if $|Z_i| > D$, then d_i is an outlier, meaning that we can infer that point i is an outlier. D is a constant which has to be experimentally evaluated. In our case, it is set to 1.5 since for this value we experience satisfactory performance of our system. We treat differently the special case where $\text{MAD} = 0$. This case is quite likely since many quiplets are expected to collide with the cluster center. In that case, we consider a point i that belongs in profile C_j as an outlier if $d_{\mathcal{Q}}(i, \mu_j) > \bar{d_{\mu_j}} + 2 \cdot \sigma_j$. In the above equation, $\bar{d_{\mu_j}}$ corresponds to the mean of the distances of all the points in cluster C_j from the representative of cluster i, namely μ_j. Likewise, σ_j corresponds to the standard deviation of those distances.

2.4.4 Experimental Evaluation

We now present the experimental results for the unsupervised case. The objective of the unsupervised case, after forming the user-cluster mapping is similar to that of the supervised case. For every new query under observation, we check if the user associated with the query is indeed a member of its mapped cluster. The dataset that we use for this evaluation is similar to the dataset 2 used in the supervised case. However, we reduce the number of tables to 20 and the number of columns per table to 10. The number of training records used for clustering are 1000. The results are averaged over 5 iterations of the clustering algorithms.

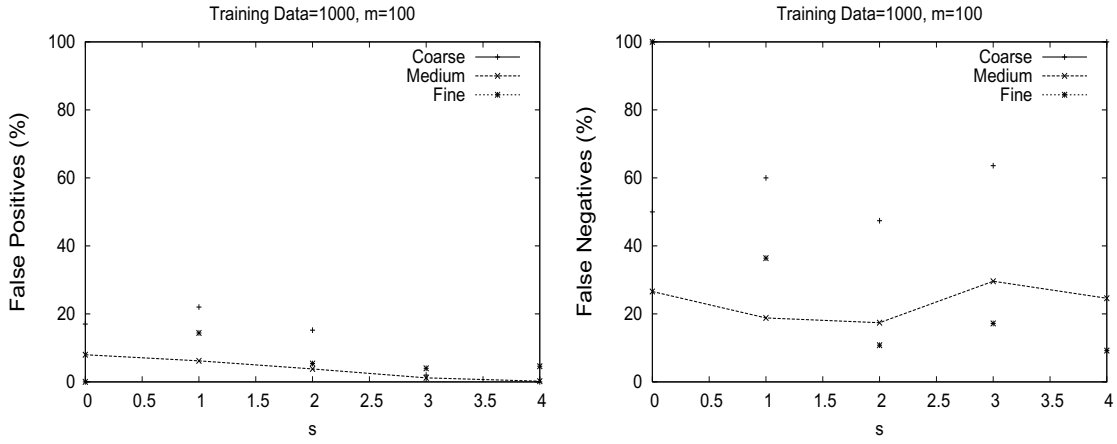


Fig. 2.5. Unsupervised Dataset: k-means - False Positive and False Negative rate for the naive bayes detection methodology

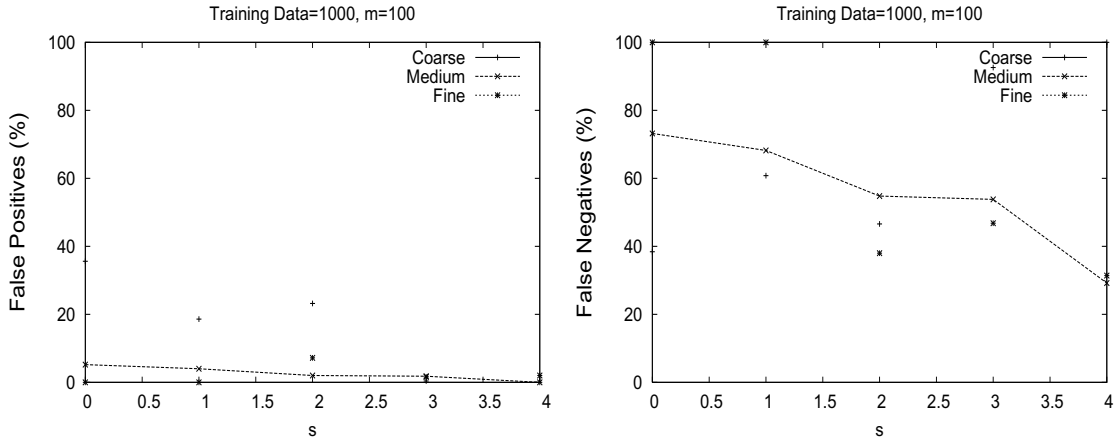


Fig. 2.6. Unsupervised Dataset: k-centers - False Positive and False Negative rate for the naive bayes detection methodology

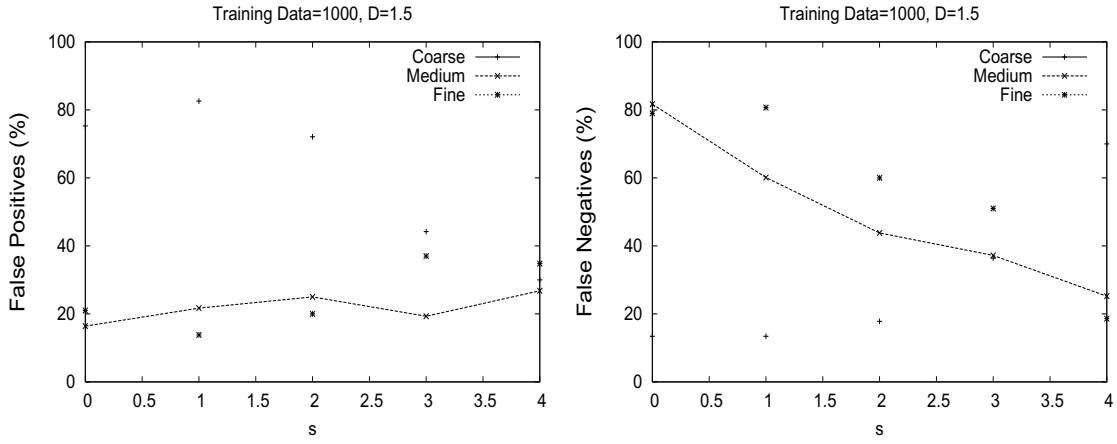


Fig. 2.7. Unsupervised Dataset: k-means - False Positive and False Negative rate for the outlier detection methodology

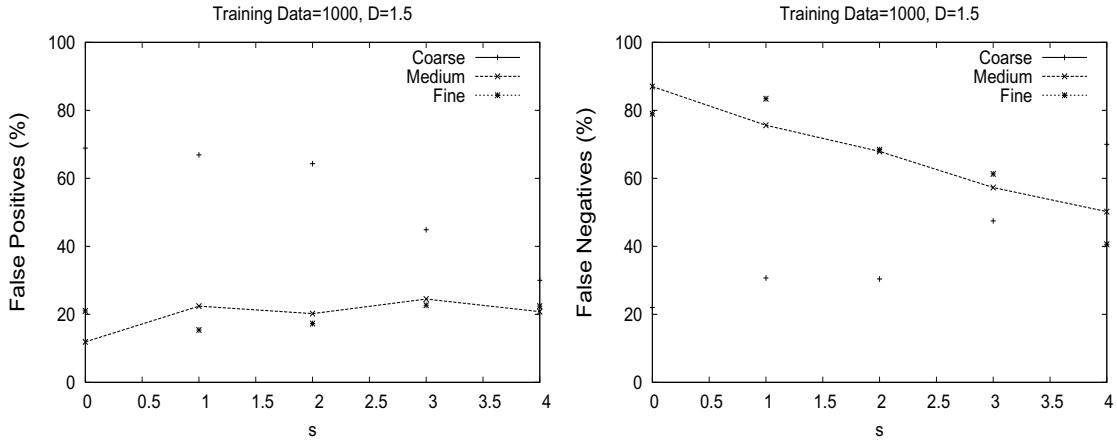
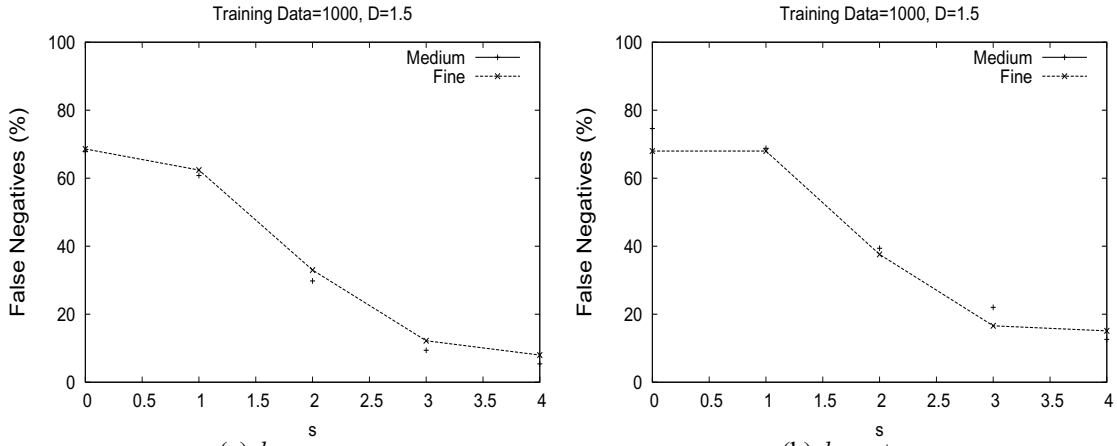


Fig. 2.8. Unsupervised Dataset: k-centers - False Positive and False Negative rate for the outlier detection methodology



(a) k-means (b) k-centers

Fig. 2.9. Unsupervised Dataset: False Negative rate for the outlier detection methodology with intrusion queries from a different probability distribution

Figures 2.5 and 2.6 show the results for the naive bayes detection methodology for both k-means and k-centers clustering algorithms. The FP rate for both the algorithms is extremely low. However, the corresponding FN rate for k-means is much better than that of k-centers. This makes k-means the algorithm of choice for the kind of dataset considered in this test case. Another noticeable observation is the better performance of m-quiplet over f-quiplet for datasets with smaller values of s.

Figures 2.7 and 2.8 report the performance of the experiments for the outlier detection methodology. The results for the outlier detection methodology are not very impressive for either of the clustering algorithms. One probable reason for this result is that anomalous queries considered in this test case come from the same probability distribution as that of the normal queries, although with role information inverted. Since they come from the same distribution, they no longer behave as outliers in the metric space and therefore, the outlier detection methodology fails to characterize most of the anomalous queries as outliers. We illustrate this with the help of Figure 2.9 which shows the FN rate for k-means and k-centers when the anomalous queries are generated from a uniform random probability distribution. For such queries, the FN rate decreases as the access pattern becomes more specific. This

shows the usefulness of the outlier detection based methodology when the access pattern of users deviate from the overall distribution of the normal access pattern.

Overall, the clustering based approach for the unsupervised case shows promising results for both m and f-quiplet. Among the clustering algorithms, the results for k-means are better than those for the k-centers algorithm. This is because k-means better captures the trend of the dataset.

2.5 Conclusion

In this Chapter, we proposed an approach for detecting anomalous access patterns in DBMS. We developed three models, of different granularity, to represent the SQL commands appearing in the database log files. We were thus able to extract useful information from the log files regarding the access patterns of the queries. When role information is available in the log records, we use it for training a classifier that is then used as the basic component for our anomaly detection mechanism. For the other case, when no role information is present in the log records, the user profiles are created using standard clustering algorithms. The anomaly detection phase is then addressed in a manner similar to the supervised case or as an outlier detection problem. Experimental results for both real and synthetic data sets showed that our methods perform reasonably well.

3. RESPONDING TO ANOMALOUS ACCESS PATTERNS IN DATABASES

The response subsystem of our IDR mechanism is in-charge of issuing a suitable response action when an anomaly is detected by the detection subsystem. It is a policy-driven mechanism in which the response policies are pre-defined by DBAs to take an action when an anomaly matches a policy. In what follows, Section 3.1 presents the details of the response policy language. Section 3.2 presents the design and implementation details of our Joint Threshold Administration Model (JTAM) for managing these policies. We discuss the policy matching algorithms, that search for policies applicable to an anomalous request in Section 3.3. Section 3.4 discusses the implementation details of our response mechanism in the PostgreSQL DBMS, and reports the experimental results concerning the overhead incurred by our techniques. We summarize this Chapter in Section 3.5.

3.1 Policy Language

The detection of an anomaly by the detection engine can be considered as a system event. The attributes of the anomaly, such as user, role, SQL command, then correspond to the environment surrounding such an event. Intuitively, a policy can be specified taking into account the anomaly attributes to guide the response engine in taking a suitable action. Keeping this in mind, we propose an **Event-Condition-Action** (ECA) language for specifying response policies. Later in this section, we extend the ECA language to support novel response semantics. ECA rules have been widely investigated in the field of active databases [17]. An ECA rule is typically organized as follows:

```
ON {Event} IF {Condition} THEN {Action}
```

Table 3.1
Anomaly Attributes

Attribute	Description
CONTEXTUAL	
User	The user associated with the request.
Role	The role associated with the request.
Client App	The client application associated with the request.
Source IP	The IP address associated with the request.
Date Time	Date/Time of the anomalous request.
STRUCTURAL	
Database	The database referred to in the request.
Schema	The schema referred to in the request.
Obj Type	The object types referred to in the request such as table, view, stored procedure
Obj(s)	The object name(s) referred in the request
SQLCmd	The SQL Command associated with the request
Obj Attr(s)	The attributes of the object(s) referred in the request.

As it is well known, its semantics is as follows: if the *event* arises and the *condition* evaluates to true, the specified *action* is executed. In our context, an event is the detection of an anomaly by the detection engine. A condition is specified on the attributes of the detected anomaly. An action is the response action executed by the engine. In what follows, we use the term ECA policy instead of the common terms ECA rules and triggers to emphasize the fact that our ECA rules specify policies driving response actions. We next discuss in detail the various components of our language for ECA policies.

3.1.1 Attributes and Conditions

Anomaly Attributes. The anomaly detection mechanism provides its assessment of the anomaly using the anomaly attributes. We have identified two main categories for such attributes. The first category, referred to as *contextual category*, includes all attributes describing the context of the anomalous request such as user, role, source, and time. The second category, referred to as *structural category*, includes all attributes conveying information about the structure of the anomalous request such as SQL command, and accessed database objects. Details concerning these attributes are reported in Table 3.1. The detection engine submits its characterization of the anomaly using the anomaly attributes. Therefore, the anomaly attributes also act as an interface for the response engine, thereby hiding the internals of the detection mechanism. Note that the list of anomaly attributes provided here is not exhaustive. Our implementation of the response system can be configured to include/exclude other user-defined anomaly attributes.

Policy Conditions. A response policy condition is a conjunction of predicates where each predicate is specified against a single anomaly attribute. Note that to minimize the overhead of the policy matching procedure (cfr. Section 3.3), we do not support disjunctions between predicates of different attributes such as SQLCmd = 'Select' OR 'IPAddress' = '10.10.21.200'. However, disjunctions between predicates of the same attribute are still supported. For example, if an administrator wants to create a policy with the condition SQLCmd = 'Select' OR SQLCmd = 'Insert'; such condition can be supported by our framework by specifying a single predicate as SQLCmd IN {'Select', 'Insert'}. More examples of such predicates are given below:

```
Role            != DBA
Source IP       IN 192.168.0.0/16
Objs            IN {dbo.*}
```

We formally define a response policy condition as follows:

Definition 3.1.1 (Policy Condition.) Let $PA = \{A_1, A_2...A_n\}$ be the set of anomaly attributes where each attribute A_i has domain T_i of values. Let a predicate Pr be defined

as Pr: $A_k\ \theta\ c$, where $A_k \in PA$, θ is a comparison operator in $\{>, <, >=, <=, =, != , like, IN, BETWEEN\}$, and c is a constant value in T_k. The condition of a response policy Pol is defined as $Pol(C)$: Pr_k and Pr_l and ... and Pr_m where $Pr_k, Pr_l \ldots Pr_m$ are predicates of type Pr.

3.1.2 Response Actions

Once a database request has been flagged off as anomalous, an action is executed by the response system to address the anomaly. The response action to be executed is specified as part of a response policy. Table 3.2 presents a taxonomy of response actions supported by our system. The *conservative* actions are low severity actions. Such actions may log the anomaly details or send an alert, but they do not pro-actively prevent an intrusion. *Aggressive* actions, on the other hand, are high severity responses. Such actions are capable of preventing an intrusion pro-actively by either dropping the request, disconnecting the user or revoking/denying the necessary privileges. *Fine-grained* response actions are neither too conservative nor too aggressive. Such actions may *suspend* or *taint* an anomalous request. A suspended request is simply put on hold, until some specific actions are executed by the user, such as the execution of further authentication steps. A tainted request is simply marked as a potential suspicious request resulting in further monitoring of the user and possibly in the suspension or dropping of subsequent requests by the same user. We refer the reader to [18] for further details on request suspension and tainting. Note that a sequence of response actions can also be specified as a valid response. For example, LOG can be executed before $ALERT$ in order to log the anomaly details as well as send a notification to the security administrator.

Table 3.3 describes two response policy examples. The threat scenario addressed by Policy 1 is as follows. In many cases, the database users and applications have read access to the system catalogs tables by default. Such access is sometimes misused during a SQL Injection attack to gather sensitive information about the DBMS structure. An anomaly detection engine will be able to catch such requests since they will not match the normal

Table 3.2
Taxonomy of Response Actions

Action	**Description**
CONSERVATIVE: low severity	
NOP	No OPeration. This option can be used to filter unwanted alarms.
LOG	The anomaly details are logged.
ALERT	A notification is sent.
FINE-GRAINED: medium severity	
TAINT	The request is audited.
SUSPEND	The request is put on hold till execution of a confirmation action.
AGGRESSIVE: high severity	
ABORT	The anomalous request is aborted.
DISCONNECT	The user session is disconnected.
REVOKE	A subset of user-privileges are revoked.
DENY	A subset of user-privileges are denied.

Table 3.3
Response Policy Examples

Policy 1 ON *ANOMALY DETECTION* IF *Role != DBA* and *Obj Type = table* and *Objs IN dbo.** and *SQLCmd IN* {*Select*} THEN *DISCONNECT*
Policy 2 ON *ANOMALY DETECTION* IF *Role = DBA* and *Source IP IN 192.168.0.0/16* and *Date Time BETWEEN* 0800 - 1700 THEN *NOP*

profile of the user. Suppose that we want to protect the DBMS from anomalous reads to the system catalogs ('dbo' schema) from unprivileged database users. Policy 1 aggressively prevents against such attacks by disconnecting the user.

Policy 2 prevents the false alarms originating from the privileged users during usual business hours. The policy is formulated to take no action on any anomaly that originates from the internal network of an organization from the privileged users during normal business hours.

3.1.3 Interactive ECA Response Policies

An ECA policy is sufficient to trigger simple response measures such as disconnecting users, dropping an anomalous request, sending an alert, and so forth. In some cases, however, we need to engage in interactions with users. For example, as described in Section 3.1.2, suppose that upon detection of an anomaly, we want to execute a fine-grained response action by suspending the anomalous request. Then we ask the user to authenticate with a second authentication factor as the next action. In case the authentication fails,

the user is disconnected. Otherwise, the request proceeds. As ECA policies are unable to support such sequence of actions, we extend them with a confirmation action construct. A *confirmation action* is the second course of action after the initial response action. Its purpose is to interact with the user to resolve the effects of the initial action. If the confirmation action is successful, the *resolution action* is executed, otherwise the *failure action* is executed[1].

Thus, a response policy in our framework can be symbolically represented as follows[2]:

```
ON          {Event}
IF          {Condition}
THEN        {Initial Action}
CONFIRM     {Confirmation Action}
ON SUCCESS  {Resolution Action}
ON FAILURE  {Failure Action}
```

An example of an interactive ECA response policy is presented in Table 3.4. The initial action is to suspend the anomalous user request. As a confirmation action, the user is prompted for re-authentication. If the confirmation action fails, the failure action is to abort the request and disconnect the user. Otherwise, no action is taken and the request is processed by the DBMS.

3.2 Policy Administration

As discussed in Chapter 1, the main issue in the administration of response policies is how to protect a policy from *malicious* modifications made by a DBA that has *legitimate* access rights to the policy object. To address this issue, we propose an administration

[1]Note that implementing the confirmation actions such as a re-authentication or a second factor of authentication require changes to the communication protocol between the database client and the server. The scenarios in which such confirmation actions may be useful are when a malicious subject (user/process) is able to bypass the initial authentication mechanism of the DBMS due to software vulnerabilities (such as buffer overflow) or due to social engineering attacks (such as using someone else's unlocked unattended terminal).

[2]Note that in case where an interactive response with the user is not required, the confirmation/resolution/-failure actions may be omitted from the policy.

Table 3.4
Interactive ECA Response Policy Example

Policy 3**:** Re-authenticate unprivileged users who are logged from inside the organization's internal network for write anomalies to tables in the dbo schema. If re-authentication fails, drop the request and disconnect the user else do nothing.

ON *ANOMALY DETECTION*
IF *Role != DBA* and *Source IP IN 192.168.0.0/16* and
Obj Type = table and *Objs IN dbo.** and
SQLCmd IN {*Insert,Update,Delete*}
THEN *SUSPEND*
CONFIRM *RE-AUTHENTICATE*
ON SUCCESS *NOP*
ON FAILURE *ABORT,DISCONNECT*

model referred to as the **Joint Threshold Administration Model (JTAM)**. The threat scenario that we assume is that a DBA has all the privileges in the DBMS, and thus it is able to execute arbitrary SQL *insert*, *update*, and *delete* commands to make malicious modifications to the policies. Such actions are possible even if the policies are stored in the system catalogs[3]. JTAM protects a response policy against malicious modifications by maintaining a digital signature on the policy definition. The signature is then validated either periodically or upon policy usage to verify the integrity of the policy definition.

One of the key assumptions in JTAM is that we do not assume the DBMS to be in possession of a secret key for verifying the integrity of policies. If the DBMS had possessed such key, it could simply create a HMAC (Hashed Message Authentication Code) of each policy using its secret key, and later use the same key to verify the integrity of the policy. However, management of such secret key is an issue since we cannot assume the key to be hidden from a malicious DBA. The fundamental premise of our approach is that we do not trust a single DBA (with the secret key) to create or manage the response policies, but the threat is mitigated if the trust (the secret key) is distributed among multiple DBAs. This is also the fundamental problem in *threshold cryptography*, that is, the problem of secure sharing of a secret. We thus base JTAM on a *threshold cryptographic signature* scheme.

Threshold Signatures: A k out of l threshold signature scheme is a protocol that allows any subset of k users out of l users to generate a valid signature, but that disallows the creation of a valid signature if fewer than k users participate in the protocol [20]. The basic paradigm of most well-known threshold signature schemes is as follows [21]. Each user U_i has a secret key share s_i corresponding to the signature key d. Each of the users U_i participating in the signature generation protocol generates a signature share that takes as input the message m (or the hash of the message) that needs to be signed, the secret key share s_i,

[3]Although it is strongly discouraged, many popular DBMSs allow DBAs to make ad-hoc updates to the system catalogs. For example, in PostgreSQL 8.3, the system catalogs can be updated by a DBA if the *rolcatupdate* column is set to 'true' in the *pg_authid* catalog [5]. In Oracle $11g$ Database, the system catalogs may be updated by users holding the SYS account [19].

and other public information. Signature shares from different users are then combined to form the final valid signature on m.

For a threshold signature scheme to be practical for JTAM, it scheme must meet the following three key requirements. First, the signature share generation procedure should be asynchronous, and the signature share combining operation should be completely non-interactive. In addition, the signature shares should be such that they can be made public without compromising the security of the secret shares. Such requirement eliminates the need for all k users to be present simultaneously to generate the final signature on a policy. Second, a single incorrect signature share should invalidate the final signature on the policy. Third, the signature verification mechanism should be very efficient to reduce the overhead on the DBMS's normal operations. All such requirements are supported by the Practical Threshold Signature scheme by Victor Shoup [20], and thus we employ such scheme in the design of JTAM. Shoup's protocol is based on RSA threshold signatures, and uses the concept of Lagrange interpolating polynomial [22] to create the final signature from the signature shares. In what follows, we present the details of Shoup's protocol in the context of administration of our response policies.

3.2.1 JTAM Set-Up

Before the response policies can be used, some security parameters are registered with the DBMS as part of a *one-time* registration phase. The details of the registration phase are as follows: The parameter l is set equal to the number of DBAs registered with the DBMS[4]. Such requirement allows any DBA to generate a valid signature share on a policy object, thereby making our approach very flexible. Shoup's scheme also requires a *trusted* dealer to generate the security parameters. This is because it relies on a special property of the RSA modulus, namely, that it must be the product of two *safe primes*. We assume the

[4]The registration of the DBAs (including assigning initial passwords) will be typically handled by a DBA itself. The security parameters needed for JTAM operations are presented as DBMS configuration options that may also be set by any DBA. The scenario that we assume here is that there are multiple administrators, each holding the DBA role, and thus having the same level of privileges. We assume that the DBAs are individually trusted to perform the administration tasks such as registration of DBAs, database configuration, etc since these tasks do not lead to the kind of *conflict-of-interest* that we address in the paper.

DBMS to be the *trusted* component that generates the security parameters [5]. For all values of k, such that $2 \leq k \leq l-1$, the DBMS generates the following parameters:

- **RSA Public-Private Keys.** The DBMS chooses p, q as two large prime numbers such that

$$p = 2p' + 1 \text{ and } q = 2q' + 1$$

where p' and q' are themselves large primes. Let $n = p * q$ be the RSA modulus. Let $m = p' * q'$. The DBMS also chooses e as the RSA public exponent such that $e > l$. Thus, the RSA public key is $PK = (n, e)$. The server also computes the private key $d \in Z$ such that $de \equiv 1 \bmod m$.

- **Secret Key Shares.** The next step is to create the secret key shares for each of the l DBAs. For this purpose, the DBMS sets $a_0 = d$ and randomly assigns a_i from $\{0, \ldots, m-1\}$ for $1 \leq i \leq k-1$. The numbers $\{a_0 \ldots a_{k-1}\}$ define the unique polynomial $p(x)$ of degree $k-1$, $p(x) = \sum_{i=0}^{i=k-1} a_i x^i$. For $1 \leq i \leq l$, the server computes the secret share, s_i, of each DBA, DBA_i, as follows:

$$s_i = p(i) \bmod m.$$

The secret shares can be stored in a *smartcard* or a *token* for every DBA, and submitted to the DBMS when required to sign a policy. The other alternative, that we implement in JTAM, is to let the DBMS store the shares in the database encrypted with keys generated out of the DBA's passwords[6]. Thus, during the registration phase, each DBA must submit its password to the DBMS for encrypting its secret key shares. Using this strategy, whenever a DBA needs to sign a policy for authorization, it submits its password which is used by the DBMS to decrypt the DBA's secret share, and use that to generate the correct signature share.

[5] In practice, only a small portion of the DBMS code base that deals with JTAM operations needs to be trusted.
[6] We use the widely used OpenPGP (RFC 4880) standard [23] to generate high-entropy keys from the passwords, then use such keys to encrypt the secret shares.

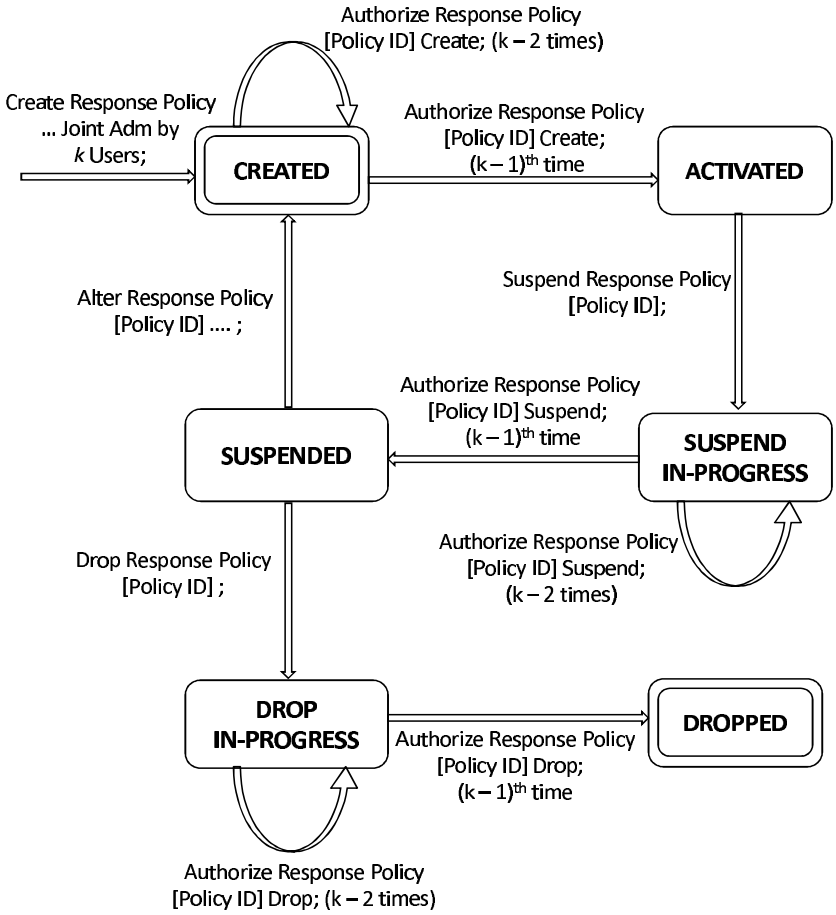


Fig. 3.1. Policy State Transition Diagram

The three key observations regarding the registration phase of JTAM are as follows. First, the security parameters, that is, the public-private key pairs, and the secret shares, need to be generated for every k value ($2 \leq k \leq l-1$), and not for every policy. This means that any policy that uses the same value of k will have the same security parameters. Second, the private key d is only used temporarily to generate the secret key shares and is not stored by the DBMS. Third, the registration phase needs to be performed as an ACID database transaction.

3.2.2 Lifecycle of a Response Policy Object

In this section, we describe the signature share generation, the signature share combining, and the final signature verification operations, in the context of the administrative lifecycle of a response policy object. The steps in the life-cycle of a policy object are policy *creation*, *activation*, *suspension*, *alteration*, and *deletion*. The life-cycle is shown in Figure 3.1 using a policy state transition diagram. The initial state of a policy object after policy creation is CREATED. After the policy has been authorized by $k-1$ administrators, the policy state is changed to ACTIVATED. A policy in an ACTIVATED state is *operational*, that is, it is considered by the policy matching procedure in its search for matching policies. If a policy needs to be altered, dropped or made non-operational, it must be moved to the SUSPENDED state. The transition from the ACTIVATED state to the SUSPENDED state must also be authorized by $k-1$ administrators, before which the policy is in the SUSPEND IN$-$PROGRESS state. Note that a policy in the SUSPEND IN$-$PROGRESS state is also considered to be operational. From the SUSPENDED state, a policy can be either moved back to the CREATED state or it can be moved to the DROPPED state. A single administrator can move a policy to the CREATED state from the SUSPENDED state, while a policy drop operation must be authorized by $k-1$ administrators (before which the policy is in the DROP IN$-$PROGRESS state). We begin our detailed discussion of a policy object's life-cycle with the policy creation procedure.

Policy Creation

The policy creation command has the following format:

Create Response Policy [*Policy Data*] Jointly Administered By k Users;

Policy Data refers to the interactive ECA response policy conditions and actions that were described in Section 3.1. Suppose that DBA_1 issues such command and that $k = 3$, and $l = 5$. DBA_1 becomes the owner of the newly created policy object. The newly created policy will be administered by 3 users (including the owner). We define an *admin-*

istrator of a policy as a user that has *owner-like* privileges on the policy object. Owner-like privileges means that the user has *all* privileges on the object along with the ability to grant these privileges to other users[7]. Note that the DBAs are assumed to possess the owner-like privileges on all database objects by default.

After the *Create Response Policy* command is issued, the DBMS performs the following operations in a sequence:

1. It prompts DBA_1 for its password.

2. It uses the password received at step 1 to decrypt the encrypted secret share of DBA_1 corresponding to the value of $k = 3$ to get s_1.

3. It generates a cryptographic hash (such as SHA1) of the policy. The hash is taken on all the policy attributes (cfr. Section 3.1) that need to be protected from malicious modifications. Thus,

$$H(Pol) = SHA1(\textit{Policy ID}, \textit{Conditions}, \textit{Initial Action(s)}, \textit{Optional Action(s)}, k, \textit{State}). \tag{3.1}$$

   Policy ID is a unique identifier generated by the DBMS for every policy. The hash is taken on the ACTIVATED policy state since that is the state of the policy after the policy has been authorized for activation by $k - 1$ administrators.

4. It creates a signature share on $H(Pol)$ using the secret share s_1 of DBA_1. Let $x = H(Pol)$. The signature share of DBA_1, is $W(DBA_1) = x^{2\Delta s_1} \in Q_n$, where $\Delta = l!$, and Q_n is the subgroup of squares in Z_n^*. $W(DBA_1)$ does not leak any information about the secret share s_1 because of the intractability of the generalized discrete logarithm problem [24].

[7]For example, SQL Server 2005 defines a CONTROL privilege for every database object that confers owner-like privileges.

Table 3.5
sys_response_policy catalog after Policy Creation

PolID	**PolData**	**k**	**r**	**hash**	**sig shares**
...	...	3	2	$H(Pol)$	$W(DBA_1)$
state			**final sig**		
CREATED					

The policy data along with the signature share and $H(Pol)$ is stored in the *sys_response_policy* system catalog as shown in Table 3.5. The column r stores the number of users that have yet to authorize the policy. The initial value of r after completion of the policy creation step is $k - 1 = 2$.

Policy Activation

Once the policy has been created, it must be authorized for activation by at least $k - 1$ administrators after which the DBMS changes the state of the policy to ACTIVATED. The policy activation command has the following format:

Authorize Response Policy [*Policy ID*] Create;

Suppose that DBA_3 issues such command. After the command is issued, the DBMS performs the following operations in a sequence:

1. It prompts DBA_3 for its password.
2. It uses the password received in step 2 to decrypt the encrypted secret share of DBA_3 corresponding to $k = 3$ to get s_3.
3. It creates a signature share on $H(Pol)$ using the secret share s_3 in a manner similar to the *Create Response Policy* command. Let $W(DBA_3)$ denote the signature share. $W(DBA_3)$ is also stored in *sys_response_policy* system catalog as shown in Table 3.6.

Table 3.6
sys_response_policy catalog after Policy Activation - I

PolID	**PolData**	**k**	**r**	**hash**	**sig shares**
...	...	3	1	$H(Pol)$	$W(DBA_1); W(DBA_3)$
state		**final sig**			
CREATED					

Table 3.7
sys_response_policy catalog after final Policy Activation

PolID	**PolData**	**k**	**r**	**hash**	**sig shares**
...	...	3	0	$H(Pol)$	
state			**final sig**		
ACTIVATED			W_{final}		

4. It decrements the value in column r by 1.

A similar procedure is adopted when another administrator, DBA_4, issues the *Authorize Response Policy* $[Policy\ ID]$ *Create* command. When $k - 1$ administrators have authorized the policy for activation, the signature combining and verification algorithms are executed (Section 3.2.2). If the final signature, W_{final}, obtained after the signature combining procedure is valid, the DBMS changes the state of the policy to ACTIVATED. The updated policy signature and state are shown in Table 3.7.

Signature Combining and Verification

Let S be the set of DBAs that have submitted the signatures shares on a policy; $S = \{i_1, \ldots, i_k\} \subset \{1, \ldots, l\}$.[8] Let $x = H(Pol) \in Z_n^*$, and $x_{i_j}^2 = W(U_{i_j})^2 = x^{4\Delta s_{i_j}}$. To combine the signature shares, we compute w such that

[8]For example, $S = \{1, 3, 4\}$ since DBA_1, DBA_3 and DBA_4 submitted the signature shares on the policy.

$$\begin{aligned} w &= x_{i_1}^{2\lambda_{0,i_1}^S} \ldots x_{i_k}^{2\lambda_{0,i_k}^S} \\ &= x^{4\Delta(\sum_{j\in S}\lambda_{0,j}^S s_{i_j})} \end{aligned}$$

where the λ's are the integers defined as follows:

$$\lambda_{i,j}^S = \Delta\ \frac{\prod_{j'\in S\setminus\{j\}}(i-j)}{\prod_{j'\in S\setminus\{j\}}(j-j')} \in Z, i \in \{0,\ldots,l\} \setminus S, j \in S.$$

These values of λ are derived from the standard Lagrange polynomial interpolation formula [22]. Using the Lagrange interpolation formula, we have

$$\Delta.f(i) \equiv \sum_{j\in S}\ \lambda_{i,j}^S\ f(j)\ mod\ m$$

Thus,

$$\begin{aligned} w^e &= x^{4\Delta(\sum_{j\in S}\lambda_{0,j}^S s_{i_j})e} \\ &= x^{4\Delta(\sum_{j\in S}\lambda_{0,j}^S f(j)\ mod\ m)e} \\ &= x^{4\Delta(\Delta f(0)e\ mod\ m)} \\ &= x^{4\Delta^2(de\ mod\ m)} \\ &= x^{e'} \end{aligned}$$

where $e' = 4\Delta^2$ since de mod $m \equiv 1$ (RSA property). Since Shoup's scheme is based on RSA threshold signatures, the final signature is the classical RSA signature [24]. To compute the final signature $W_{final} = y$ such that $y^e = x$, we set $y = w^a x^b$ where a and b are integers such that $e'a + eb = 1$. This is possible since $gcd(e, e') = 1$. The values of a and b are obtained from the standard Euclidean algorithm on e and e' [24].

The final signature, W_{final}, is verified using the public key (n, e) corresponding to the value of k. We recreate the hash of the policy, $H(Pol)$, according to Equation (3.1). If $(W_{final})^e = H(Pol)$, the signature is valid otherwise not.

Policy Suspension

To alter/drop a policy or to make it non-operational, the policy state must be changed to SUSPENDED. To change the policy state to SUSPENDED, an administrator issues the *Suspend Response Policy [Policy ID]* command. Suppose that DBA_2 issues this command. The sequence of steps followed by the DBMS upon receiving this command is as follows:

1. It prompts DBA_2 for its password.
2. It uses the password received in step 2 to decrypt the encrypted secret share of DBA_2 corresponding to $k = 3$ to get s_2.
3. It creates a signature share, $W(DBA_2)$, on $H(Pol)$ using the secret share s_2 in a manner similar to the *Create Response Policy* command; the difference in this case is that the hash, $H(Pol)$, is taken on the SUSPENDED policy state.
4. It resets the value of r to $k - 1 = 2$.
5. It changes the state of the policy to SUSPEND IN$-$PROGRESS.

Note that a policy in the SUSPEND IN-PROGRESS state is also considered to be operational. Thus, to allow for verification of the policy integrity, the final signature, W_{final}, that was obtained after the policy activation phase is left unchanged in the *sys_response_policy* catalog.

A policy in the SUSPEND IN-PROGRESS state must be authorized for suspension by at least $k - 1$ administrators by executing the *Authorize Response Policy [Policy ID] Suspend* command. The signature share generation, and the signature combining operations for such command are similar to that in the *Authorize Response Policy [Policy ID] Create* command. When $k - 1$ administrators have submitted their signature shares, the shares are combined to get the final signature, W'_{final}. The *sys_response_policy* catalog is then updated with the new final signature as shown in Table 3.8.

Table 3.8
sys_response_policy catalog after final authorization of Policy Suspension

PolID	**PolData**	**k**	**r**	**hash**	**sig shares**
...	...	3	0	$H(Pol)$	
state			**final sig**		
SUSPENDED			W'_{final}		

Policy Alteration

An administrator can alter a policy in the SUSPENDED state by executing the *Alter Response Policy [Policy ID] [Policy Data]* command. Upon receiving such command, the DBMS, creates a new hash, $H(Pol)$, on the policy according to Equation (3.1) (with state set as ACTIVATED), generates a signature share on $H(Pol)$ (for the administrator who has issued the command), clears the existing final signature from the *sys_response_policy* catalog, and changes the policy state to CREATED. The policy activation procedure must now be repeated for activating the policy.

Policy Drop

A response policy is dropped by executing the *Drop Response Policy [Policy ID]* command. The sequence of steps performed to drop a policy is similar to the steps performed for policy suspension; the difference in this case is that the hash, $H(Pol)$, in Equation (3.1) is taken on the DROPPED policy state. Also, the final signature, W'_{final}, obtained after the policy suspension phase is left unchanged when the policy state is DROP-IN PROGRESS. After the policy drop has been authorized by $k-1$ administrators, a new final signature, W''_{final}, is obtained and stored in the *sys_response_policy* catalog. The DROPPED state is the final state in the lifecycle of a policy, that is, a policy can not be re-activated after it has been dropped.

3.2.3 Attacks and Protection

In this section, we describe possible attacks on JTAM and strategies to protect from them. Recall that the threat scenario that we address is that a DBA has all the privileges in the DBMS, and thus it is able to execute arbitrary SQL commands on the *sys_response_policy* catalog.

Signature share verification. It is possible for a malicious administrator to replace a valid

signature share with some other signature share that is generated on a different policy definition. However, such attack will fail as the final signature that is produced by the signature share combining algorithm will not be valid. Note that by submitting an invalid signature share, a malicious administrator can block the creation of a valid policy. We do not see this as a major problem since the threat scenario that we address is malicious modifications to existing policies, and not generation of policies themselves.

Final signature verification. A final signature on a policy is present in all the policy states except the CREATED state. As described earlier, the final signature is verified using the public key (n, e) corresponding to the value of k. The public key is assumed to be signed using a *trusted third party* certificate that can not be forged. Thus, if a malicious DBA replaces the server generated public key, the final signature will be invalidated. Apart from verifying the final signature immediately after policy activation, suspension, and drop, the signature must also be verified before a policy may be considered in the policy matching procedure. Such strategy ensures that only the set of response policies, that have not been tampered, are considered for responding to an anomaly. Note that RSA signature verification requires modular exponentiation of the exponent e [25]. The overhead to carry out such modular exponentiation increases with the number of bits set to one in the exponent e. As we show later in our experiments, an appropriate choice of e, such as 3, 17, or 65537 can lead to a very low signature verification overhead. However, the cumulative overhead of verifying signatures on every policy during the policy matching procedure may be high. An alternative strategy is to create a dedicated DBMS process that periodically polls the *sys_response_policy* table, and verifies the signature on all policies.

Malicious Policy Update. A policy may be modified by a malicious DBA using the SQL update statement. However, all policy definition attributes that need to be protected (see Equation (3.1)) are hashed and signed; therefore any modification to such attributes through the SQL update command will invalidate the final signature on the policy.

Table 3.9
sys_response_policy_count catalog after Policy Creation

PolID	**k**	**state**	**sig**
...	k	*INVALID*	$W'(DBA_1)$

Malicious Policy Deletion. An authorized policy may be deleted by a malicious DBA using the SQL delete command. However in JTAM, a policy tuple is never physically deleted; only its state is changed to DELETED. Thus, a signature on the policy-count can be used to detect malicious deletion of policy tuples. The detailed approach is as follows: When the *Create Response Policy* command is executed, the DBMS counts the number of policy tuples present in the database. It increments such policy-count by 1 to account for the new policy being created. A hash is taken on the new policy-count and state $=$ VALID, and a signature share is generated on such hash. The signature share, policy id of the policy being created, the k value of the policy being created, and the initial state $=$ INVALID are all stored in the *sys_response_policy_count* catalog as shown in Table 3.9. These values replace the tuple that is already present in the table. Note that the policy id that is inserted in the *sys_response_policy_count* table represents the latest policy that has been created. During policy activation, the DBMS first checks if the policy id present in *sys_response_policy_count* matches the id of the policy currently being activated. If the check succeeds, it counts the number of policy tuples in the database, and generates a signature share on the hash of the policy-count, and state $=$ VALID. If the check fails, no signature share is generated. Such strategy ensures that always the correct policy-count is signed as multiple policies may be in CREATED stage at the same time. The final signature on the policy-count is generated when the $k - 1^{th}$ administrator activates the policy. The state of the policy-count signature is then changed to VALID. The dedicated DBMS process that verifies the individual policy signatures also verifies the signature on the policy-count. If a policy tuple is deleted, the signature on the policy-count is invalidated.

Signature Replay Attacks. A malicious DBA can create a copy of the final signature on a policy. It can then replay the copied signature, that is, it can replace the existing signature on the policy with the copied signature and change the policy state to the state corresponding to the copied signature. This attack is possible since we allow alterations to an existing policy object. To address this attack, we duplicate the policy state to a system column of the *sys_response_policy* catalog. A system column of a table is a column that is managed solely by the DBMS and its contents can not be modified by *any* user. In case the policy state in the system column does not match the policy state in the column visible to the user, a policy integrity violation is detected.

Policy Replay Attacks. A malicious DBA may insert a previously authorized policy tuple, whose state has been changed to DROPPED, into the *sys_response_policy* catalog. Such attack can be prevented as follows. There is a unique policy id associated with each policy tuple that is generated by the DBMS. If a malicious DBA tries to insert a previously authorized policy tuple, the DBMS will generate a fresh policy id for the new tuple. Since the hash of the policy, $H(Pol)$, takes into account the policy id, the final signature on such maliciously inserted policy tuple will be invalidated. In addition, since policy tuples are not physically deleted, the policy id generated by the DBMS is guaranteed to be unique.

3.3 Policy Matching

In this section, we present our algorithms for finding the set of policies matching an anomaly. Such search is executed by matching the attributes of the anomaly assessment with the conditions in the policies. We first state the policy matching problem formally:

Policy Matching Problem: Let $AA = \{A_1, A_2, \ldots, A_n\}$ be the set of anomaly attributes. Let $POL = \{Pol_1, Pol_2, \ldots, Pol_k\}$ be the set of response policies.
Let $PR = \{Pr_1, Pr_2, \ldots, Pr_m\}$ be the set of all distinct policy predicates. Let $Pol_i(C)$ be the policy condition for a policy Pol_i (cfr. Definition 3.1.1). Let $AAS : A_1 = a_1, A_2 =$

$a_2, \ldots, A_n = a_n$ be the assessment of an anomaly submitted by the detection mechanism to the response system. A policy Pol_i is said to match AAS if $Pol_i(C) = true$ evaluated over AAS. The policy matching problem is to find the set of all policies in POL that match a given anomaly assessment AAS.

We first present details of our approach towards policy storage in the DBMS. The policies are stored in the system catalog tables; the main reason is that the PostgreSQL DBMS maintains a cache of the catalog tables in its buffer pool. Assume a policy database consisting of 4 anomaly attributes, 6 policy predicates and 4 policies as shown in Table 3.10. The graph shown in Figure 3.2 conceptually describes how the policy cache is maintained. The graph contains three types of nodes: *attribute* nodes, *predicate* nodes, and *policy* nodes. A special start node is also added (denoted by s in Figure 3.2) to the graph which is connected to all the attribute nodes. There is an edge from attribute node A_i to a predicate node Pr_j if Pr_j is a predicate defined on A_i in the policy database. In addition, there is an edge from a predicate node Pr_j to a policy node Pol_k if Pr_j appears in the policy condition $Pol_k(C)$ of policy Pol_k in the policy database. This graph is used by the policy matching algorithms to get a list of all the predicates defined on an attribute, all the predicates belonging to a policy, and all the policies that a predicate belongs to.

We now present details of our approach towards policy matching. There are two variations of our policy matching algorithm. The first algorithm, called the *Base Policy Matching* algorithm, is described next.

3.3.1 Base Policy Matching

The policy matching algorithm is invoked when the response engine receives an anomaly detection assessment. For every attribute A in the anomaly assessment, the algorithm evaluates the predicates defined on A. After evaluating a predicate, the algorithm visits all the policy nodes connected to the evaluated predicate node. If the predicate evaluates to true, the algorithm increments the *predicate-match-count* of the connected policy nodes by 1. A policy is matched when its *predicate-match-count* becomes equal to the number of pred-

Table 3.10
Example Policy Database

Anomaly Attributes
A_1 = Source IP, A_2 = SQLCmd, A_3 = Role, A_4 = User
Policy Predicates
Pr_1: Source IP IN $192.168.0.0/16$
Pr_2: Source IP IN $128.10.0.0/16$
Pr_3: SQLCmd IN {Insert, Delete, Update}
Pr_4: SQLCmd = 'exec'
Pr_5: Role ! = 'DBA'
Pr_6: User = 'appuser'
Policy Conditions
Pol_1(C) = Pr_1 ^ Pr_3
Pol_2(C) = Pr_2 ^ Pr_6
Pol_3(C) = Pr_4 ^ Pr_5
Pol_4(C) = Pr_1 ^ Pr_3 ^ Pr_6

icates in the policy condition. On the other hand, if the predicate evaluates to false, the algorithm marks the connected policy nodes as *invalidated*. For every invalidated policy, the algorithm decrements the *policy-match-count*[9] of the connected predicates; the rationale is that a predicate need not be evaluated if its *policy-match-count* reaches zero.

3.3.2 Ordered Policy Matching

The search procedure in the base policy matching algorithm does not go through the predicates according to a fixed order. We introduce a heuristic by which the predicates are evaluated in descending order of their *policy-count*; the *policy-count* of a predicate

[9]The *policy-match-count* of a predicate is the number of non-invalidated policies that the predicate belongs to.

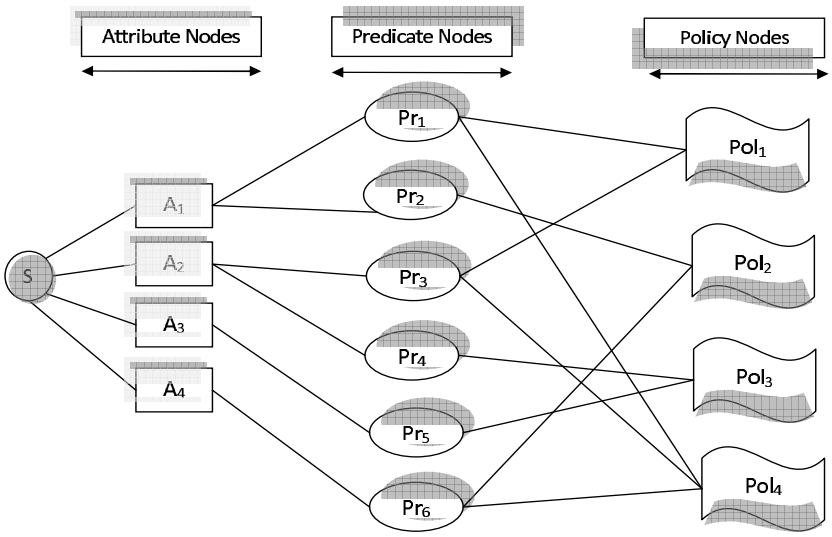


Fig. 3.2. Policy Predicate Graph Example

being the number of policies that the predicate belongs to. We refer to such heuristic as the *Ordered Policy Matching* algorithm. The rational behind the ordered policy matching algorithm is that choosing the correct order of predicates is important as it may lead to an early termination of the policy search procedure either by invalidating all the policies or by exhausting all the predicates. Note that the sorting of the predicates in decreasing order of their policy-count is a pre-computation step which is not performed during the run-time of the policy matching procedure.

3.3.3 Response Action Selection

In the event of multiple policies matching an anomaly, we must provide for a resolution scheme to determine the response to be issued. We propose the following two rank-based selection options that are based on the severity level of the response actions:

1. **Most Severe Policy (MSP).** The severity level of a response policy is determined by the highest severity level of its response action. This strategy selects the most severe policy from the set of matching policies. Note that the response actions described in Section 3.1.2 are categorized according to their severity levels. Also, in the case of interactive ECA response policies, the severity of the policy is taken as the severity level of the Failure Action.

2. **Least Severe Policy (LSP).** This strategy, unlike the MSP strategy, selects the least severe policy.

In our implementation, we provide the DBA with an option to switch between the two choices.

3.4 Implementation and Experiments

We have extended the PostgreSQL 8.3 open-source DBMS [5] with our intrusion response mechanism. We have introduced new commands in PostgreSQL for creation, activation, suspension, and dropping of response policies. We have also added six new system

Table 3.11
Response Policy System Catalogs

System Catalog	Purpose
$pg\_rpolicy\_actions$	Stores the response action definitions.
$pg\_rpolicy\_attrs$	Stores the anomaly attribute definitions.
$pg\_rpolicy\_preds$	Stores the predicate definitions.
$pg\_rpolicy\_def$	Stores the association of policies with response actions.
$pg\_rpolicy\_policypreds$	Stores the association of policies with predicates.
$pg\_rpolicy\_shares$	Stores the JTAM security parameters.
$pg\_rpolicy\_adm$	Stores the policy administration data.

catalog tables that store the response policy data. The catalogs and their purposes are described in Table 3.11. We have instrumented the query processing sub-system of PostgreSQL with our anomaly detection and response mechanism. A user request, after being parsed, passes through the detection mechanism. The policy matching procedure is invoked for every request that is detected as anomalous. We then apply the MSP or the LSP option to choose a single policy out of the set of policies returned by the policy matching algorithm.

3.4.1 Experimental Evaluation

The goal of the experimental evaluation is to measure the overhead incurred by the base policy matching, and the ordered policy matching algorithms. We also report experimental results on the overhead of the signature verification scheme in JTAM. In what follows, we first describe the experimental set-up, and then report the evaluation results.

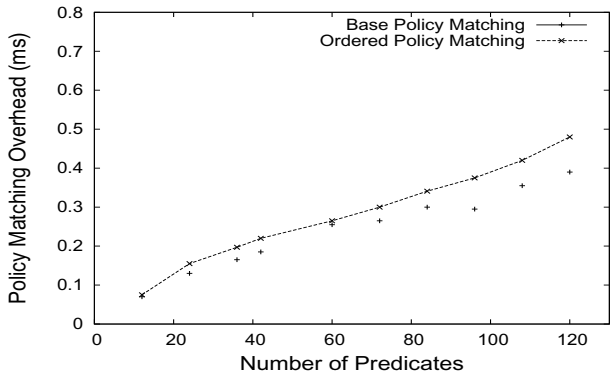


Fig. 3.3. Experiment 1: Number of Predicates vs Policy Matching Overhead

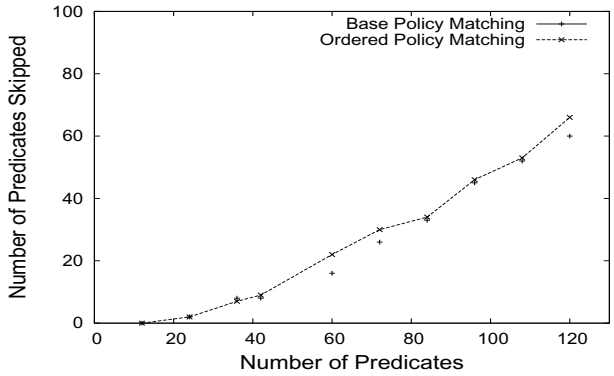


Fig. 3.4. Experiment 1: Number of Predicates vs Number of Predicates Skipped

Set-Up

We use the following six anomaly attributes for our experimental evaluation: *User*, *Client App*, *Source IP*, *Database*, *Objs*, and *SQLCmd* (see Table 3.1). The predicate generation code randomly assigns set-valued data to these anomaly attributes to create the policy predicates. The policy generation code randomly assigns such predicates to policy conditions to create the policies.

The experiments were conducted on a Intel(R) Core(TM)2 Duo CPU @ 2.33Ghz machine with 4GB of RAM. The operating system was OpenSuse 10.3.

Results

We perform three sets of experiments. The first two experiments report and compare the overhead of the policy matching algorithms. The third experiment reports results on the overhead of the signature verification mechanism in JTAM.

In the first experiment, the anomaly assessment is set such that the number of matching policies for an anomaly is kept constant at 4. The number of predicates, and correspondingly the number of policies, are varied in order to assess the policy matching overhead time. Figure 3.3 shows the policy matching overhead for the two algorithms as a function of the number of predicates. Figure 3.4 reports the number of predicates skipped as a function of the number of predicates. As expected, the policy matching overhead time increases linearly with the increase in the number of predicates in the policy database. Interestingly, the number of predicates skipped in both the algorithms is almost same. Thus, counter-intuitively, the ordered policy matching algorithm does not lead to a decrease in the number of predicate evaluations. In fact, for larger number of predicates, the policy matching overhead of the ordered predicate algorithm is higher than that of the base policy matching algorithm. Such increase in matching overhead may be explained by the fact that the predicates evaluated by the ordered policy matching are more computationally expensive than the ones evaluated by the base policy matching algorithm. The key observation from this experiment, however, is that predicate ordering based on the *policy-count* parameter has no benefits in terms of decreasing the overhead of the policy matching procedure.

In the second experiment, we keep the number of predicates in the policy database constant at 60. The number of policies is also kept constant at 20. The number of matching policies is varied in order to assess the policy matching overhead. Figure 3.5 shows the policy matching overhead for the two algorithms as a function of the number of matching policies. As expected, the policy matching overhead increases with the increase in the number of matching policies. Moreover, in this experiment as well, the overhead of the ordered policy matching algorithm is higher than that of the base policy matching algorithm. Figure 3.6 reports the variation in the number of predicates skipped by varying the number

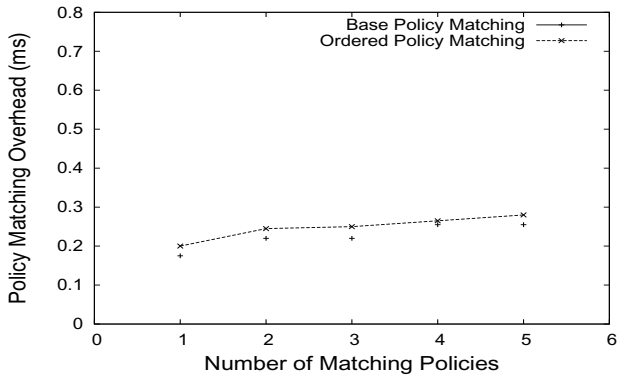


Fig. 3.5. Experiment 2: Number of Matching Policies vs Policy Matching Overhead

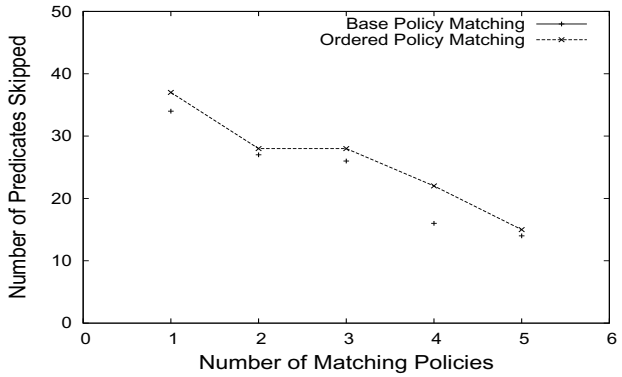


Fig. 3.6. Experiment 2: Number of Matching Policies vs Number of Predicates Skipped

of matching policies. For both the algorithms, the number of predicates skipped by the search procedure decreases for increasing numbers of matching policies. Such result is expected since an increase in the number of matching policies leads to an increasing number of predicate evaluations.

Overall, the fist two experiments confirm the low overhead associated with our policy matching algorithms. They also show that predicate ordering based on the descending *policy-count* parameter has no significant impact on reducing the overhead of the policy matching procedure.

We now report results on the overhead of the signature verification scheme in JTAM. For this experiment, we set $k = 2$, $l = 5$, and $e = 17$. The size of the RSA modulus, n, is set to 1024 bits. For such set-up, the signature verification overhead for a single pol-

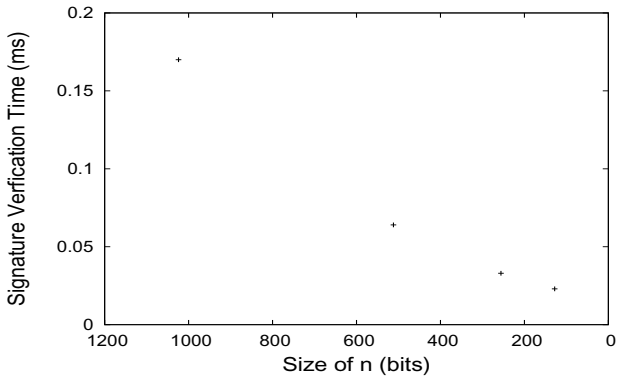


Fig. 3.7. Size of n (bits) vs Signature Verification Overhead for a single policy

icy is approximately 0.17 ms. Such overhead value confirms the low computational complexity associated with the RSA signature verification scheme. However, as mentioned in Section 3.2.3, the cumulative overhead of verifying the signatures on every policy during policy matching may be high. One approach to reduce the signature verification overhead is by decreasing the size of n (see Figure 3.7). Such strategy, however, is not recommended since 1024 bits is the recommended size of n to ensure *sufficient* security of the RSA algorithm. Therefore, a better strategy is to create a dedicated DBMS process that periodically polls the policy tables, and verifies the signature on all the policies.

3.5 Conclusion

In this Chapter, we described the response component of our intrusion detection and response system for a DBMS. The response component is responsible for issuing a suitable response to an anomalous user request. We proposed the notion of database response policies for specifying appropriate response actions. We presented an interactive Event-Condition-Action type response policy language that makes it very easy for the database security administrator to specify appropriate response actions for different circumstances depending upon the nature of the anomalous request. The two main issues that we addressed in the context of such response policies are *policy matching*, and *policy administration*. For the policy matching procedure, we described algorithms to efficiently search the policy database for policies matching an anomalous request assessment. We extended the PostgreSQL open-source DBMS to implement our methods. Specifically, we added support for new system catalogs to hold policy related data, implemented new SQL commands for the policy administration tasks, and integrated the policy matching code with the query processing subsystem of PostgreSQL. The experimental evaluation of our policy matching algorithms showed that our techniques are efficient. The other issue that we addressed is the administration of response policies to prevent malicious modifications to policy objects from legitimate users. We proposed a *Joint Threshold Administration Model* (JTAM), a novel administration model, based on Shoup's threshold cryptographic signature

scheme. We presented the design and the implementation details of JTAM, and reported experimental results on the efficiency of the policy signature verification mechanism.

4. PRIVILEGE STATE BASED ACCESS CONTROL FOR FINE GRAINED INTRUSION RESPONSE

4.1 Introduction

An access control mechanism is typically based on the notion of authorizations. An authorization is traditionally characterized by a three-element tuple of the form $< A, R, P >$ where A is the set of permissible actions, R is the set of protected resources, and P is the set of principals. When a principal tries to access a protected resource, the access control mechanism checks the rights (or privileges) of the principal against the set of authorizations in order to decide whether to allow or deny the access request.

The main goal of the work described in this Chapter is to extend the decision semantics of an access control system beyond the *all-or-nothing* allow or deny decisions. Specifically, we provide support for more *fine-grained* decisions of the following two forms: *suspend*, wherein further negotiation (such as a second factor of authentication) occurs with the principal before deciding to allow or deny the request, and *taint*, that allows one to audit the request in-progress, thus resulting in further monitoring of the principal, and possibly in the suspension or dropping of subsequent requests by the same principal. The main motivation for proposing such fine-grained access check decisions is to provide system support for extending the response action semantics of an application level anomaly detection (AD) system that detects the anomalous patterns of requests submitted to it.

Most AD systems, in the event of detecting an anomaly, would either log the anomalous request and allow it to proceed or block the request. We want to extend such responses with actions like request suspension (supported by the *suspend* decision semantics) and request tainting (supported by the *taint* decision semantics). Why do we extend the access control mechanism to support such response actions? Certainly, such responses may also be issued by an AD mechanism working independently of the underlying access control system. The

usefulness of our approach is evident from the following scenario. Suppose we model a user request as the usage of a set of *privileges* in the system where a privilege is defined as an operation on a resource. For example, the SQL query '*SELECT * FROM orders, parts*' is modeled as using the privileges {select,orders} and {select,parts} in the context of a database management system (DBMS). After detecting such request as anomalous (using any anomaly detection algorithm), consider that we want to re-authenticate the user and drop the request in case the re-authentication procedure fails. Suppose that every time a similar request is detected to be anomalous, we want the same re-authentication procedure to be repeated. If our response mechanism does not *remember* the requests, then the request will always undergo the detection procedure, detected to be anomalous and then submitted to the response mechanism to trigger the re-authentication procedure. A more generic and flexible approach for achieving such response semantics is to *attach a suspend state to the privileges* associated with the anomalous request. Then for every subsequent similar request (that uses the same set of privileges as the earlier request detected to be anomalous), the semantics of the privilege in the *suspend* state automatically triggers the re-authentication sequence of actions for the request under consideration without the request being subjected to the detection mechanism. Moreover, if the system is set-up such that the request is always subjected to the detection mechanism (in case access control enforcement is performed after the intrusion detection task), more advanced response logic can be built based on the fact that a request is detected to be anomalous whose privileges are already in the *suspend* state.

In addition to supporting fine-grained intrusion response, manually moving a privilege to the *suspend* state (using administrative commands) provides the basis for an event based continuous authentication mechanism. Similar arguments can be made for attaching the *taint* state to a privilege that triggers auditing of the request in progress. Since we extend the decision semantics of our access control system using privilege states, we call it a *privilege state based access control* (PSAC) system. For the completeness of the access control decisions, a privilege, assigned to a user or role, in PSAC can exist in the following five states: *unassign*, *grant*, *taint*, *suspend*, and *deny*. The privilege states, the state transition

semantics and a formal model of PSAC are described in detail in Section 4.2. Note that the PSAC model that we present in Section 4.2 is flexible enough to allow more that the above mentioned five states.

We have developed PSAC in the context of a role based access control (RBAC) system [26]. Extending PSAC with roles presents the main challenge of *state conflict resolution*, that is, deciding on the final state of a privilege when a principal receives the same privilege in different states from other principals. Moreover, additional complexity is introduced when the roles are arranged in a hierarchy where the roles higher-up in the hierarchy inherit the privileges of the lower level roles. We present precise semantics in PSAC to deal with such scenarios.

The main contributions of this work can be summarized as follows:

1. We present the design details, and a formal model of PSAC in the context of a DBMS.
2. We extend the PSAC semantics to take into account a role hierarchy.
3. We implement PSAC in the PostgreSQL DBMS [5] and discuss relevant design issues.
4. We conduct an experimental evaluation of the access control enforcement overhead introduced by the maintenance of privilege states in PSAC, and show that our implementation design is very efficient.

The rest of the Chapter is organized as follows. Section 4.2 presents the details of PSAC and its formal model; it also discusses how a role hierarchy is supported. Section 4.3 presents the details of the system implemented in PostgreSQL, and the experimental results concerning the overhead introduced by the privilege states on the access control functions. We conclude the paper in Section 4.4.

4.2 PSAC Design and Formal Model

In this section, we introduce the design and the formal model underlying PSAC. We assume that the authorization model also supports roles, in that RBAC is widely used by

Table 4.1
Privilege States

State	Access Check Result Semantics
unassign	The access to the resource is not granted.
grant	The access to the resource is granted.
taint	The access to the resource is granted; the system audits access to the resource.
suspend	The access to the resource is not granted until further negotiation with the principal is satisfied.
deny	The access to the resource is not granted.

access control systems of current DBMSs [27–29]. In what follows, we first introduce the privilege state semantics and state transitions. We then discuss in detail how those notions have to be extended when dealing with role hierarchies.

4.2.1 Privilege States Dominance Relationship

PSAC supports five different privilege states that are listed in Table 4.1. For each state, the table describes the semantics in terms of the result of an access check.

A privilege in the *unassign* state is equivalent to the privilege not being assigned to a principal; and a privilege in the *grant* state is equivalent to the privilege being granted to a principal. We include the *deny* state in our model to support the concept of negative authorizations in which a privilege is specifically denied to a principal [30]. The *suspend* and the *taint* states support the fine-grained decision semantics for the result of an access check.

In most DBMSs, there are two distinct ways according to which a user/role[1] can obtain a privilege p on a database object o:

1. **Role-assignment**: the user/role is assigned a role that has been assigned p;

[1]From here on, we use the terms *principal* and *user/role* interchangeably.

2. **Discretionary**: the user is the owner of o; or the user/role is assigned p by another user/role that has been assigned p with the GRANT option[2].

Because of the multiple ways by which a privilege can be obtained, conflicts are natural in cases where the same privilege, obtained from multiple sources, exists in different states. Therefore, a *conflict resolution* strategy must be defined to address such cases. Our strategy is to introduce a *privilege states dominance* (PSD) relation (see Figure 4.1). The PSD relation imposes a total order on the set of privilege states such that any two states are comparable under the PSD relation. Note the following characteristics of the semantics of the PSD relation. First, the *deny* state overrides all the other states to support the concept of a negative authorization [30]. Second, the *suspend*, and the *taint* states override the *grant* state as they can be triggered as potential response actions to an anomalous request. Finally, the *unassign* state is overridden by all the other states thereby preserving the traditional semantics of privilege assignment.

The PSD relation is the core mechanism that PSAC provides for resolving conflicts. For example, consider a user u that derives its privileges by being assigned a role r. Suppose that a privilege p is assigned to r in the *grant* state. Now suppose we directly deny p to u. The question is which is the state of privilege p for u, in that u has received p with two different states. We resolve such conflicts in PSAC using the PSD relation. Because in the PSD relation, the *deny* state overrides the *grant* state, p is denied to u.

We formally define a PSD relation as follows:

Definition 4.2.1 (**PSD Relation**) Let n be the number of privilege states.
Let $S = \{s_1, s_2 \ldots s_n\}$ be the set of privilege states. The PSD relation is a binary relation (denoted by $\preceq$) on S such that for all $s_i, s_j, s_k \in S$:

1. $s_i \preceq s_j$ means s_i overrides s_j

2. if $s_i \preceq s_j$ and $s_j \preceq s_i$, then $s_i = s_j$ (anti-symmetry)

[2]A privilege granted to a principal with the GRANT option allows the principal to grant that privilege to other principals [31].

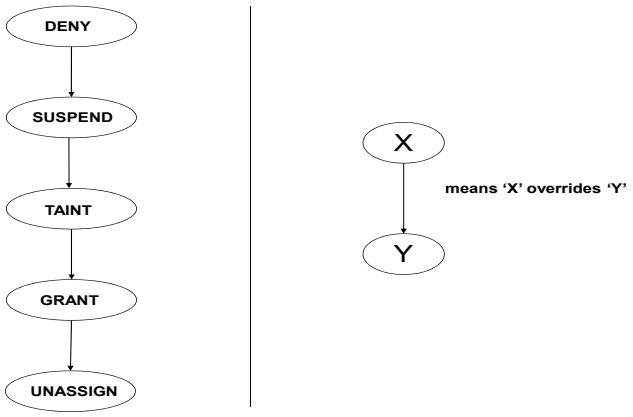


Fig. 4.1. Privilege States Dominance Relationship

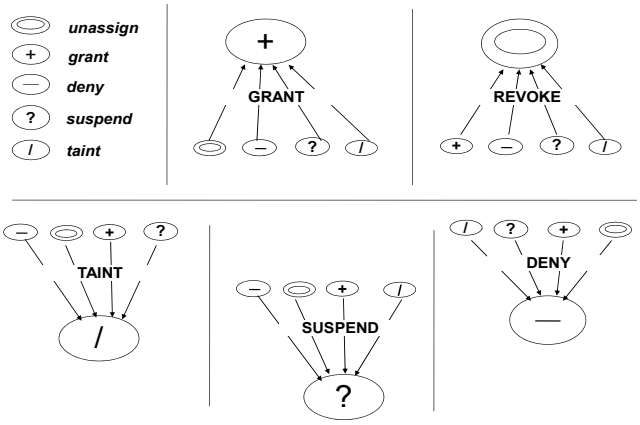


Fig. 4.2. Privilege State Transitions

3. if $s_i \preceq s_j$ and $s_j \preceq s_k$, then $s_i \preceq s_k$ (transitivity)

4. $s_i \preceq s_j$ or $s_j \preceq s_i$ (totality) $\square$

4.2.2 Privilege State Transitions

We now turn our attention to the privilege state transitions in PSAC. Initially, when a privilege is not assigned to a principal, it is in the *unassign* state for that principal. Thus, the *unassign* state is the default (or initial) state of a privilege. The state transitions can be triggered as internal response actions by an AD system, or as ad-hoc administrative

commands. In what follows, we discuss the various administrative commands available in PSAC to trigger privilege state transitions.

The GRANT command is used to assign a privilege to a principal in the *grant* state whereas the REVOKE command is used to assign a privilege to a principal in the *unassign* state. In this sense, these commands support similar functionality as the SQL-99 GRANT and REVOKE commands [31]. The DENY command assigns a privilege to a principal in the *deny* state. We introduce two new commands in PSAC namely, SUSPEND and TAINT, for assigning a privilege to a principal in the *suspend* and the *taint* states, respectively. The privilege state transitions are summarized in Figure 4.2. Note the constraint that a privilege assigned to a principal on a DBMS object can only exist in one state at any given point in time.

4.2.3 Formal Model

In this section, we formally define the privilege model for PSAC in the context of a DBMS. The model is based on the following relations and functions:

Relations

1. U, the set of all users in the DBMS.
2. R, the set of all roles in the DBMS.
3. $PR = U \cup R$, the set of principals (users/roles) in the DBMS.
4. OT, the set of all DBMS object types such as *server*, *database*, *schema*, *table*, and so forth.
5. O, the set of all DBMS objects of all object types.
6. OP, the set of all operations defined on the object types in OT, such as *select*, *insert*, *delete*, *drop*, *backup*, *disconnect*, and so forth.

7. $S = \{$*deny,suspend,taint,grant,unassign*$\}$, a totally ordered set of privilege states under the PSD relation (Definition 4.2.1).

8. $P \subset OP \times OT$, a many-to-many relation on operations and object types representing the set of all privileges. Note that not all operations are defined for all object types. For example, tuples of the form $(select, server)$ or $(drop, server)$ are not elements of P.

9. $URA \subseteq U \times R$, a many-to-many user to role assignment relation.

10. $PRUPOSA \subset PR \times U \times P \times O \times S$, a principal to user to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to users (assignees) by other principals (assigners) on DBMS objects[3].

11. $PRRPOSA \subset PR \times R \times P \times O \times S$, a principals to role to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to roles (assignees) by principals (assigners).

These relations capture the state of the access control system in terms of the privilege and the role assignments. The functions defined below determine the state of a privilege assigned to a user/role on a DBMS object.

Functions

1. $assigned\_roles(u) : U \rightarrow 2^R$, a function mapping a user u to its assigned roles such that $assigned\_roles(u) = \{r \in R \mid (u, r) \in URA\}$. This function returns the set of roles that are assigned to a user.

[3]In PSAC, a role can also be an assigner of privileges. Consider a situation when a user u gets a privilege p (with grant option) through assignment of role r. If u grants p to some other user u', PSAC records p as being granted to u' by r even though the actual GRANT command was executed by u.

2. $priv\_states(pr, r', p, o) : PR \times R \times P \times O \rightarrow 2^S$, a function mapping a principal pr (privilege assigner), a role r', a privilege p, and an object o to a set of privilege states such that $priv\_states(pr, r', p, o) = \{s \in S \mid (pr, r', p, o, s) \in PRRPOSA\}$. This function returns the set of states for a privilege p, that is directly assigned to the role r' by the principal pr, on an object o.

3. $priv\_states(pr, u', p, o) : PR \times U \times P \times O \rightarrow 2^S$, a function mapping a principal pr (privilege assigner), a user u', a privilege p, and an object o to a set of privilege states such that $priv\_states(pr, u', p, o) = \{s \in S \mid (pr, u', p, o, s) \in PRUPOSA\}$ $\cup_{r \in assigned\_roles(u')}\ priv\_states(pr, r, p, o)$. The set of states returned by this function is the union of the privilege state directly assigned to the user u' by the principal pr, and the privilege states (also assigned by pr) obtained through the roles assigned to u'.

4. $priv\_states(r, p, o) : R \times P \times O \rightarrow 2^S$, a function mapping a role r, a privilege p, and an object o to a set of privilege states such that $priv\_states(r, p, o) = \cup_{pr \in PR}$ $priv\_states(pr, r, p, o)$. This function returns the set of states for a privilege p, that is directly assigned to the role r by any principal in the DBMS, on an object o.

5. $priv\_states(u', p, o) : U \times P \times O \rightarrow 2^S$, a function mapping a user u', a privilege p, and an object o to a set of privilege states such that $priv\_states(u', p, o) = \cup_{pr \in PR}$ $priv\_states(pr, u', p, o)$. This function returns the set of states for a privilege p, that is directly assigned to the user u' by any principal in the DBMS, on an object o.

6. $PSD\_state(2^S) : 2^S \rightarrow S$, a function mapping a set of states 2^S to a state $\in S$ such that $PSD\_state(2^S) = s' \in 2^S \mid \forall_{s \in 2^S | s \neq s'}\ s' \preceq s$. This function returns the final state of a privilege using the PSD relation.

4.2.4 Role Hierarchy

Traditionally, roles can be arranged in a conceptual hierarchy using the role-to-role assignment relation. For example, if a role r_2 is assigned to a role r_1, then r_1 becomes a parent

of r_2 in the conceptual role hierarchy. Such hierarchy signifies that the role r_1 inherits the privileges of the role r_2 and thus, is a more *privileged* role then r_2. However, in PSAC such privilege inheritance semantics may create a problem because of a $deny/suspend/taint$ state attached to a privilege. The problem is as follows. Suppose a privilege p is assigned to the role r_2 in the *deny* state. The role r_1 will also have such privilege in the *deny* state since it inherits it from the role r_2. Thus, denying a privilege to a lower level role has the affect of denying that privilege to all roles that inherit from that role. This defeats the purpose of maintaining a role hierarchy in which roles higher up the hierarchy are supposed to be more privileged than the descendant roles. To address this issue, we introduce the concept of *privilege orientation*. We define three privilege orientation modes namely, *up, down*, and *neutral*. A privilege assigned to a role in the *up* orientation mode means that the privilege is also assigned to its parent roles. On the other hand, a privilege assigned to a role in the *down* orientation mode means that the privilege is also assigned to its children roles; while the *neutral* orientation mode implies that the privilege is neither assigned to the parent roles nor to the children roles. We put the following two constraints on the assignment of orientation modes on the privileges.

- A privilege assigned to a role in the *grant* or in the *unassign* state is always in the *up* orientation mode thereby maintaining the traditional privilege inheritance semantics in a role hierarchy.

- A privilege assigned to a role in the *deny, taint*, or *suspend* state may only be in the *down* or in the *neutral* orientation mode. Assigning such privilege states to a role in the *down* or *neutral* mode ensures that the role still remains more privileged than its children roles. In addition, the *neutral* mode is particularly useful when a privilege needs to be assigned to a role without affecting the rest of the role hierarchy (when responding to an anomaly, for example).

We formalize the privilege model of PSAC in the presence of a role hierarchy as follows:

1. $RRA \subset R \times R$, a many-to-many role to role assignment relation. A tuple of the form $(r_1, r_2) \in R \times R$ means that the role r_2 is assigned to the role r_1. Thus, role r_1 is a parent of role r_2 in the conceptual role hierarchy.

2. $OR = \{up, down, neutral\}$, the set of privilege orientation modes.

3. $PRRPOSORA \subset PR \times R \times P \times O \times S \times OR$, a principal to role to privilege to object to state to orientation mode assignment relation. This relation captures the state of the access control system in terms of the privileges, their states, and their orientation modes that are directly assigned to roles by principals.

4. $assigned\_roles(r') : R \rightarrow 2^R$, a function mapping a role r' to its assigned roles such that $assigned\_roles(r') = \{r \in R \mid (r', r) \in RRA\} \cup assigned\_roles(r)$. This function returns the set of the roles that are directly and indirectly (through the role hierarchy) assigned to a role; in other words, the set of descendant roles of a role in the role hierarchy.

5. $assigned\_roles(u) : U \rightarrow 2^R$, a function mapping a user u to its assigned roles such that $assigned\_roles(u) = \{r \in R \mid (u, r) \in URA\} \cup assigned\_roles(r)$. This function returns the set of roles that are directly and indirectly (through the role hierarchy) assigned to a user.

6. $assigned\_to\_roles(r') : R \rightarrow 2^R$, a function mapping a role r' to a set of roles such that $assigned\_to\_roles(r') = \{r \in R \mid (r, r') \in RRA\} \cup assigned\_to\_roles(r)$. This function returns the set of roles that a role is directly and indirectly (through the role hierarchy) assigned to; in other words, the set of ancestor roles of a role in the role hierarchy.

   We redefine the $priv\_states(pr, r', p, o)$ function in the presence of a role hierarchy taking into account the privilege orientation constraints as follows:

7. $priv\_states(pr, r', p, o) : PR \times R \times P \times O \rightarrow 2^S$, a function mapping a principal pr, a role r', a privilege s, and an object o to a set of privilege states such that

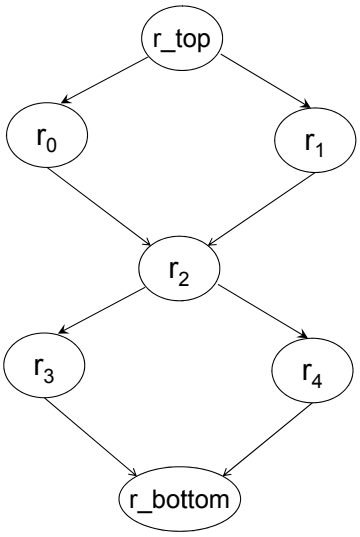


Fig. 4.3. A Sample Role Hierarchy

$priv\_states(pr, r', p, o) = \{s \in S \mid \forall\ or \in OR, (pr, r', p, o, s, or) \in PRRPOSORA\ \} \cup \{s \in \{grant, unassign\} \mid \forall\ r \in assigned\_roles(r'), (pr, r, p, o, s, `up') \in PRRPOSORA\ \} \cup \{s \in \{deny, suspend, taint\} \mid \forall\ r \in assigned\_to\_roles(r'), (pr, r, p, o, s, `down') \in PRRPOSORA\ \}$. The set of privilege states returned by this function is the union of the privilege states directly assigned to the role r' by the principal pr, the privilege states in the *grant* or the *unassign* states (also assigned by pr) obtained through the descendant roles of r', and the privilege states in the *deny*, *suspend*, and *taint* states (also assigned by pr) obtained through the roles that are the ancestor roles of r', and that are in the *down* orientation mode.

We now present a comprehensive example of the above introduced relations and functions in PSAC. Consider a sample role hierarchy in Figure 4.3. Table 4.2 shows the state of a sample $PRRPOSORA$ relation.

Table 4.2
$PRRPOSORA$ relation

PR	**R**	**P**	**O**	**S**	**OR**
SU_1	r_top	$select$	t_1	$deny$	$neutral$
SU_1	r_0	$select$	t_1	$taint$	$down$
SU_1	r_bottom	$select$	t_1	$grant$	up
SU_2	r_top	$select$	t_1	$suspend$	$down$

Let the role r_2 be assigned to the user u_1. To determine the final state of the $select$ privilege on the table t_1 for the user u_1, we evaluate $priv\_states(u_1, select, t_1)$ as follows:

$$
\begin{aligned}
&priv\_states(u_1, select, t_1) \\
&= priv\_states(SU_1, u_1, select, t_1) \cup \\
&\quad priv\_states(SU_2, u_1, select, t_1) \\
&= priv\_states(SU_1, r_2, select, t_1) \cup \\
&\quad priv\_states(SU_2, r_2, select, t_1) \\
&= \{taint\} \cup \\
&\quad \{grant\} \cup \{suspend\} \\
&= \{taint, grant, suspend\}
\end{aligned}
$$

The final state is determined using the $PSD\_state()$ function as follows:

$$PSD\_state(taint, grant, suspend) = suspend$$

4.3 Implementation and Experiments

In this section, we present the details on how to extend a real-world DBMS with PSAC. We choose to implement PSAC in the PostgreSQL 8.3 open-source object-relational DBMS [5]. In the rest of the section, we use the term PSAC:PostgreSQL to indicate PostgreSQL extended with PSAC, and BASE:PostgreSQL to indicate the official PostgreSQL 8.3 release. The implementation of PSAC:PostgreSQL has to meet two design requirements. The first requirement is to maintain backward compatibility of PSAC:PostgreSQL with BASE:PostgreSQL. We intend to release PSAC:PostgreSQL for general public use in the near future; therefore it is important to take into account the backward compatibility issues in our design. The second requirement is to minimize the overhead for maintaining privilege states in the access control mechanism. In particular, we show that the time taken for the access control enforcement code in the presence of privilege states is not much

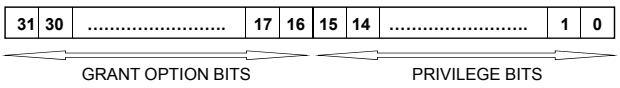


Fig. 4.4. ACLItem privs field

higher than the time required by the access control mechanism of BASE:PostgreSQL. In what follows, we first present the design details of PSAC:PostgreSQL, and then we report experimental results showing the efficiency of our design.

4.3.1 PSAC:PostgreSQL

Access control in BASE:PostgreSQL is enforced using access control lists (ACLs). Every DBMS object has an ACL associated with it. An ACL in BASE:PostgreSQL is a one-dimensional array; the elements of such an array have values of the *ACLItem* data type. An ACLItem is the basic unit for managing privileges of an object. An ACLItem is implemented as a structure with the following fields: *granter*, the user/role granting the privileges; *grantee*, the user/role to which the privileges are granted; and *privs*, a 32 bit integer (on 32 bit machines) managed as a bit-vector to indicate the privileges granted to the grantee. A new ACLItem is created for every unique pair of granter and grantee. There are 11 pre-defined privileges in BASE:PostgreSQL with a bit-mask associated with each privilege [32]. As shown in Figure 4.4, the lower 16 bits of the privs field are used to represent the granted privileges, while the upper 16 are used to indicate the $grant$ option[4]. If the k^{th} bit is set to 1 ($0 \leq k < 15$), privilege p_k is granted to the user/role. If the $(k + 16)^{th}$ bit is also set to 1, then the user/role has the grant option on privilege p_k.

Design Details

There are two design options to extend BASE:PostgreSQL to support privilege states. The first option is to extend the ACLItem structure to accommodate privilege states. The

[4]Recall that the grant option is used to indicate that the granted privilege may be granted by the grantee to other users/roles.

second option is to maintain the privilege states in a separate data structure. We chose the latter option. The main reason is that we want to maintain backward compatibility with BASE:PostgreSQL. Extending the existing data structures can introduce potential bugs at other places in the code base that we want to avoid. In BASE:PostgreSQL, the pg_class system catalog is used to store the metadata information for database objects such as tables, views, indexes and sequences. This catalog also stores the ACL for an object in the *acl* column that is an array of ACLItems. We extend the pg_class system catalog to maintain privilege states by adding four new columns namely: the *acltaint* column to maintain the tainted privileges; the *aclsuspend* column to maintain the suspended privileges; the *acldeny* column to maintain the denied privileges; and the *aclneut* column to indicate if the privilege is in the *neutral* orientation mode. Those state columns and the *aclneut* column are of the same datatype as the *acl* column, that is, an array of ACLItems. The lower 16 bits of the privs field in those state and *aclneut* columns are used to indicate the privilege states and the orientation mode respectively. This strategy allows us to use the existing privilege bit-masks for retrieving the privilege state and orientation mode from these columns. The upper 16 bits are kept unused. Table 4.3 is the truth table capturing the semantics of the privs field bit-vector in PSAC:PostgreSQL.

Authorization Commands

We have modified the BASE:PostgreSQL GRANT and REVOKE authorization commands to implement the privilege state transitions. In addition, we have defined and implemented in PSAC:PostgreSQL three new authorization commands, that is, the DENY, the SUSPEND, and the TAINT commands. As discussed in the Section 4.2, the DENY command moves a privilege to the *deny* state, the SUSPEND command moves a privilege to the *suspend* state, and the TAINT command moves a privilege to the *taint* state. The default privilege orientation mode for these commands is the *down* mode with the option to override that by specifying the *neutral* orientation mode. The administrative model for these commands is similar to that of the SQL-99 GRANT command, that is, a DENY/SUS-

Table 4.3
Privilege States/Orientation Mode for the privs field in PSAC:PostgreSQL

acl k^{th} bit	**acl taint** k^{th} bit	**acl suspend** k^{th} bit	**acl deny** k^{th} bit	**acl neut** k^{th} bit	p_k **state**
0	0	0	0	0	unassign/up
1	0	0	0	0	grant/up
0	1	0	0	0	taint/down
0	0	1	0	0	suspend/down
0	0	0	1	0	deny/down
0	1	0	0	1	taint/neutral
0	0	1	0	1	suspend/neutral
0	0	0	1	1	deny/neutral
Rest all other combinations are not allowed by the system.					

Table 4.4
New Authorization Commands in PSAC:PostgreSQL

TAINT {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]
SUSPEND {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]
DENY {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]

PEND/TAINT command can be executed on privilege p for object o by a user u iff u has the grant option set on p for o or u is the owner of o. The syntax for the commands is reported in Table 4.4. Note that in the current version of PSAC:PostgreSQL, the new commands are applicable on the database objects whose metadata are stored in the pg_class system catalog.

Access Control Enforcement

We have instrumented the access control enforcement code in BASE:PostgreSQL with the logic for maintaining the privilege states and orientation modes. The core access control function in BASE:PostgreSQL returns a true/false output depending on whether the privilege under check is granted to the user or not. In contrast, the core access control function in PSAC:PostgreSQL returns the final state of the privilege to the calling function. The calling function then executes a pre-configured action depending upon the state of the privilege. As a proof of concept, we have implemented a re-authentication procedure in PSAC:PostgreSQL when a privilege is in the *suspend* state. The re-authentication procedure is as follows:

Re-authentication Procedure. Recall that when a privilege is in the $suspend$ state, further negotiation with the end-user must be satisfied before the user-request is executed by the

DBMS. In the current version of PSAC, we implement a procedure that re-authenticates the user if a privilege, after applying the PSD relationship, is found in the *suspend* state. The re-authentication scheme is as follows. In BASE:PostgreSQL, an authentication protocol is carried out with the user whenever a new session is established between a client program and the PostgreSQL server. In PSAC:Postgresql, the same authentication protocol is replayed in the middle of a transaction execution when access control enforcement is in progress, and a privilege is found in the *suspend* state. We have modified the client library functions of BASE:PostgreSQL to implement such protocol in the middle of a transaction execution. If the re-authentication protocol fails, the user request is dropped. If it succeeds, the request proceeds as usual, and no changes are made to the state of the privilege. Note that such re-authentication procedure scheme is implemented as a proof-of-concept in PSAC:Postgresql. More advanced forms of actions such as a second-factor of authentication can also be implemented.

Access Control Enforcement Algorithm. The pseudo-code for the access control enforcement algorithm in PSAC:PostgreSQL is presented in the Listing 4.1. The function *aclcheck()* takes as input a privilege in_priv - whose state needs to be determined, a database object in_object - that is the target of a request, and a user in_user - the user exercising the usage of in_priv. The output of the algorithm is the state of the in_priv. The algorithm proceeds as follows. Since we define a total order on the privilege states, it is sufficient to check each state in the order of its rank in the PSD relation (cfr. Section 4.2). Thus, we first check for the existence of in_priv in the *deny* state, followed by the *suspend* state, the *taint* state, and then the *grant* state. The function for checking the state of in_priv (function *check_priv()*) in an Acl is designed to take into account all the roles that are directly and indirectly (through a role hierarchy) assigned to the in_user. Note that most expensive operation in the *check_priv()* function is the run-time inheritance check of roles, that is, to check whether the *user_role* is an ancestor or descendant of the *acl_role* (lines 58 and 62). We make such check a constant time operation in our implementation by maintaining a cache of the assigned roles for every user/role in the DBMS. Thus, the running

time of the access control enforcement algorithm is primarily dependent upon the sizes of various Acls.

If the privilege is not found to be in the above mentioned states, the *unassign* state is returned as the output of the access check algorithm.

```
1 ------------------------------------------------------------
2 Input
3 in_user    : The user executing the command
4 in_object  : Target database object
5 in_priv    : Privilege to check
6
7 Output
8 The privilege state
9 ------------------------------------------------------------
10 function aclcheck(in_user, in_object, in_priv) returns state
11 {
12   //Get the neutral orientation ACL for in_object
13   NeutACL = get_neut_ornt(in_object);
14
15   //Deny if in_user has in_priv in DENY state
16   DenyACL = get_deny_state_acl(in_object);
17   if (check_priv(in_priv,DenyACL,in_user,NeutACL,DENY) == true)
18       return DENY;
19
20   //Suspend if in_user has in_priv in SUSPEND state
21   SuspendACL = get_suspend_state_acl(in_object);
22   if (check_priv(in_priv,SuspendACL,in_user,NeutACL,SUSPEND) ==
       true)
23       return SUSPEND;
24
25   //Taint if in_user has in_priv in TAINT state
26   TaintACL = get_taint_state_acl(in_object);
```

```
27   if (check_priv(in_priv,TaintACL,in_user,NeutACL,TAINT) == true)
28     return TAINT;
29
30   //Grant if in_user has in_priv in GRANT state
31   GrantACL = get_grant_state_acl(in_object);
32   if (check_priv(in_priv,GrantACL,in_user,NeutACL,GRANT) == true)
33     return GRANT;
34
35   //Else return UNASSIGN state
36   return UNASSIGN;
37 }
38 ------------------------------------------------------------
39 function check_priv(in_priv,AclToCheck,in_user,NeutACL,
     state_to_check)
40 returns boolean
41 {
42   //First, perform the inexpensive step of checking the
       privileges directly assigned to the in_user
43   if (in_user has in_priv in AclToCheck)
44     return true;
45
46   //Get all the roles directly assigned to in_user
47   user_role_list = get_roles(in_user);
48
49   //Do the following for every role directly assigned to in_user
50   for each user_role in user_role_list
51   {
52     //Do the following for every role entry in AclToCheck
53     for each acl_role in AclToCheck
54     {
55       if (state_to_check == GRANT)
```

```
56        {
57          //Orientation of privileges in GRANT state is UP
58          if ((user_role == acl_role OR user_role is an ANCESTOR of
                acl_role) AND acl_role has in_priv)
59            return true;
60        }
61        else if ((user_role == acl_role OR user_role is a
             DESCENDANT of acl_role) AND acl_role has in_priv)
62        {
63          if (acl_role has in_priv in NeutACL)
64            continue looping through AclToCheck;
65          else
66            return true;
67        }
68      }
69    }
70
71    return false;
72  }
```

Listing 4.1 Access Control Enforcement Algorithm in PSAC:PostgreSQL

4.3.2 Experimental Results

In this section, we report the experimental results comparing the performance of the access control enforcement mechanism in BASE:PostgreSQL and PSAC:PostgreSQL. Specifically, we measure the time required by the access control enforcement mechanism to check the state of a privilege, $test\_priv$, for a user, $test\_user$, on a database table, $test\_table$. We vary the *ACL Size* parameter in our experiments. For BASE:ProstgreSQL, the *ACL Size* is the number of entries in the acl column of the pg_class catalog. For PSAC:PostgreSQL, the ACL size is the combined number of entries in the acl, the $acldeny$, the $aclsuspend$, and

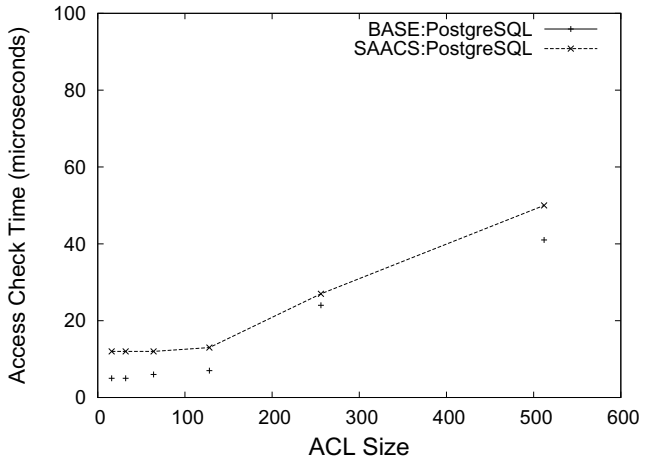


Fig. 4.5. Exp 1: Access Control Enforcement Time in BASE and PSAC PostgreSQL in the absence of a role hierarchy

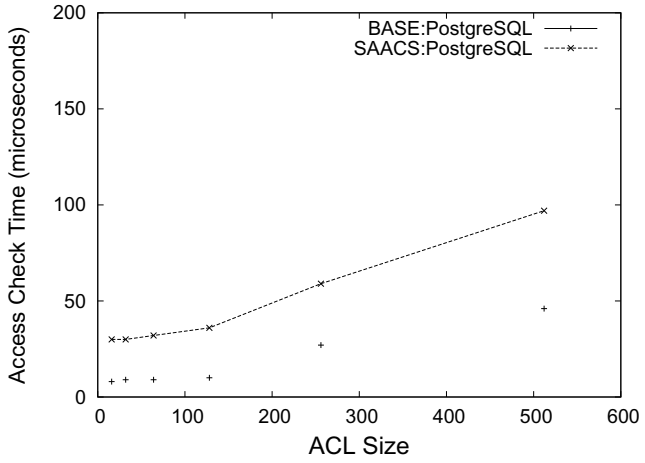


Fig. 4.6. Exp 2: Access Control Enforcement Time in BASE and PSAC PostgreSQL in the presence of a role hierarchy

the $acltaint$ columns. Note that for the purpose of these experiments we do not maintain any privileges in the *neutral* orientation mode.

We perform two sets of experiments. The first experiment compares the access control overhead in the absence of a role hierarchy. The results are reported in Figure 4.5. As expected, the access control overhead for both BASE and PSAC PostgreSQL increases with the ACL Size. The key observation is that the access control overhead for PSAC:PostgreSQL is not much higher than that of BASE:PostgreSQL.

The second experiment compares the access control overhead in the presence of a hypothetically large role hierarchy. We use a role hierarchy of 781 roles with depth equal to 4. The edges and cross-links in the role hierarchy are randomly assigned. The rational behind such set-up is that we want to observe a reasonable amount of overhead in the access control enforcement code. The role hierarchy is maintained in PSAC:PostgreSQL in a manner similar to that in BASE:PostgreSQL, that is, a role r_p is the parent of a role r_c if r_c is assigned to r_p using the GRANT ROLE command. A role and its assigned roles are stored in the $pg\_auth\_members$ catalog [5]. Next, in the experiment, we randomly assigned 10 roles to the $test\_user$. In order to vary the size of the ACL on the $test\_table$, we developed a procedure to assign privileges on the $test\_table$ to randomly chosen roles. Figure 4.6 reports the results of this experiment. First, observe that the access check time in the presence of a role hierarchy is not much higher than that in the absence of a role hierarchy. As mentioned before, this is mainly because we maintain a cache of the roles assigned to a user (directly or indirectly), thus preventing expensive role inheritance tests at the run-time. Second, the access control enforcement algorithm of PSAC:PostgreSQL reported in Section 4.3.1 is very efficient with a maximum time of approximately 97 microseconds for an ACL of size 512. Also, it is not much higher than the maximum access control enforcement time in BASE:PostgreSQL which stands at approximately 46 microseconds.

Overall, the two experiments confirm the extremely low overhead associated with our design in PSAC:PostgreSQL.

4.4 Conclusion

In this Chapter, we presented the design, formal model and implementation of a privilege state based access control (PSAC) system tailored for a DBMS. The fundamental idea in PSAC is that a privilege, assigned to a principal on an object, has a state attached to it. We identify five states in which a privilege can exist namely, unassign, grant, taint, suspend and deny. A privilege state transition to either the taint or the suspend state acts as a fine-grained response to a database anomaly. We designed PSAC to take into account a role hierarchy. We also introduced the concept of privilege orientation to control the propagation of privilege states in a role hierarchy. We extended the PostgreSQL DBMS with PSAC describing various design issues related to the actual implementation of PSAC. We also reported experimental results that confirm that our techniques are efficient.

5. INTRUSION DETECTION IMPLEMENTATION IN POSTGRESQL

A major portion of the thesis involves a prototype implementation of our intrusion detection and response mechanism in the Postgresql 8.3 DBMS. In Chapter 3, we have described our approach towards extending PostgreSQL with the intrusion response mechanism. We presented the implementation details, and experimental results on the overhead of the policy matching and policy signature verification procedures. In Chapter 4, we have presented the details on the integration of the PSAC model with PostgreSQL's access control system. We also reported experimental results on the overhead of for maintaining the privilege states in PostgreSQL's access control enforcement mechanism.

In this Chapter, we describe in detail our design choices and strategies towards implementing our intrusion detection mechanism in PostgreSQL. We begin with revisiting the intrusion detection algorithm in the context of our implementation in Section 5.1. In Section 5.2, we discuss the internals of the core query processing and statistics collection architecture in PostgreSQL. In Section 5.3, we describe how we integrate the intrusion detection procedure with the existing query processing architecture. In Section 5.4, we present extensive experimental results on the overhead of our implementation on the transaction processing capabilities of PostgreSQL. The experimental results show that our methods are not only feasible but efficient as well (considering that the implementation is a research prototype).

5.1 Anomaly Detection Algorithm

We have implemented the Naive Bayes Classification (NBC) algorithm for the role-based anomaly detection procedure described in Chapter 2. There are a few differences in the extraction of features from a SQL query between our implementation and the the-

ory presented in Chapter 2. In the implementation, we only consider the tables in the [RELATION LIST] of the FROM clause of the SQL query and the columns in the [TARGET LIST] of the projection clause when extracting the features. This limitation is due to our approach towards role profile creation in our implementation. As we explain later in Section 5.3, for creating the role profiles during the training phase, we update the table and column access count on a per role basis during the access control enforcement procedure for the SQL command under consideration. This is necessary since we want to collect the table and the column information on a per role basis, and this information is only available during the access control enforcement procedure in PostgreSQL's flow of query execution. Also, the access control enforcement procedure only checks the privileges of the user (or role) against the tables mentioned in the [RELATION LIST] of the FROM clause of the SQL query, thus we are only able to gather such information for the detection process. In future, we plan to extend our implementation to gather additional information in the QUALIFICATION component of the query to make it consistent with the feature extraction procedure presented in Chapter 2.

With this modification in the feature extraction process, the information that we gather for various *quiplet* types introduced in Chapter 2 is described in Table 5.1:

The information in the role-profiles for the various quiplet types is described in Table 5.2. Note that for the medium-quiplets we do not maintain the frequency count of the tables accessed in the [RELATION LIST]. This is because in a SQL command, if the number of columns accessed for a table is greater than 0, this implies that the table was accessed in the SQL command. Since we use the NBC for the classification task, in which the underlying assumption is the independence of the features, we can not have a feature that can be implied from another feature thereby breaking the independence assumption. Similar argument is applicable for not maintaining the frequency count of the tables accessed for the fine-triplets since if a column in a table was accessed in a SQL command, it implies that the table containing that column was also accessed in the command.

We now give some examples of the feature extraction and the role-profile construction process for SELECT, INSERT, and UPDATE SQL commands. Consider a database schema

Table 5.1
Quiplet Feature Extraction

Quiplet Type	features
Coarse	SQL Command Number of tables in the `[RELATION LIST]` Number of Columns in the `[TARGET LIST]`
Medium	SQL Command Tables in the `[RELATION LIST]` Number of columns per table in the `[TARGET LIST]`
Fine	SQL Command Tables in the `[RELATION LIST]` Columns in the `[TARGET LIST]`

Table 5.2
Role Profile Information for Various Quiplet Types

Quiplet Type	profile information
Coarse	SQL Command Count Frequency Count of Number of tables in the `[RELATION LIST]` Frequency Count of Number of columns in the `[TARGET LIST]`
Medium	SQL Command Count Frequency count of number of columns per table for all tables
Fine	SQL Command Count Frequency access count of every column in every table Frequency non-access count of every column in every table

Table 5.3
Quiplet construction example

SQL Command	c-quiplet	m-quiplet	f-quiplet
Select $R_1.A_1, R_3.C_3$, $R_3.B_3$ From R_1, R_3	select $< 2 > < 2 >$	select $< 1, 0, 1 >$ $< 1, 0, 2 >$	select $< 1, 0, 1 >$ $<< 1, 0, 0 >$ $< 0, 0, 0 >$ $< 0, 1, 1 >>$
INSERT INTO R_2 VALUES $(1, 1, 1)$	insert $< 1 > < 3 >$	insert $< 0, 1, 0 >$ $< 0, 3, 0 >$	insert $< 0, 1, 0 >$ $<< 0, 0, 0 >$ $< 1, 1, 1 >$ $< 0, 0, 0 >>$
UPDATE R_3 SET $R_3.C_3 = 100$	update $< 1 > < 1 >$	update $< 0, 0, 1 >$ $< 0, 0, 1 >$	update $< 0, 0, 1 >$ $<< 0, 0, 0 >$ $< 0, 0, 0 >$ $< 0, 0, 1 >>$

consisting of the following three relations $R_1 = \{A_1, B_1, C_1\}$, $R_2 = \{A_2, B_2, C_2\}$, and $R_3 = \{A_3, B_3, C_3\}$. Table 5.3 shows the quiplet construction for a SELECT, a INSERT, and an UPDATE command respectively.

The application of the NBC in the implementation follows directly from our earlier discussion in Chapter 2. The probabilities are calculated using the m-estimate technique as defined in Definition 2.3.1. The parameter m determines how heavily to weight the observed probability relative to the observed data. The fraction $\frac{n_e}{|D_T|}$ in the Definition 2.3.1 is the initial (or prior) probability of an event e. Since we are free to choose a prior in our implementation, we choose the $zipf$ probability distribution to model the prior probabilities of the events.

In what follows, we briefly describe the internals of the query execution flow in PostgreSQL. We then describe how we extend such flow to integrate our intrusion detection procedure.

5.2 PostgreSQL Internals

PostgreSQL, often simply Postgres, is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department [33]. It is released under an MIT-style license and is thus free and open source software. The key features of PostgreSQL are, but not limited to, support for user-defined functions (in many flavors of languages such as C++, Java, R etc), B+-tree, hash, GiST and GiN indexes, triggers, multi-version concurrency control, query re-writing, wide-variety of built-in data types (and support for user-defined types), table inheritance, and support for user-defined roles [34]. Our implementation is based on the PostgreSQL version 8.3 [5][1].

The core query processing architecture in PostgreSQL is shown in Figure 5.1. The main server process called *postmaster* spawns a new server process called *postgres* for every new connection to a database. Every SQL query sent on that connection is handled by this new postgres process. A SQL query is received by the postgres process via data packets arriving through TCP/IP or Unix domain sockets. The query string passes through the *query parser* which creates a parse tree of the query structure. The next step is for the parse tree to be modified by any VIEWS or RULES that may apply to the query. This is performed by the *query rewrite* system. After the query has been rewritten, the *query optimizer* takes the parse tree and generates an optimal query plan that contains the operations to be performed to execute the query. The plan is then passed to the *query executor* that is responsible for execution of the query and passing the results back to the client. Before the executor begins the query execution, it checks whether the user has the privileges (directly or indirectly through role membership) to execute the query under consideration [35].

[1]PostgreSQL 8.3 was the latest release available when we started development of our methods in 2007.

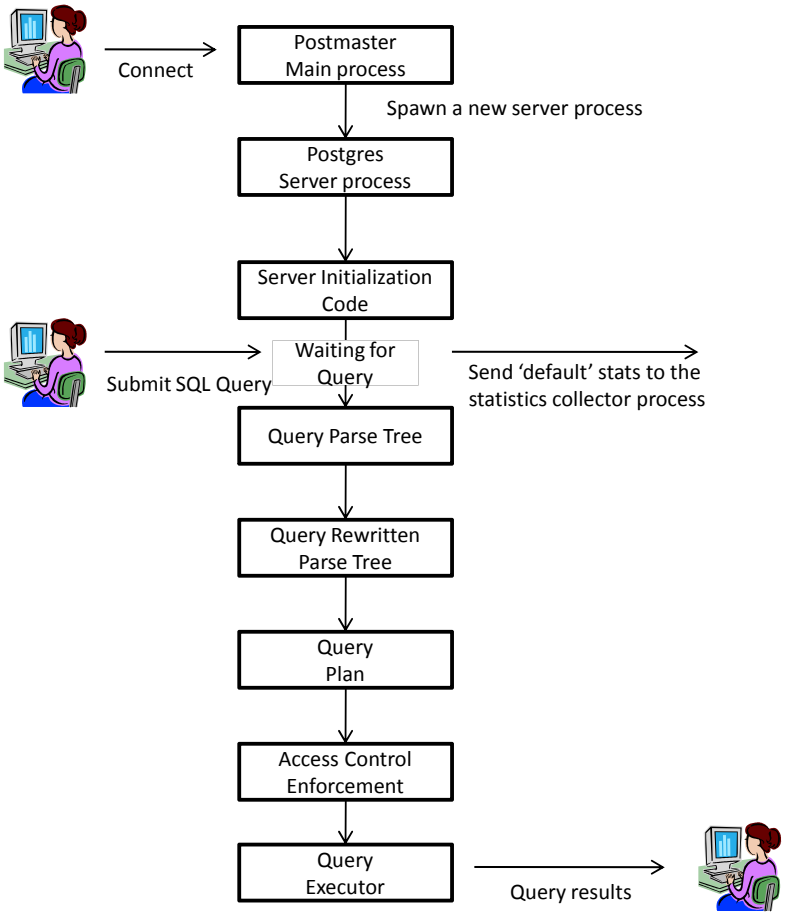


Fig. 5.1. PostgreSQL Query Processing Flow

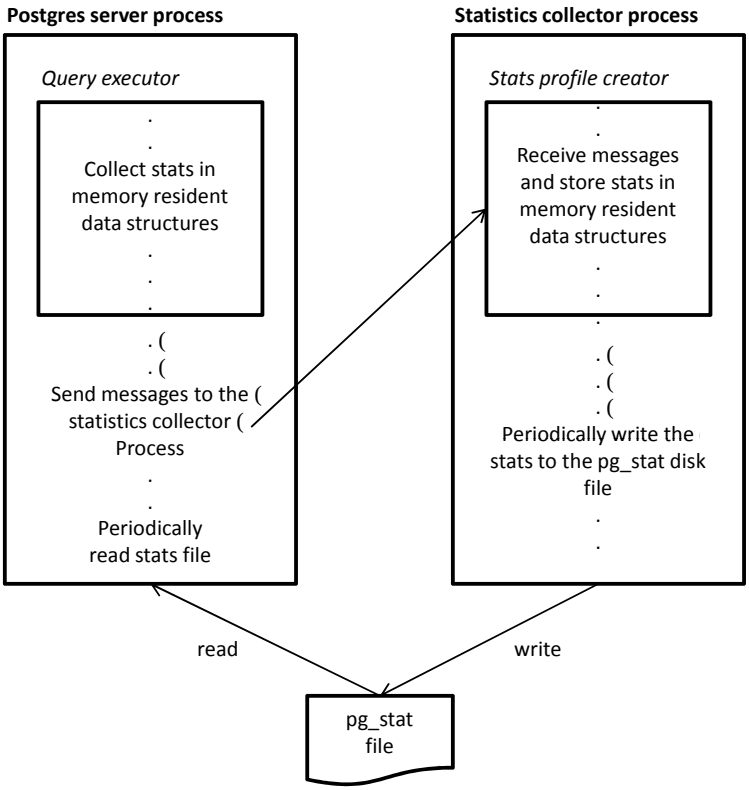


Fig. 5.2. PostgreSQL Statistics Collector Framework

A key component of the PostgreSQL DBMS is the statistics collection framework. PostgreSQL's statistics collector is a mechanism that supports collection and reporting of information about server activity such as a count of accesses to tables and indexes in both disk-block and individual-row terms, a count of total numbers of rows in each table, and so forth. The DBMS can be configured to collect or not collect statistics based on some configuration parameters. Several pre-defined views are available to show the results of the statistics collection. Figure 5.2 shows the interaction between the statistics collector process and the postgres server process. The statistics are collected by the postgres server process in memory resident data structures during query execution. At regular intervals (500 ms by default), the collected statistics are sent to the statistics collector process using UDP messages. More that one message may be sent at a time since the size of a single message is kept at 1024 bytes at the maximum to avoid any fragmentation of the packets at the network layer. The statistics collector process upon receiving a message updates the memory resident data structures for maintaining the various statistics. At regular intervals (500 ms by default), the statistics are written to a memory resident pg_stat file. The postgres server process, that needs to access the statistics, reads the pg_stat file at regular intervals (500 ms by default). When the postgres server is shutdown, the memory resident pg_stat file is persisted to a disk file.

In what follows, we describe our implementation strategy for the detection and response mechanism within such framework.

5.3 Our Implementation Strategy

Figure 5.3 shows the query processing architecture, that was described earlier in Figure 5.1, with our hooks for statistics collection required for the detection task, the anomaly detection, and anomaly response mechanisms. When a new connection is established to the DBMS by a database user, we report the login statistics to the statistics collector process that includes the roles activated by the user, and the list of tables under intrusion detection. Note that we allow administrators to configure the database schemas on which we collect

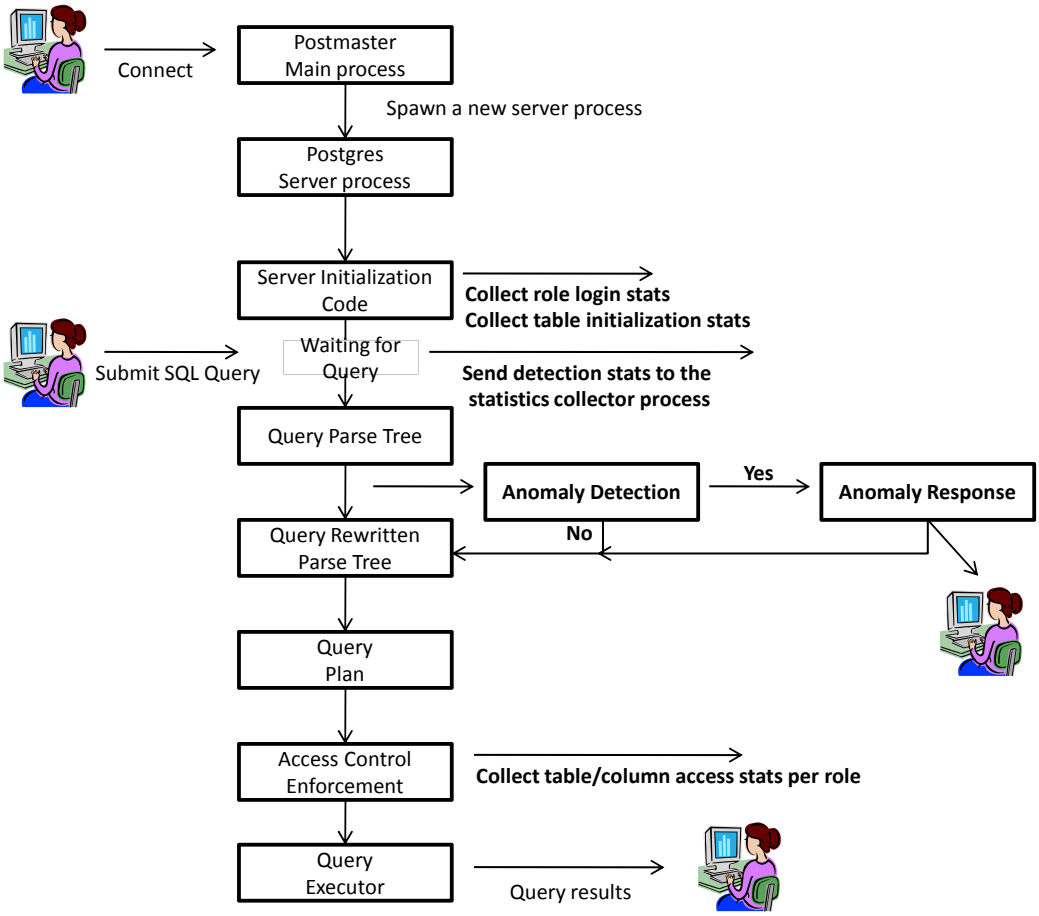


Fig. 5.3. Anomaly Detection and Data Collection Hooks in PostgreSQL

the statistics and perform the detection task. The major portion of the statistics required to carry out the detection task are collected during the access control enforcement procedure. The collected statistics include the command count per role, the tables accessed per command per role, and the columns accessed per table per command per role. Note that we assume that a strict RBAC model is under operation and thus all privileges required to access any portion of the database table are inherited from roles. This allows us to capture the required statistics on a per role basis. For the detection algorithm based on the NBC, we only require the statistics on the command count, the table access count, and the column access count on a per role basis. Thus, the table and column statistics are aggregated on a per role basis before being sent to the statistics collector. Note that the aggregation of the statistics differs based on the type of quiplet in use. As discussed in Section 5.1, for coarse quiplets, we only require the count of number of tables and columns accessed per role, for medium quiplets we require the count of number of columns accessed per table per role, while for the fine quiplets we require the count of table column that was accessed on a per role basis. The statistics collector, upon receiving the statistics, updates the memory resident role profiles that are then periodically written to the pg_stat file in a manner similar to the description in Section 5.2.

The intrusion detection algorithm task is performed on a query under consideration after the query parser has generated the parse tree. Using the parse tree generated by PostgreSQL means that we do not have to parse the query again to get the features required for the detection task. The pseudo-code for the detection algorithm is presented in Listing 5.1. The algorithm uses pre-defined functions to access the statistics required for the NBC from the pg_stat file. The result of the detection algorithm is whether an anomaly has been detected or not. As explained earlier, we specify a query as anomalous if the role associated with the database user (submitting the query) does not match the role predicted by the NBC. Our current implementation, thus, only supports single role activation by a user in a session.

```
1 ------------------------------------------------------------------
2 Input
3 in_user    : The user executing the command
```

```
in_query     : Features of the query under detection

Output
Boolean: true if the query is anomalous, false otherwise
-------------------------------------------------------------
function detect_anomaly(in_user, in_query) returns boolean
{
  // get the role associated with the query
  inp_role = get_user_role(in_user);

  // get the role predicted by the NBC
  map_role = get_map_role(in_user, in_query);

  if (map_role != inp_role)
    return true;

  return false;
}
-------------------------------------------------------------
function get_map_role(in_query) returns role
{
  for each role in the database
  {
     role_prior_prob = calculate_role_prior(role);

     role_likelihood = calculate_role_likeli(role, in_query);

     role_log_aposteriori = role_prior_prob + role_likelihood;

     if (role_log_aposteriori >= max_role_prob)
     {
```

```
35            max_prob_role = role;
36        }
37    }
38
39    return max_prob_role;
40 }
```

Listing 5.1 Anomaly Detection Algorithm in PostgreSQL

Once a SQL command is detected as anomalous by the detection mechanism, it builds an anomaly assessment structure that is then passed on to the response mechanism. The response mechanism takes as input the anomaly assessment structure, searches through the response policy database to find the policies matching the information in the anomaly assessment structure, and then issues a suitable response based on the matched policies. The response mechanism and its implementation strategy was described in detail in Chapter 3.

In what follows, we present experimental results on the overhead of the statistics collection procedure and the detection mechanism on the transaction processing capabilities of the PostgreSQL DBMS.

5.4 Experimental Results

The goal of the experiments described in this section is to measure the overhead introduced by our anomaly detection mechanism on the transaction processing capabilities of the PostgreSQL DBMS. In what follows, we first describe the experimental set-up and then report the results.

5.4.1 Set-up

We use the *pgbench* tool distributed with the PostgreSQL DBMS to run the benchmarking tests [5]. The *pgbench* tool takes as input a script file containing a series of SQL commands and executes the commands as a transaction against the database. In our experiments, every transaction consists of 5 SELECT, 5 INSERT, and 5 UPDATE commands.

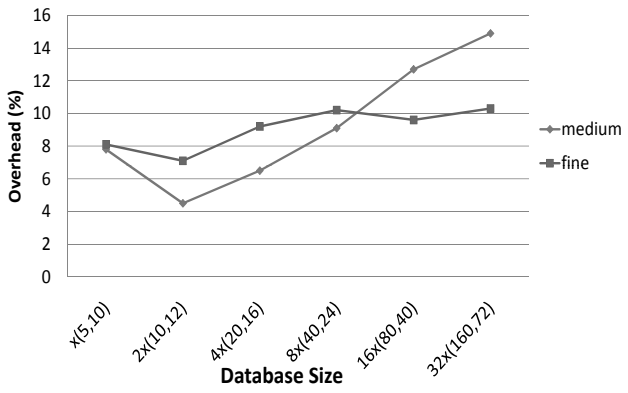


Fig. 5.4. Exp 1: Database Size vs Statistics Collection Overhead

When run for a specified amount of time or for a specific number of transactions, the tool reports the number of transactions per second (tps) processed by the database. In our experiments, we always run the *pgbench* tool for a duration of 60 seconds.

5.4.2 Results

We perform two sets of experiments. In the first set, we run our tests against an increasing size of the database where the database size is measured in terms of the number of tables and number of columns per table in the database. The base database size that we consider is $x = 5$ tables, 10 columns per table. We effectively double the database size for every run of the *pgbench* tool. In addition, for each run, a table is initialized with a constant amount of 100 rows of data. The database is configured with 3 roles and 3 users. Every user is assigned to exactly one role in the database.

Figure 5.4 shows the results for the overhead on the transaction processing capabilities introduced by the statistics collection procedure in this experiment for both the medium and fine quiplets. The overhead is very reasonable for both quiplet types specially considering that we also update the probabilities of each individual quiplet feature every time the statistics are updated.

Figure 5.5 reports the results for the *absolute* time taken by the anomaly detection procedure during the query processing stage. Note that since we maintain the logarithm of probability of every feature during the statistics collection phase, the calculation of *role_likelihood* (see Listing 5.1) during the detection procedure only requires a summation over all the relevant feature probabilities. Considering this fact, the detection time for the medium quiplets is very small as shown in the Figure 5.5.

The detection time for the fine quiplets, however, does increase and becomes noticeable (see the Figure 5.5 when the database size is large. This is primarily because the number of features considered by the detection algorithm for fine-quiplets becomes very large in case of a large database size. For example, considering the database size of $16x$, corresponding to a database of 80 tables and 40 columns per table, gives rise to $80 * 40 = 3200$ features to be considered by the detection algorithm (for calculating likelihood of every role) for the fine quiplets. Thus, if fine quiplets are being used for the anomaly detection procedure, the number of tables in the database that need to be considered for the anomaly detection procedure must be carefully configured so as not to adversely impact the performance of the database.

Since we run the anomaly detection algorithm during the query processing stage, the overhead introduced by the detection algorithm on the throughput of the DBMS (or tps) will depend significantly on the complexity of the transactions. Figure 5.6 shows the results for the overhead on the throughput introduced by the detection algorithm when the transaction consists of a mix of *simple* SELECT, UPDATE and INSERT commands. The number of roles configured in the system is still kept at 3. As shown, the performance of the fine triplets in this case degrades significantly as the database size increases. This is largely due to the fact that the SQL queries themselves are very simple and thus the ratio of the detection time to the actual query processing time is high. Compare this result with the results shown in Figure 5.7 in which the transactions consists of complex SQL queries (with joins). The overhead introduced by the fine triplets in this case is much less since the processing time for the queries themselves is quite high.

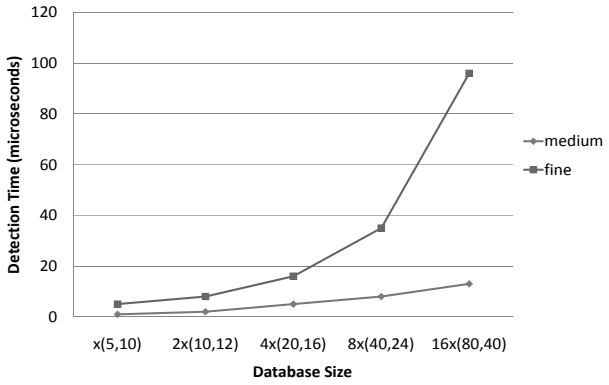


Fig. 5.5. Exp 1: Database Size vs Anomaly Detection Time[2]

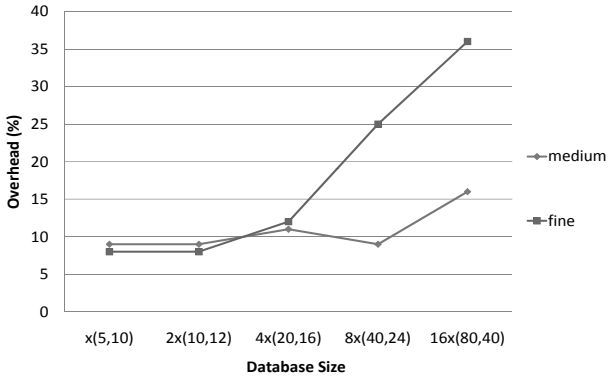


Fig. 5.6. Exp 1: Database Size vs Anomaly Detection Overhead (Simple Queries)[3]

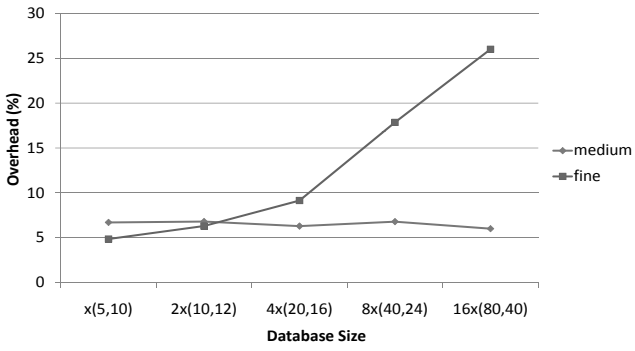


Fig. 5.7. Exp 1: Database Size vs Anomaly Detection Overhead (Join Queries)[5]

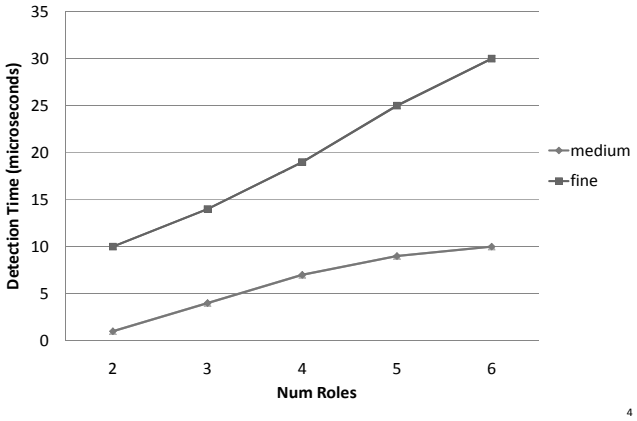


Fig. 5.8. Exp 2: Number of Roles vs Anomaly Detection Time

In the second experiment set, we measure the changes in anomaly detection time by varying the number of roles in the DBMS. The database size in this case is kept at 20 tables and 16 columns per table. Figure 5.8 presents the results for this experiment. As expected, the detection time increases with increasing number of roles because in the detection algorithm we have to calculate *role_likelihood* for every role in the DBMS. However, the performance impact is small since calculation of the *role_likelihood* only requires a summation over the relevant feature probabilities.

Overall, the experimental results show that our anomaly detection procedure integrated with the database query processing mechanism is very efficient and does not have a substantial impact on the transaction processing capabilities of the database.

6. RELATED WORK

In this chapter, we briefly review some of the related work in the area of database intrusion detection and response.

6.1 Database Intrusion Detection

Several approaches dealing with ID for operating systems and networks have been developed [36–40]. However, as we have already argued in Chapter 1, they are not adequate for protecting databases.

An abstract and high-level architecture of a DBMS incorporating an ID component has been recently proposed [41]. However, this work mainly focuses on discussing generic solutions rather than proposing concrete algorithmic approaches. Similar in spirit is the work of Shu et al. [42] who have developed an architecture for securing web-based database systems without proposing any specific ID mechanisms. Finally, in [43] a method for ID is described which is applicable only to real-time applications, such as a programmed stock trading that interacts with a database. The key idea pursued in this work is to exploit the real-time properties of data for performing the ID task.

Anomaly detection techniques for detecting attacks on web applications have been discussed by Vigna et al. [44]. A learning based approach to the detection of SQL attacks is proposed by Valeur et al. [45]. The motivation of this work is similar to ours as in the use of machine learning techniques to detect SQL based attacks on databases. Their methodologies, however, focus on detection of attacks against back-end databases used by web-based applications. Thus, their ID architecture and algorithms are tailored for that context. We, on the other hand, propose a general purpose approach towards detection of anomalous access patterns in a database as represented by SQL queries submitted to the database.

An anomaly detection system for relational databases is proposed by Spalka et al. [46]. This work focuses on detecting anomalies in a particular database state that is represented by the data in the relations. Their first technique uses basic statistical functions to compare reference values for relation attributes being monitored for anomaly detection. The second technique introduces the concept of Δ relations that record the history of changes of data values of monitored attributes between two runs of the anomaly detection system. This work complements our work as it focuses on the semantic aspects of the SQL queries by detecting anomalous database states as represented by the data in the relations, while we focus on the syntactic aspects by detecting anomalous access patterns in a DBMS.

Hu et al. [47] propose an approach for identifying malicious transactions from the database logs. They propose mechanisms for finding data dependency relationships among transactions and use this information to find hidden anomalies in the database log. The rationale of their approach is the following: if a data item is updated, this update does not happen alone but is accompanied by a set of other events that are also logged in the database log files. For example, due to an update of a given data item, other data items may also be read or written. Therefore, each item update is characterized by three sets: the *read set*, the set of items that have been read because of the update; the *pre-write set*, the set of items that have been written before the update but as consequence of it; and the *post-write set*, the set of items that have been written after the update and as consequence of it. They use data mining techniques to generate dependency rules among the data items. These rules are in the following two forms: before a data item is updated, what other data items are read, and after a data item is updated what other data items are accessed by the same transaction. Once these rules are generated, they are used to detect malicious transactions. The transactions that make modifications to the database without following these rules are termed as malicious.

The approach is novel, but its scope is limited to detecting malicious behavior in user transactions. Within that as well, it is limited to user transactions that conform to the read-write patterns assumed by the authors. Also, the system is not able to detect malicious behavior in individual read-write commands.

DEMIDS is a misuse-detection system, tailored for relational database systems [48]. It uses audit log data to derive profiles describing typical patterns of accesses by database users. Essential to such an approach is the assumption that the access pattern of users typically forms a working scope which comprises sets of attributes that are usually referenced together with some values. The idea of working scopes is captured by mining frequent itemsets which are sets of features with certain values. Based on the data structures and integrity constraints encoded in the system catalogs and the user behavior recorded in the audit logs, DEMIDS describes distance measures that capture the closeness of a set of attributes with respect to the working scopes. These distance measures are then used to guide the search for frequent itemsets in the audit logs. Misuse of data, such as tampering with the data integrity, is detected by comparing the derived profiles against the organization's security policies or new audit information gathered about the users. The goal of the DEMIDS system is two-fold. The first goal is detection of malicious insider behavior. Since a profile created by the DEMIDS system is based on frequent sets of attributes referenced by user queries, the approach is able to detect an event when a SQL query submitted by an insider does not conform to the attributes in the user profile. The second goal is to serve as a tool for security re-engineering of an organization. The profiles derived in the training stage can help to refine/verify existing security policies or create new policies. The main drawback of the approach presented as in [48] is a lack of implementation and experimentation. The approach has only been described theoretically, and no empirical evidence have been presented of its performance as a detection mechanism.

Lee et al. [49] present an approach for detecting illegitimate database accesses by fingerprinting transactions. The main contribution of this work is a technique to summarize SQL statements into compact regular expression fingerprints. The system detects an intrusion by matching new SQL statements against a known set of legitimate database transaction fingerprints. In this respect, this work can be classified as a signature-based ID system which is conceptually different from the learning-based approach that we propose in this paper.

In addition to the above approaches, our previous work on query floods [50] can also be characterized as a DBMS-specific ID mechanism. However, in that work we have focused on identifying specific types of intruders, namely those that cause query-flood attacks. A user can engineer such an attack by "flooding" the database with queries that can exhaust DBMS's resources making it incapable of serving legitimate users.

6.2 Database Intrusion Response

Various commercial database monitoring and intrusion detection products are today available on the market [3]. We categorize them into two broad categories: *network-appliance-based* and *agent-based*. Network-appliance-based solutions consist of a dedicated hardware appliance that taps into an organization's network, and monitors network traffic to and from the data center. Agent-based solutions, on the other hand, have a software component installed on the database server that interacts with the DBMS in order to monitor accesses to the data. Each method has its own advantages and disadvantages. Network appliances, in general, are unable to monitor privileged users who can log into the database server directly [3]. Agent-based solutions, on the other hand, result in more overhead because of the additional software running on the database server and its usage of CPU and memory resources. Moreover, as mentioned earlier in Chapter 3, a common shortcoming of these products is their inability to issue a suitable response to an ongoing attack.

Peng Liu et al. have proposed architectures and algorithms for intrusion tolerant databases [41, 51]. Their work focuses on techniques to restore the state of the DBMS to a 'correct' state after rolling back the effects of a malicious transaction. We instead focus on creating a framework for providing a real-time response to a malicious transaction so that the transaction is prevented from being executed.

A taxonomy and survey of intrusion response systems is presented in [52]. According to this taxonomy, our response mechanism may be termed as 'static' *by ability to adjust*, 'autonomous' *by cooperation ability*, 'dynamic mapping' *by response selection method*

and both ‘proactive’ and ‘delayed’ *by time of response*. We direct the reader to [52] for further details on the taxonomy.

Foo et. al. [53] have also presented a survey of intrusion response systems. However, the survey is specific to distributed systems. Since the focus of our work is development of a response mechanism in context of a stand-alone database server, most of the techniques described in [53] are not applicable our scenario.

6.3 Policy Administration

An approach towards addressing the problem of insider threats from malicious DBAs is to apply the *principle of least privilege*. The principle dictates that a user must be assigned only those privileges that are necessary to serve its legitimate purpose. This effectively means to restrict the privileges of the DBAs, and to create new roles for administration of response policy objects. Such approach is followed by Oracle Database using the concept of a *protected schema* for the administration of the *database vault* policies [54]. Database vault is a mechanism introduced by Oracle Database to reduce the risk of insider threats by using policies that prevent the DBAs from accessing application data. The database vault policy objects are themselves stored in the DVSYS protected schema. A protected schema guards the schema against *improper* use of system privileges such as SELECT ANY TABLE, DROP ANY, and so forth. Only the DVDSYS user and *other database vault roles* can have the privileges to modify objects in the DVSYS schema. The powerful ANY system privileges for database definition language and data manipulation language commands are also restricted in the DVSYS protected schema. For further details on the administration model of Oracle Database Vault, we refer the reader to [54]. Note that the Oracle Database Vault and the anomaly response system presented in this paper are both policy-driven mechanisms. Thus, an approach similar to Oracle Database Vault may be followed to administer response policies as well. However, there are some disadvantages in following such approach. First, since the approach is *preventive*, it requires fundamental changes to the existing access control mechanism of a DBMS. For example, the semantics of the

ANY system privilege in the Oracle Database is required to be changed to ANY except the *protected schema objects*. Second, even though the principle of least privilege is a recommended security best practice, it is often not complied with by many organizations. The reason is that such practice requires an organization to invest in additional man-power to assign users to the new roles that can administer the objects in the protected schema. Such strategy is not financially feasible for many organizations, thereby leaving them exposed to the risk of insider threats from malicious DBAs.

A discussion of the related work on threshold signature schemes can be found in [20]. To the best of our knowledge, ours is the first work that applies the technique of threshold signatures for the administration of DBMS objects.

6.4 Policy Matching

The policy matching problem is similar to the event matching problem in content based publish-subscribe (pub-sub) systems [55]. A subscription in a pub-sub system is similar to a response policy, and an event is the anomaly detection event in our system. Many algorithms have been proposed to date for efficient matching of events to subscriptions in pub-sub systems [55–60]. In what follows, we briefly discuss the applicability of such algorithms to the policy matching problem.

An algorithm for event-matching based on the concept of subscription trees is described in context of the GRYPHON project [56]. The algorithm pre-processes the set of subscriptions to build a subscription tree such that each node of the tree is an elementary test on an event attribute. The leaves of the subscription tree are the actual subscriptions. The matching algorithm walks through the subscription tree to find the set of matching subscriptions. Since no analysis of the pre-processing algorithm is provided, it is not clear if the order according to which subscriptions are chosen affects the size of the subscription tree. Also, the scheme is formulated only for elementary predicates, and it has been optimized only for the *equality* predicates. However, for the policy matching problem, we need to consider arbitrary predicates.

Many algorithms for content-based event matching are described by Pereira et al. [57]. The focus of their main algorithm is to improve the cache hit ratio of main memory access, which is not our main concern since we store the policies in the system catalogs, the contents of which are cached by the DBMS in its main memory.

Our base policy matching algorithm is similar to the counting algorithm proposed by Yan et al. [58]. However, we provide an extension to the counting algorithm by pro-actively eliminating predicates that no longer need to be evaluated.

An algorithm for matching predicates in database rule systems using a interval binary tree is proposed by Hanson et al. [60]. The focus of the algorithm is on equality and inequality predicates on totally ordered domains, whereas our policy matching problem need to support arbitrary predicates.

Event matching using Binary Decision Diagrams (BDD) is proposed by Campailla et al. [59]. The scheme considers arbitrary predicates, and also supports disjunctions in the subscription language. We do not need to support disjunctions; thus employing a BDD-based scheme will introduce unnecessary complexity to our response system.

Event matching is also related to the problem of continuous query processing in streaming databases [61]. In continuous query processing, the problem that is addressed is matching multiple streaming tuples, belonging to different relations, to the stored queries. This is different (and much harder) from the policy matching problem in which we only need to match a single tuple (anomaly assessment) to the stored queries (policy conditions).

6.5 State Based Access Control

Access control models have been widely researched in the context of DBMSs [62]. To the best of our knowledge, ours is the first solution formally introducing the concept of privilege states in an access control model.

The implementation of the access control mechanism in the Windows operating system [63], and Network File System protocol V4.1 [64] is similar to the semantics of the *taint* privilege state. In such implementation, the security descriptor of a protected resource

can contain two types of ACLs: a Discretionary Access Control List (DACL), and a System Access Control List (SACL). A DACL is similar to the traditional ACL in that it identifies the principals that are allowed or denied some actions on a protected resource. A SACL, on other hand, identifies the principals and the type of actions that cause the system to generate a record in the security log. In that sense, a SACL ACL entry is similar to a PSAC ACL entry with *taint* privilege state. Our concept of privilege states, however, is more general as reflected by the semantics of the other states introduced in our work.

Much research work has been carried out in the area of network and host based anomaly detection mechanisms [65]. Similarly, much work on intrusion response methods is also in the context of networks and hosts [66, 67]. The fine-grained response actions that we propose in this work are more suitable in the context of application level intrusion detection systems in which there is an end user interacting with the system.

The *up,down*, and *neutral* privilege orientations (in terms of privilege inheritance) have been introduced by Jason Crampton [68]. The main purpose for such privilege orientation in [68] is to show how such scheme can be used to derive a role-based model with multi-level secure policies. However, our main purpose for introducing the privilege orientation modes is to control the propagation of privilege states in a role hierarchy.

7. SUMMARY AND FUTURE RESEARCH DIRECTIONS

7.1 Summary

The main goal of this dissertation is to build an intrusion detection and response mechanism for relational databases integrated with the core database query processing mechanism. Building such a system specifically for databases is important as attacks on databases are semantically different from attacks on the underlying network infrastructure or the host platform. In this context, first we introduce mechanisms for detecting anomalous access patterns of users and roles in a database. The access pattern profiles are created by extracting features from the SQL queries submitted to the database by the users. An intrusion is identified if a query under consideration deviates from the current user (or role) profile being maintained by the system. Second, we extend the detection mechanism with an intrusion response component. The intrusion response component is responsible for issuing a suitable response action to a detected anomalous request. The response component of our system is a policy driven mechanism in which the response policies are pre-configured by the database administrators. The three key issues that we address in the context of the response sub-system are that of response policy matching, response policy administration, and fine-grained response actions. We propose heuristic algorithms for the policy matching task, a joint threshold administration model for the administration of response policies, and a privilege state based access control system for supporting the fine-grained response actions. We also perform a prototype implementation of the role based anomaly detection procedure, the response policy matching algorithms, the joint threshold administration model, and the privilege state based access control mechanism in the PostgreSQL DBMS and report experimental results on the overhead of every implementation. The experimental results show that our approach is not only feasible but also efficient.

In what follows, we describe the possible directions for future research based on the ideas presented in this dissertation.

7.2 Future Research Directions

7.2.1 Detection Mechanism

We have presented two scenarios for the intrusion detection task in databases. For the first scenario, when a role based access control system is in place, we identify role intruders, that is, users that while holding a specific role, behave in a manner that of some other role. The first limitation of our current approach is that we assume the user to activate only one role in a session. This is because we use the naive bayes classification algorithm for predicting the role associated with a query; and only one role can be predicted by the classifier. A possible research direction to extend the scheme is to assume multiple role activation by a user in a session. The classification algorithm, in such case, may need to be enhanced with *bayesian network* based approaches. The second limitation of our approach is that we assume that the roles form a partitioning of the universe of database access behavior. With this assumption, we are not able to identify users that while holding a specific role, behave differently from that role and from any other role in the system. One approach towards identifying such behavior is to train a one class Support Vector Machine (SVM) [69] with the normal role behavior SQL query features. Then any behavior deviating from the normal role behavior learned by the SVM classifier will be identified as anomalous. A similar approach may be adopted for the unsupervised learning scenario for the clusters of similar SQL queries. The one class SVM classifier, trained for every cluster, may be applied to detect SQL queries deviating from their representative cluster.

Apart from the above mentioned research directions, the traditional issues related to application of machine learning techniques to real-world problems are applicable to our approach as well. Such issues include, but are not limited to, the problem of concept drift, the problem of overfitting or underfitting the training data, and so forth.

7.2.2 Response Mechanism

The response mechanism described in this dissertation works on the basis of pre-configured policies. The policies are based on attributes related to the structure of a SQL query and also the context surrounding the query. In this regard, our response mechanism may be considered to be *static* by its ability to adjust. One possible research direction is to come up with more *dynamic* approaches that are suitable for responding to a database intrusion.

An interactive response policy that requires a second factor of authentication provides a second layer of defense when certain anomalous actions are executed against critical system resources such as anomalous access to system catalog tables. This opens the way to new research on how to organize applications to handle such interactions for the case of legacy applications and new applications. In the security area there is a lot work dealing with retrofitting legacy applications for authorization policy enforcement [70]; we believe that such approaches can be extended to support such an interactive approach. For new applications, one can devise methodologies to organize applications that support such interactions. Notice that, however, because our approach is policy-based, the database administrators have the flexibility of designing policies that best fit the way applications are organized.

The joint-threshold administration model described in this dissertation was applied towards administration of response policies. However, the principles behind the model are general enough to be applied to joint administration of any sensitive database operation such as *user creation/modification/deletion*, *grant/revoke of permissions*, and so forth.

The privilege state based access control (PSAC) model may be extended in the following directions. The current version of PSAC does not take into account selective role activation and deactivation within a user session. The PSAC model may be extended with such additional features.

LIST OF REFERENCES

LIST OF REFERENCES

[1] A. Anton, E.Bertino, N.Li, and T.Yu, "A roadmap for comprehensive online privacy policies," in *CERIAS Technical Report*, 2004.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Morgan-Kaufmann, 2002.

[3] A. Conry-Murray, "The threat from within." Network Computing (Aug 2005), http://www.networkcomputing.com/showArticle.jhtml? articleID=166400792.

[4] R. Mogull, "Top five steps to prevent data loss and information leaks." Gartner Research (July 2006), http://www.gartner.com.

[5] "Postgresql 8.3." http://www.postgresql.org/.

[6] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The nist model for role based access control: Towards a unified standard," in *Proceedings of the 5th ACM Workshop on Role Based Access Control*, 2000.

[7] P. Domingos and M. J. Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Machine Learning*, vol. 29, no. 2-3, pp. 103–130, 1997.

[8] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[9] J. Hilden, "Statistical diagnosis based on conditional independence does not require it," *Computers in biology and medicine*, vol. 14, no. 4, pp. 429–435, 1984.

[10] P. Langley, W. Iba, and K. Thompson, "An analysis of bayesian classifiers," in *National Conference on Artificial Intelligence*, pp. 223–228, 1992.

[11] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2-3, pp. 131–163, 1997.

[12] G. F. Cooper, "The computational complexity of probabilistic inference using bayesian belief networks," *Artificical Intelligence*, vol. 42, no. 2-3, pp. 393–405, 1990.

[13] E. Bertino, A. Kamra, and E. Terzi, "Intrusion detection in rbac-administered databases," in *Proceedings of the Applied Computer Security Applications Conference (ACSAC)*, 2005.

[14] Q. Yao, A. An, and X. Huang, "Finding and analyzing database user sessions," in *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.

[15] D. S. Hochbaum and D. B. Shmoys, "A best possible approximation algorithm for the k-center problem," *Mathematics and Operation Research*, vol. 10, pp. 180–184, 1985.

[16] B. Iglewicz and D. C. Hoaglin, *How to Detect and Handle Outliers*. ASQC Quality Press, 1993.

[17] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995.

[18] A. Kamra, E. Bertino, and R. V. Nehme, "Responding to anomalous database requests.," in *Secure Data Management (SDM)*, pp. 50–66, Springer, 2008.

[19] "Oracle database concepts 11g release 1 (11.1).." Online. Available at http://download.oracle.com/docs/cd/B28359_01/server.111/ b28318/datadict.htm. 03 Jul 2009.

[20] V. Shoup, "Practical threshold signatures.," in *EUROCRYPT*, pp. 207–220, 2000.

[21] R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk, "Robust and efficient sharing of rsa functions.," *Journal of Cryptology*, vol. 20, no. 3, p. 393, 2007.

[22] D. Kincaid and W. Cheney, *Numerical Analysis: Mathematics of Scientific Computing.* Brooks Cole, 2001.

[23] "Openpgp message format. rfc 4800.." Online. Available at http://www.ietf.org/rfc/rfc4880.txt. 03 Jul 2009.

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 2001.

[25] C. K. Koc, "High-speed rsa implementation. tr-201, version 2.0.," tech. rep., RSA Laboratories, 1994.

[26] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The nist model for role-based access control: Towards a unified standard," in *ACM workshop on Role-based access control*, pp. 47–63, 2000.

[27] R. Chandramouli and R. Sandhu, "Role based access control features in commercial database management systems," in *Proceedings of 21st National Information Systems Security Conference*, 1998.

[28] "Oracle database security guide 11g release 1 (11.1).." Online. Available at http://download.oracle.com/docs/cd/B28359_01/ network.111/b28531/toc.htm. 02 Jan 2009.

[29] "Sql server 2008 books online. identity and access control (database engine).." Online. Available at http://msdn.microsoft.com/en-us/library/bb510418(SQL.100).aspx. 02 Jan 2009.

[30] E. Bertino, P. Samarati, and S. Jajodia, "An extended authorization model for relational databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 1, pp. 85–101, 1997.

[31] "Sql-99 standard. incits/iso/iec 9075.." Online. Available at http://webstore.ansi.org/. 02 Feb 2009.

[32] "Postgresql 8.3 documentation. postgresql and global and development and group." Online. Available at http://www.postgresql.org/docs/8.3/static/sql-grant.html. 02 Feb 2009.

[33] "University postgres 4.2." Online. Available at http://db.cs.berkeley.edu//postgres.html. 18 Apr 2010.

[34] "Postgresql - wikipedia, the free encyclopedia." Online. Available at http://en.wikipedia.org/wiki/PostgreSQL. 18 Apr 2010.

[35] "How postgresql processes a query." Bruce Momjian. http://anoncvs.postgresql.org/cvsweb.cgi/ checkout /pgsql/src/tools/backend/index.html. 03 Jul 2008.

[36] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Tech. Rep. 99-15, Chalmers Univ., Mar. 2000.

[37] K. H. A. Hoglund and A. Sorvari, "A computer host-based user anomaly detection using the self-organizing map," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN)*, 2000.

[38] T. Lane and C. E. Brodley, "Temporal sequence learning and data reduction for anomaly detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 3, pp. 295–331, 1999.

[39] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. Neumann, H. Javitz, A. Valdes, and T. Garvey, "A real - time intrusion detection expert system (ides) - final technical report," *Technical Report, Computer Science Laboratory, SRI International*, 1992.

[40] R. Talpade, G. Kim, and S. Khurana, "Nomad: Traffic-based network monitoring framework for anomaly detection," in *Proceedings of the 4th IEEE Symposium on Computers and Communications (ISCC)*, 1998.

[41] P. Liu, "Architectures for intrusion tolerant database systems," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2002.

[42] S. Wenhui and T. Tan, "A novel intrusion detection system model for securing web-based database systems," in *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC)*, 2001.

[43] V. Lee, J. Stankovic, and S. Son, "Intrusion detection in real-time databases via time signatures," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2000.

[44] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.

[45] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *Proceedings of the International Conference on detection of intrusions and malware, and vulnerability assessment (DIMVA)*, 2003.

[46] A. Spalka and J. Lehnhardt, "A comprehensive approach to anomaly detection in relational databases.," in *DBSec*, pp. 207–221, 2005.

[47] Y. Hu and B. Panda, "A data mining approach for database intrusion detection," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, (New York, NY, USA), pp. 711–716, ACM, 2004.

[48] C. Chung, M. Gertz, and K. Levitt, "Demids: a misuse detection system for database systems," in *Integrity and Internal Control in Information Systems: Strategic Views on the Need for Control. IFIP TC11 WG11.5 Third Working Conference*, 2000.

[49] S. Y. Lee, W. L. Low, and P. Y. Wong, "Learning fingerprints for a database intrusion detection system," in *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, (London, UK), pp. 264–280, Springer-Verlag, 2002.

[50] E. Bertino, T. Leggieri, and E. Terzi, "Securing dbms: Characterizing and detecting query floods," in *Proceedings of the International Security Conference (ISC)*, 2004.

[51] P. Ammann, S. Jajodia, and P. Liu, "Recovery from malicious transactions," *IEEE Transanctions on Knowledge and Data Engineering (TKDE)*, vol. 14, no. 5, pp. 1167–1185, 2002.

[52] N. Stakhanova, S. Basu, and J. Wong, "A taxonomy of intrusion response systems," *International Journal of Information and Computer Security (IJICS)*, vol. 1, no. 2, pp. 169–184, 2007.

[53] B. Foo, M. Glause, G. Modelo-Howard, Y.-S. Wu, S. Bagchi, and E. H. Spafford, *Information Assurance: Dependability and Security in Networked Systems*. Morgan Kaufmann, 2007.

[54] "Oracle database vault administrator's guide 11g release 1 (11.1).." Online. Available at http://download.oracle.com/docs/cd/B28359_01/ server.111/b31222/toc.htm. 02 Jan 2009.

[55] F. Fabret, F. Llirbat, J. A. Pereira, I. Rocquencourt, and D. Shasha, "Efficient matching for content-based publish/subscribe systems.," tech. rep., INRIA, 2000.

[56] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system.," in *Symposium on Principles of Distributed Computing (PODC)*, (New York, NY, USA), pp. 53–61, ACM, 1999.

[57] J. A. Pereira, F. Fabret, F. Llirbat, and D. Shasha, "Efficient matching for web-based publish/subscribe systems.," in *International Conference on Cooperative Information Systems (CooplS)*, (London, UK), pp. 162–173, Springer-Verlag, 2000.

[58] T. W. Yan and H. García-Molina, "Index structures for selective dissemination of information under the boolean model," *ACM Transactions on Database Systems (TODS)*, vol. 19, no. 2, pp. 332–364, 1994.

[59] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams.," in *International Conference on Software Engineering (ICSE)*, (Washington, DC, USA), pp. 443–452, IEEE Computer Society, 2001.

[60] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang, "A predicate matching algorithm for database rule systems.," *SIGMOD*, vol. 19, no. 2, pp. 271–280, 1990.

[61] H.-S. Lim, J.-G. Lee, M.-J. Lee, K.-Y. Whang, and I.-Y. Song, "Continuous query processing in data streams using duality of data and queries," in *SIGMOD*, (New York, NY, USA), pp. 313–324, ACM, 2006.

[62] E. Bertino and R. S. Sandhu, "Database security-concepts, approaches, and challenges.," *IEEE Trans. Dependable Sec. Comput.*, vol. 2, no. 1, pp. 2–19, 2005.

[63] "Access control lists in win32.." Online. Available at http://msdn.microsoft.com/en-us/library/aa374872(VS.85).aspx. 07 Jun 2009.

[64] "Nfs version 4 minor version 1." Online. Available at http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-29.txt. 07 Jun 2009.

[65] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends.," *Computer Networks*, vol. 51, no. 12, pp. 3448–3470, 2007.

[66] A. Somayaji and S. Forrest, "Automated response using system-call delays," *Proceedings of the 9th USENIX Security Symposium (USENIX Association; Berkeley, CA)*, p. 185, 2000.

[67] T. Toth and C. Krügel, "Evaluating the impact of automated intrusion response mechanisms," pp. 301–310, IEEE Computer Society, 2002.

[68] J. Crampton, "Understanding and developing role-based administrative models," in *ACM Conference on Computer and Communications Security*, pp. 158–167, 2005.

[69] L. M. Manevitz and M. Yousef, "One-class svms for document classification," *Journal of Machine Learning Research*, vol. 2, pp. 139–154, 2002.

[70] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting legacy code for authorization policy enforcement.," in *IEEE Symposium on Security and Privacy*, pp. 214–229, IEEE Computer Society, 2006.

VITA

VITA

Ashish Kamra was born in Bikaner, Rajasthan (India) in the year 1979. He completed his schooling from various central schools in India finally settling in Delhi from Class VIII onwards. After finishing high school from The Air Force School (TAFS), Delhi in the year 1997, Ashish got admitted to the Visvesvaraya Regional College of Engineering (now VNIT), Nagpur, Maharashtra (India) for his under-graduate studies. He obtained his Bachelor's Degree in Electronics Engineering from VNIT in the year 2001.

From 2001 to 2004, Ashish worked in Tata Consultancy Services (TCS) in capacity of a systems engineer. During his stint at TCS, Ashish worked on one of the pioneering e-governance projects in India called "Secretariat Knowledge and Information Management System" (SKIMS, later names smartGOV). The SKIMS project aimed at creating a workflow cum document management system for processing of "files" in a state secretariat.

Ashish joined Purdue University in the Fall of 2004 for the Interdisciplinary Masters in Information Security program offered by CERIAS in collaboration with the College of Technology. He then enrolled in the direct PhD program of the Electrical and Computer Engineering department in the Fall of 2005 under the guidance of Prof. Elisa Bertino.

Ashish did an internship with the Rainfinity group at EMC^2 from May 2007 to Aug 2007. He did another internship with the same group from May 2009 to Dec 2009. Since Jan 2010, he has been working in the Integrated Systems and Components group of the Unified Storage Division at EMC^2. Ashish received his doctorate in Computer Engineering in Aug 2010 with Prof. Arif Ghafoor and Prof. Elisa Bertino as the major advisers. Ashish's research interests are in the area of intrusion detection and prevention technologies, application of machine learning techniques to computer security, and insider threat modeling, detection and prevention.