Design and Implementation of an Intrusion Response System for Relational Databases

Article *in* IEEE Transactions on Knowledge and Data Engineering · June 2011

DOI: 10.1109/TKDE.2010.151 · Source: DBLP

2 authors, including:

Elisa Bertino
Purdue University
1,102 PUBLICATIONS **21,569** CITATIONS

SEE PROFILE

Design and Implementation of an Intrusion Response System for Relational Databases

Ashish Kamra and Elisa Bertino, *Fellow*, *IEEE*

Abstract—The intrusion response component of an overall intrusion detection system is responsible for issuing a suitable response to an anomalous request. We propose the notion of *database response policies* to support our intrusion response system tailored for a DBMS. Our interactive response policy language makes it very easy for the database administrators to specify appropriate response actions for different circumstances depending upon the nature of the anomalous request. The two main issues that we address in context of such response policies are that of *policy matching*, and *policy administration*. For the policy matching problem, we propose two algorithms that efficiently search the policy database for policies that match an anomalous request. We also extend the PostgreSQL DBMS with our policy matching mechanism, and report experimental results. The experimental evaluation shows that our techniques are very efficient. The other issue that we address is that of administration of response policies to prevent malicious modifications to policy objects from legitimate users. We propose a novel *Joint Threshold Administration Model* (JTAM) that is based on the principle of separation of duty. The key idea in JTAM is that a policy object is jointly administered by at least k database administrator (DBAs), that is, any modification made to a policy object will be invalid unless it has been authorized by at least k DBAs. We present design details of JTAM which is based on a cryptographic threshold signature scheme, and show how JTAM prevents malicious modifications to policy objects from authorized users. We also implement JTAM in the PostgreSQL DBMS, and report experimental results on the efficiency of our techniques.

Index Terms—Databases, intrusion detection, response, prevention, policies, threshold signatures.

1 Introduction

Recently, we have seen an interest in products that continuously monitor a database system and report any relevant suspicious activity [1]. Database activity monitoring has been identified by Gartner research as one of the top five strategies that are crucial for reducing data leaks in organizations [2], [3]. Such step-up in data vigilance by organizations is partly driven by various US government regulations concerning data management such as SOX, PCI, GLBA, HIPAA, and so forth [4]. Organizations have also come to realize that current attack techniques are more sophisticated, organized, and targeted than the broad-based hacking days of past. Often, it is the sensitive and proprietary data that is the real target of attackers. Also, with greater data integration, aggregation and disclosure, preventing data theft, from both inside and outside organizations, has become a major challenge. Standard database security mechanisms, such as access control, authentication, and encryption, are not of much help when it comes to preventing data theft from insiders [5]. Such threats have thus forced organizations to reevaluate security strategies for their internal databases [4]. Monitoring a database to detect potential intrusions, *intrusion detection* (ID), is a crucial technique that has to be part of any comprehensive security solution for high-assurance database security. Note that the ID systems that are developed must be tailored for a Database Management System (DBMS) since database-related attacks such as SQL injection and data exfiltration are not malicious for the underlying operating system or the network.

Our approach to an ID 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 database-access profiles of roles and users, and on the use of such profiles for the AD task. A user-request that does not conform to the normal access profiles is characterized as anomalous. Profiles can record information of different levels of details; we refer the reader to [6] for additional information and experimental results. The second element of our approach—the focus of this paper—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 [7], [8]. 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.

- *A. Kamra is with the School of Electrical and Computer Engineering, Purdue University, 1085 Tasman Dr Spc 15, Sunnyvale, CA 94089. E-mail: akamra@purdue.edu.*
- *E. Bertino is with the Department of Computer Science and the Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University, 305 N. University Street, West Lafayette, IN 47907-2107. E-mail: bertino@cs.purdue.edu.*

Manuscript received 19 Feb. 2009; revised 5 July 2009; accepted 30 Oct. 2009; published online 27 Aug. 2010.
Recommended for acceptance by K.-L. Tan.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2009-02-0078.
Digital Object Identifier no. 10.1109/TKDE.2010.151.

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 security administrator to specify appropriate response actions for different circumstances. In this paper, 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 Section 4, 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 [9], 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 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 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 multiple 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 nonrepudiable, 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.

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

1. We present a framework for specifying intrusion response policies in the context of a DBMS.
2. We present a novel administration model called JTAM for administration of response policies.
3. We present algorithms to efficiently search the policy database for policies that match an anomalous request.
4. We extend the PostgreSQL DBMS with our response policy mechanism, and conduct an experimental evaluation of our techniques.

The rest of the paper is organized as follows: Section 2 presents the details of the response policy language. Section 3

presents the design and implementation details of JTAM. We discuss the policy matching algorithms in Section 4. Section 5 discusses the implementation details of our response mechanism, and reports the experimental results concerning the overhead incurred by our techniques. Section 6 discusses related work. We conclude in Section 7 with directions for future work.

2 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 [10]. An ECA rule is typically organized as follows:

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

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.

2.1 Attributes and Conditions

2.1.1 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 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.

TABLE 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.

2.1.2 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 (cf. Section 4), 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 (Policy Condition). *Let $PA = \{A_1, A_2 \dots A_n\}$ 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 $Pr_k, Pr_l \dots Pr_m$ are predicates of type Pr.*

2.2 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 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 proactively prevent an intrusion. *Aggressive* actions, on the other hand, are high severity responses. Such actions are capable of preventing an intrusion proactively 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

TABLE 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.

of the user and possibly in the suspension or dropping of subsequent requests by the same user. We refer the reader to [7] 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 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 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 one aggressively prevents against such attacks by disconnecting the user.

Policy two 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.

2.3 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 2.2, 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 4. The initial action is to suspend the anomalous user request. As a confirmation action, the user is prompted for reauthentication. 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.

TABLE 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*

TABLE 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*

3 Policy Administration

As discussed in Section 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

1. Note that implementing the confirmation actions such as a reauthentication 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.

propose an administration model referred to as the **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 [12]. The basic paradigm of most well-known threshold signature schemes is as follows [13]: 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, 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 [12], 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 [14] 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.

3.1 JTAM Setup

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 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,

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 [9]. In Oracle 11*g* Database, the system catalogs may be updated by users holding the SYS account [11].

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, databse configuration, etc since these tasks do not lead to the kind of *conflict-of-interest* that we address in the paper.

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 [15] to generate high-entropy keys from the passwords, then use such keys to encrypt the secret shares.

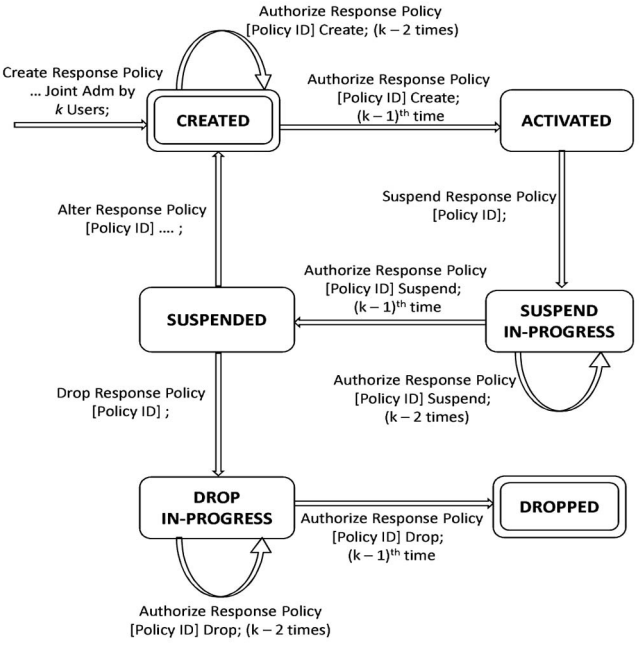

Fig. 1. Policy state transition diagram.

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.

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 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 lifecycle of a policy object are policy *creation*, *activation*, *suspension*, *alteration*, and *deletion*. The lifecycle is shown in Fig. 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 nonoperational, 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 lifecycle with the policy creation procedure.

3.2.1 Policy Creation

The policy creation command has the following format:

$$\text{Create Response Policy } [\textit{Policy Data}] \text{ Jointly Administered By } k \text{ Users;}$$

Policy Data refers to the interactive ECA response policy conditions and actions that were described in Section 2. 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 three users (including the owner). We define an *administrator* 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 (cf. Section 2) that need to be protected from malicious modifications. Thus,

$$H(Pol) = SHA1(PolicyID, Conditions, InitialAction(s), OptionalAction(s), k, State). \quad (1)$$

   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 [16].

The policy data along with the signature share and $H(Pol)$ is stored in the *sys_response_policy* system catalog as

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

TABLE 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					

shown in Table 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 $k-1=2$.

3.2.2 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:

$$\text{Authorize Response Policy } [Policy\ ID] \text{ Create;}$$

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 6.
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* $[PolicyID]$ *Create* command. When $k-1$ administrators have authorized the policy for activation, the signature combining and verification algorithms are executed (Section 3.2.3). 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 7.

3.2.3 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

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

where the λs are the integers defined as follows:

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

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

TABLE 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					

These values of λ are derived from the standard Lagrange polynomial interpolation formula [14]. 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 [16]. 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' [16].

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 (1). If $(W_{final})^e=H(Pol)$, the signature is valid otherwise not.

3.2.4 Policy Suspension

To alter/drop a policy or to make it nonoperational, 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

TABLE 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}		

TABLE 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}		

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 8.

3.2.5 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 (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.

3.2.6 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 (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 reactivated after it has been dropped.

3.3 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.

3.3.1 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.

3.3.2 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 [17]. 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 65,537 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.

3.3.3 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 (1)) are hashed and signed; therefore any modification to such attributes through the SQL update command will invalidate the final signature on the policy.

3.3.4 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

TABLE 9
Sys_Response_Policy_Count Catalog after Policy Creation

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

of policy tuples present in the database. It increments such policy-count by one 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 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.

3.3.5 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.

3.3.6 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.

TABLE 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

4 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 (cf. Definition 2.1). 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 toward 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 bufferpool. Assume a policy database consisting of four anomaly attributes, six policy predicates and four policies as shown in Table 10. The graph shown in Fig. 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

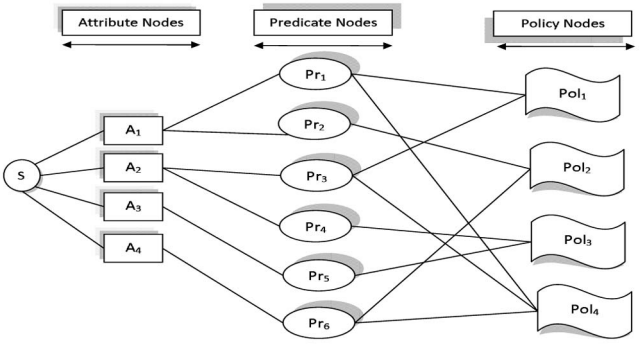

Fig. 2. Policy predicate graph example.

added (denoted by s in Fig. 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 toward policy matching. There are two variations of our policy matching algorithm. The first algorithm, called the *Base Policy Matching* algorithm, is described next.

4.1 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 one. A policy is matched when its *predicate-match-count* becomes equal to the number of predicates 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.

4.2 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 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 runtime of the policy matching procedure.

4.3 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 2.2 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.

9. The *policy-match-count* of a predicate is the number of noninvalidated policies that the predicate belongs to.

TABLE 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.

5 Implementation and Experiments

We have extended the PostgreSQL 8.3 open source DBMS [9] 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 catalog tables that store the response policy data. The catalogs and their purposes are described in Table 11. We have instrumented the query processing subsystem 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.

5.1 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 setup, and then report the evaluation results.

5.1.1 Setup

We use the following six anomaly attributes for our experimental evaluation: *User*, *Client App*, *Source IP*, *Database*, *Objs*, and *SQLCmd* (see Table 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.

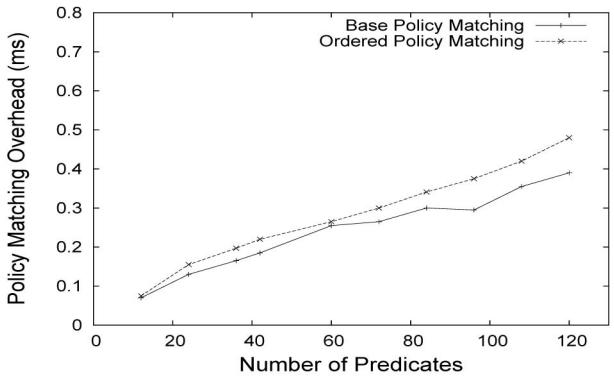

Fig. 3. Experiment 1: Number of predicates versus policy matching overhead.

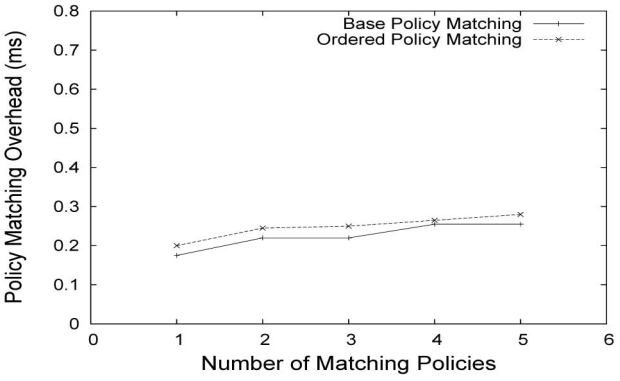

Fig. 5. Experiment 2: Number of matching policies versus policy matching overhead.

5.1.2 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 four. The number of predicates, and correspondingly the number of policies, are varied in order to assess the policy matching overhead time. Fig. 3 shows the policy matching overhead for the two algorithms as a function of the number of predicates. Fig. 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. Fig. 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. Fig. 6 reports the variation in the number of predicates skipped by varying the number 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.

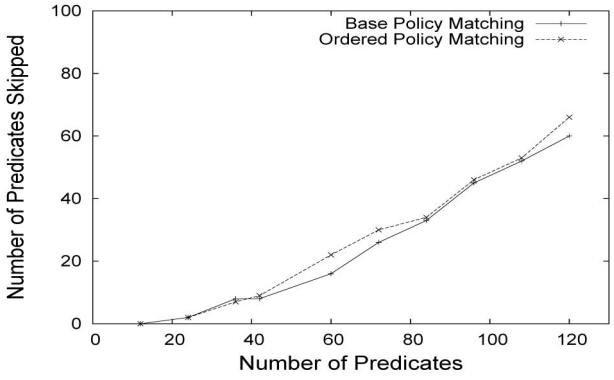

Fig. 4. Experiment 1: Number of predicates versus number of predicates skipped.

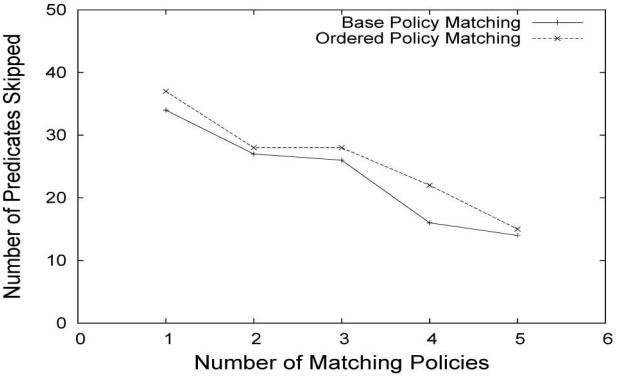

Fig. 6. Experiment 2: Number of matching policies versus number of predicates skipped.

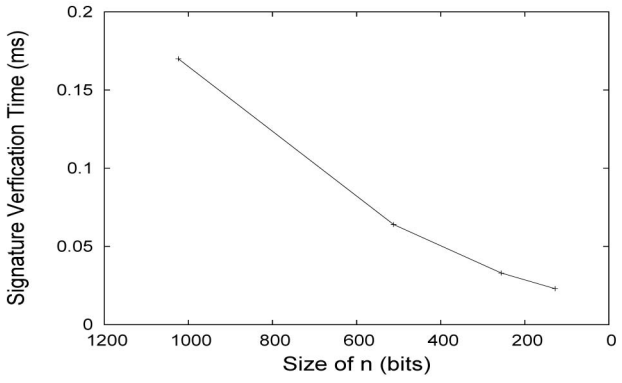

Fig. 7. Size of n (bits) versus signature verification overhead for a single policy.

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 1,024 bits. For such setup, the signature verification overhead for a single policy 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 3.3, 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 Fig. 7). Such strategy, however, is not recommended since 1,024 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.

6 Related Work

The concept of database response policies was first introduced by us [7]. The current paper is a major extension of our previous work. The policy matching algorithms in the current paper take into account arbitrary predicates while the scheme in [7] only considers equality predicates. Also, the JTAM policy administration model presented in this paper is a novel contribution.

The concept of fine-grained response actions such as *suspend*, and *taint* has been introduced by us [8]. However, the work in [8] presents the design and implementation of an access control model that is capable of supporting such fine-grained response actions.

Another approach toward 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 [18]. 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 [18]. 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 an 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 manpower 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 [12]. To the best of our knowledge, ours is the first work that applies the technique of threshold signatures for the administration of DBMS objects.

The policy matching problem is similar to the event matching problem in content based publish-subscribe (pub-sub) systems [19]. 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 [19], [20], [21], [22], [23], [24]. 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 [20]. The algorithm preprocesses 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 preprocessing 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. [21]. 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. [22]. However, we provide an extension to the counting algorithm by proactively 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. [24]. 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. [23]. 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 [25]. 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).

7 Conclusion and Future Work

In this paper, we have described the response component of our intrusion detection 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 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.

We plan to extend our work on the following lines. An interactive response policy that requires a second factor of authentication will provide 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 [26]; 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 DBAs have the flexibility of designing policies that best fit the way applications are organized.

We are currently in the process of implementing the intrusion detection algorithms in the PostgreSQL DBMS as part of our overall intrusion detection and response system in a DBMS. As part of the future work, we intend to report results on the overhead of the entire system on the transaction processing capabilities of the DBMS.

Acknowledgments

The authors would like to thank Abhilasha Bhargav-Spantzel, Rahim Sewani, and Sarvjeet Singh for sharing the threshold cryptography library.

References

[1] A. Conry-Murray, "The Threat from within. Network Computing (Aug. 2005)," http://www.networkcomputing.com/showArticle.jhtml?articleID=166400792, July 2009.
[2] R. Mogull, "Top Five Steps to Prevent Data Loss and Information Leaks. Gartner Research (July 2006)," http://www.gartner.com, 2010.
[3] M. Nicolett and J. Wheatman, "Dam Technology Provides Monitoring and Analytics with Less Overhead. Gartner Research (Nov. 2007)," http://www.gartner.com, 2010.
[4] R.B. Natan, *Implementing Database Security and Auditing*. Digital Press, 2005.
[5] D. Brackney, T. Goan, A. Ott, and L. Martin, "The Cyber Enemy within ... Countering the Threat from Malicious Insiders," *Proc. Ann. Computer Security Applications Conf. (ACSAC)*. pp. 346-347, 2004.
[6] A. Kamra, E. Terzi, and E. Bertino, "Detecting Anomalous Access Patterns in Relational Databases," *J. Very Large DataBases (VLDB)*, vol. 17, no. 5, pp. 1063-1077, 2008.
[7] A. Kamra, E. Bertino, and R.V. Nehme, "Responding to Anomalous Database Requests," *Secure Data Management*, pp. 50-66, Springer, 2008.
[8] A. Kamra and E. Bertino, "Design and Implementation of SAACS: A State-Aware Access Control System," *Proc. Ann. Computer Security Applications Conf. (ACSAC)*, 2009.
[9] "Postgresql 8.3. The Postgresql Global Development Group," http://www.postgresql.org/, July 2008.
[10] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995.
[11] "Oracle Database Concepts 11g Release 1 (11.1)," http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/datadict.htm, July 2009.
[12] V. Shoup, "Practical Threshold Signatures," *Proc. Int'l Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pp. 207-220, 2000.
[13] R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk, "Robust and Efficient Sharing of RSA Functions," *J. Cryptology*, vol. 20, no. 3, pp. 393-400, 2007.
[14] D. Kincaid and W. Cheney, *Numerical Analysis: Mathematics of Scientific Computing*. Brooks Cole, 2001.
[15] "Openpgp Message Format. rfc 4800," http://www.ietf.org/rfc/rfc4880.txt, July 2009.
[16] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001.

[17] C.K. Koc, "High-Speed RSA Implementation," Technical Report tr-201, Version 2.0, RSA Laboratories, 1994.
[18] "Oracle Database Vault Administrator's Guide 11g Release 1 (11.1)," http://download.oracle.com/docs/cd/B28359_01/server.111/b31222/toc.htm, Jan. 2009.
[19] F. Fabret, F. Llirbat, J.A. Pereira, I. Rocquencourt, and D. Shasha, "Efficient Matching for Content-Based Publish/Subscribe Systems," technical report, INRIA, 2000.
[20] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra, "Matching Events in a Content-Based Subscription System," *Proc. Symp. Principles of Distributed Computing (PODC)*, pp. 53-61, 1999.
[21] J.A. Pereira, F. Fabret, F. Llirbat, and D. Shasha, "Efficient Matching for Web-Based Publish/Subscribe Systems," *Proc. Int'l Conf. Cooperative Information Systems (CoopIS)*, pp. 162-173, 2000.
[22] T.W. Yan and H. García-Molina, "Index Structures for Selective Dissemination of Information under the Boolean Model," *ACM Trans. Database Systems*, vol. 19, no. 2, pp. 332-364, 1994.
[23] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient Filtering in Publish-Subscribe Systems Using Binary Decision Diagrams," *Proc. Int'l Conf. Software Eng. (ICSE)*, pp. 443-452, 2001.
[24] E.N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang, "A Predicate Matching Algorithm for Database Rule Systems," *Proc. ACM SIGMOD*, vol. 19, no. 2, pp. 271-280, 1990.
[25] 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," *Proc. ACM SIGMOD*, pp. 313-324, 2006.
[26] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting Legacy Code for Authorization Policy Enforcement," *Proc. IEEE Symp. Security and Privacy*, pp. 214-229, 2006.

Ashish Kamra received the BE degree in electronics from VNIT, India, in 2001. He is currently working toward the PhD degree in the School of Electrical and Computer Engineering at Purdue University and is a member of the Center for Education and Research in Information Assurance and Security (CERIAS). His research interests include database security, intrusion detection and response, and protection of computer systems from insider threats.

Elisa Bertino is professor of Computer Science at Purdue University and serves as research director of the Center for Education and Research in Information Assurance and Security (CERIAS). Previously, she was a faculty member in the Department of Computer Science and Communication of the University of Milan, where she was the department head and director of the DB&SEC laboratory. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation, at Rutgers University, and at Telcordia Technologies. Her main research interests include security, privacy, digital identity management systems, database systems, distributed systems, and multimedia systems. In those areas, she has published more than 400 papers in all major refereed journals, and in proceedings of international conferences and symposia. She is a coauthor of the books *Object-Oriented Database Systems—Concepts and Architectures* (Addison-Wesley International Publ., 1993), *Indexing Techniques for Advanced Database Systems* (Kluwer Academic Publishers, 1997), *Intelligent Database Systems* (Addison-Wesley International Publ., 2001), and *Security for Web Services and Service Oriented Architectures* (Springer, 2009). She has been a co-editor-in-chief of the *Very Large Database Systems (VLDB) Journal* from 2001 to 2007. She serves, or has served, on the editorial boards of several scientific journals, incuding the *IEEE Internet Computing*, the *IEEE Security and Privacy*, the *IEEE Transactions on Knowledge and Data Engineering*, the *ACM Transactions on Information and System Security*, the *ACM Transactions on Web*, *Acta Informatica*, and the *Parallel and Distributed Database Journal*. She is a fellow of the IEEE and a fellow of the ACM and has been been named a Golden Core member for her service to the IEEE Computer Society. She received the 2002 IEEE Computer Society Technical Achievement Award "for outstanding contributions to database systems and database security and advanced data management systems" and the 2005 IEEE Computer Society Tsutomu Kanai Award "for pioneering and innovative research contributions to secure distributed systems." She is currently serving on the Board of Governors of the IEEE Computer Science Society.