DAIS: A Real-time Data Attack Isolation System for Commercial Database Applications*

Peng Liu
Department of Information Systems, UMBC
Baltimore, MD 21250
pliu@umbc.edu

Abstract

Traditional database security mechanisms are very limited in defending successful data attacks. Authorized but malicious transactions can make a database useless by impairing its integrity and availability. This paper presents the design of a real-time data attack isolation system, denoted DAIS. DAIS isolates likely suspicious actions before a definite determination of intrusion is reported. In this way, the database can be immunized from many malicious transactions. DAIS is a COTS-DBMS-specific implementation of a general isolation algorithm that we developed in [19]. In this paper, the design of the first DAIS prototype, which is for Oracle Server 8.1.6, is discussed. DAIS uses triggers and transaction profiles to keep track of the items read and written by transactions, isolates attacks by rewriting user SQL statements, and is transparent to end users. The DAIS design is very general. In addition to Oracle, it can be easily adapted to support many other database application platforms such as Microsoft SQL Server, Sybase, and Informix.

Keywords: Isolation, Intrusion Tolerance, Database Security

1 Introduction

Database security concerns the confidentiality, integrity, and availability of data stored in a database. A broad span of research from authorization [9, 28, 14], to inference control [1], to multilevel secure databases [33, 31], and to multilevel secure transaction processing [3], addresses primarily how to protect the security of a database, especially its confidentiality. However, very limited research has been done on how to survive successful database attacks, which can seriously impair the integrity and availability of a database. Experience with data-intensive applications such as credit card billing, banking, air traffic control, logistics management, inventory tracking, and online stock trading, has shown that a variety of attacks do succeed to fool traditional database protection mechanisms. In fact, we must recognize that not all attacks – even obvious ones – can be averted at their outset. Attacks that succeed, to some degree at least, are unavoidable. With cyber attacks on data-intensive internet applications, i.e., e-commerce systems, becoming an ever more serious threat to our economy, society, and everyday lives, attack resistant database systems that can survive malicious attacks are a significant concern.

One critical step towards attack resistant database systems is intrusion detection, which has attracted many researchers [7, 21, 13, 10, 23, 26, 22, 17, 18]. Intrusion detection systems monitor system or network activity to discover attempts to disrupt or gain illicit access to systems. The methodology of intrusion detection can be roughly classed as being either based on *statistical profiles* [15, 16, 30] or on known patterns of attacks, called *signatures* [11, 8, 27, 12, 32]. Intrusion detection can supplement protection of network and information systems by rejecting the future access of detected attackers and by providing useful hints on how to strengthen the defense. However, intrusion detection has several inherent limitations: (a) Intrusion detection makes the system attack-aware but not attack-resistant, that is, intrusion detection itself cannot maintain the integrity and availability of the database in face of attacks. (b) Achieving accurate detection is usually difficult or expensive. The *false alarm rate* is high in many cases. (c) The average detection latency in many cases is too long to effectively confine the damage.

To overcome the limitations of intrusion detection, a broader perspective is introduced, saying that in addition to detecting attacks, countermeasures to these successful attacks should be planned and deployed in advance. In the literature, this is referred to as *survivability* or *intrusion tolerance*. In this paper, we will address an useful technique for database intrusion tolerance, namely *attack isolation*, and present the design of a practical system which can do attack isolation.

1.1 The Problem

The merit of attack isolation can only be clearly explained in the context of an intrusion tolerant database sys-

tem. Database intrusion tolerance can mainly be enforced at two possible levels: *operating system (OS) level* and *transaction level*. Although transaction level methods cannot handle OS level attacks, it is shown that in many applications where attacks are enforced mainly through malicious transactions transaction level methods can tolerate intrusions in a much more effective and efficient way. Moreover, it is shown that OS level intrusion tolerance techniques such as those proposed in [23, 22, 24, 25, 4], can be directly integrated into a transaction level intrusion tolerance framework to complement it with the ability to tolerate OS level attacks. This paper will focus on transaction level intrusion tolerance, and our presentation will be based on the intrusion tolerant database system architecture shown in Figure 1.

The architecture is built on top of a traditional COTS (Commercial-Of-The-Shelf) DBMS. Within the framework, the *Intrusion Detector* identifies malicious transactions based on the history kept (mainly) in the log. The *Damage Assessor* locates the damage caused by the detected transactions. The *Damage Repairer* repairs the located damage using some specific *cleaning* transactions. The *Damage Confinement Manager* restricts the access to the objects that have been identified by the Damage Assessor as damaged, and unconfines an object after it is cleaned. The *Policy Enforcement Manager* (PEM) (a) functions as a *proxy* for normal user transactions and those *cleaning* transactions, and (b) is responsible for enforcing system-wide intrusion tolerant policies. For example, a policy may require the PEM to reject every new transaction submitted by an user as soon as the Intrusion Detector finds that a malicious transaction is submitted by the user. It should be noticed that the framework is designed to do all the intrusion tolerance work on-the-fly without the need to periodically halt normal transaction processing.

The complexity of the framework is mainly caused by a phenomenon denoted *damage spreading*. In a database, the results of one transaction can affect the execution of some other transactions. Informally, when a transaction T_i reads an object x updated by another transaction T_j, T_i is directly *affected* by T_j. If a third transaction T_k is affected by T_i, but not directly affected by T_j, T_k is indirectly affected by T_j. It is easy to see that when a (relatively old) transaction B_i that updates x is identified malicious, the damage on x can spread to every object updated by a transaction that is affected by B_i, directly or indirectly. The job of the *Damage Assessor* and the *Damage Repairer* is to locate each affected transaction and recover the database from the damage caused on the objects updated by the transaction. In particular, when an affected transaction is located, the *Damage Repairer* builds a specific *cleaning* transaction to *clean* each object updated by the transaction (and not cleaned yet). Cleaning an object is simply done by restoring the value of the object to its latest undamaged version. This job gets even more difficult as the execution of new transactions continues because the damage can spread to new transactions

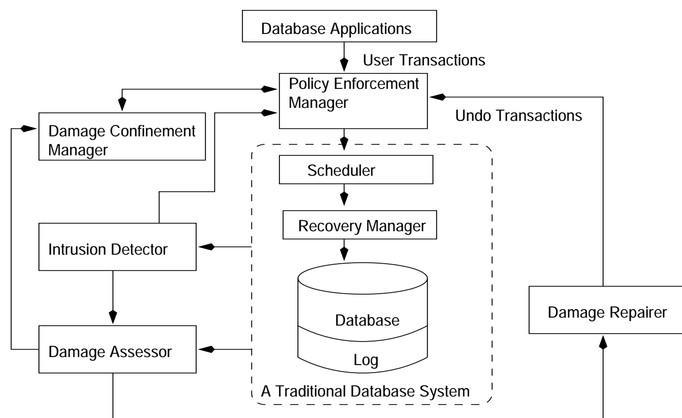

Figure 1. An Intrusion Tolerant Database System without Isolation

and cleaned objects can be re-damaged by new transactions. Therefore, the main objective of this framework is to guarantee that damage spreading is (dynamically) controlled in such a way that the database will not be damaged to a degree that is unacceptable or useless.

The limitation of this architecture is mainly due to the fact that the effectiveness of the framework is heavily dependent on the performance of the *Intrusion Detector* (For example, the framework cannot handle the malicious transactions not captured by the *Intrusion Detector*), and the fact that for accuracy a (relatively) long *detection latency* is usually caused, especially when *anomaly detection* is enforced. One main reason for the latency is that intrusion detection has to make a tradeoff between meeting the requirement of reporting an intrusion accurately (i.e., low false alarm rates) and the requirement of detecting as many intrusions as possible (i.e., high detection rates), which can often result in conflicting design goals. For example, in anomaly detection, for detection accuracy the anomaly threshold for reporting must be high, thus many intrusions with gradual anomaly cannot be identified; on the other hand, in order to capture more intrusions, the threshold should instead be lower, thus the false alarm rate would increase and many legitimate transactions can be mistaken for malicious and suffer denial-of-service. To resolve this dilemma, extending the monitoring time window is one feasible solution. By collecting and investigating more proofs about a suspicious activity, usually more accuracy can be achieved. For example, [20] shows that when using system call trails to identify *sendmail* attacks, synthesizing the anomaly scores of a sequence of system calls (longer than 6) can achieve much better accuracy than based on single system calls. Moreover, in many real world examples, i.e., fraud credit card usage detection, human intervention is needed, which can make the detection latency even longer.

Significant detection latency can cause serious damage spreading in many cases. As a result, the main objective of this framework can be seriously jeopardized. Therefore, how to overcome this limitation is a critical issue.

1.2 Our Approach and Contribution

In [19], *isolation* is proposed as an effective approach to solve the above problem, and a general algorithm to do isolation in the context of database systems is presented. In this approach, when a *suspicious* user S is discovered, although more investigation is still needed for accurate detection, S's access will be isolated transparently into a separate environment that still appears to S to be the actual system. In this way, S's activities are allowed to be kept under (possibly more careful) surveillance without risking further harm to the system. Compared with *fishbowling*, a general technique for isolation, this approach isolates attacks without consuming duplicate resources to construct an entirely separate environment, allows options for partial interaction across the isolation boundary, and provides algorithms for smoothly merging S's work back into the real system should S prove innocent.

This paper presents the design of a real-time data attack isolation system, denoted DAIS. DAIS is a COTS-DBMS-specific implementation of the general isolation algorithm, which is based on an abstract database model. In particular, the design of the first DAIS prototype, which is for Oracle Server 8.1.6, is discussed. DAIS uses triggers and transaction profiles to keep track of the items read and written by transactions, isolates attacks by rewriting user SQL statements, and is transparent to end users. The DAIS design is very general. In addition to Oracle, it can be easily adapted to support many other database application platforms such as Microsoft SQL Server, Sybase, and Informix.

The rest of the paper is organized as follows. In Section 2, we summarize the general isolation algorithm. In Section 3, we present the design of DAIS. We address some performance issues in Section 4. In Section 5, we conclude the paper.

2 The General Isolation Algorithm

2.1 The Model

[19] views a database is a collection of data *items* (or objects). The database *state* is determined by the values of these data items. Data items are operated by *transactions*. A transaction is a partial order of read and write *operations* that either commits or aborts. Two operations *conflict* if one is write. The execution of a set of transactions is modeled by a *history*, which is a partial order $(\sum, <_H)$, where $\sum$ is the set of all operations executed by these transactions, and $<_H$ indicates the execution order of those operations. Two histories are *equivalent* if (1) they are defined over the same set of transactions and have the same operations, and (2) they order conflicting operations of nonaborted transactions in the same way. A history H is *serial* if, for any two transactions T_i and T_j that appear in H, either all operations of T_i appear before those of T_j or vice versa. A history H is *serializable* if its committed projection is equivalent to a serial history [5]. [19] assumes that every history is serializable, and (for simplicity) that the read set of a transaction always contains its write set.

In the model, isolation is enforced in terms of users, when an user is identified suspicious, every transaction executed by the user is simply marked *suspicious*. A straightforward way is using a separate database, which contains a copy of every item in the real database, to isolate each suspicious user. When an user is isolated, every transaction of the user will be executed within the isolated database built for the user. As a result, at time t, if there are n suspicious users, then there are $n + 1$ databases, a real database for all trustworthy users and an isolated database for each suspicious user. However, the naive approach has two limitations: (1) for a large database many items may never be read or updated by a suspicious user, and maintaining these items in isolated databases is a waste of resources; (2) new updates by trustworthy transactions after a suspicious user is isolated are not visible to the isolated user.

To overcome these two limitations, [19] uses data *versions* to give the illusion of isolated databases, and allows isolated users to read new updates of trustworthy users if needed. In particular, a data item x has one trustworthy version and may have multiple suspicious versions. One and only one suspicious version of x is produced for each suspicious user that has updated x. If x has never been updated by a suspicious user, it has no suspicious versions. So the number of suspicious versions is usually much smaller than the number of trustworthy versions. The suspicious versions maintained for a suspicious user S_i, together with the trustworthy versions of the items that do not have a suspicious version for S_i, comprise the *virtual* isolated database for S_i.

Isolation is achieved by controlling the access of transactions to versions. In particular, in the algorithm *one-way isolation* strategy is used, where (a) trustworthy transactions can only read and update trustworthy versions, and (b) suspicious transactions can read but not update trustworthy versions. In particular, transactions of a suspicious user S_i can only update the suspicious versions produced for S_i, and when a transaction of S_i wants to read a data item x, if a suspicious version of x has not been produced for S_i, then the trustworthy version of x is read. Otherwise, the suspicious version is read. Note that in addition to one-way isolation other isolation strategies may also be useful. For example, in some cases, some updates of suspicious users may be able to be disclosed to trustworthy users.

2.2 The Algorithm

The algorithm has three parts: a version control algorithm that controls the access of suspicious users to versions, a conflict identification and resolution algorithm that identifies and resolves the inconsistency between the real database and an isolated database when the corresponding (isolated) user is proved innocent, and a merging algorithm that merges the updates of an isolated user back into the real database. Note that here multiple suspicious users could be simultaneously isolated. Note also that when the updates of an isolated user are to be merged back into the real database, the updates of another isolated user may have already been merged back into the real database.

Part I: Version Control Algorithm

- Before a database system starts to run transactions, each data item x has only one version which is associated with the same version number MAIN, denoted x[MAIN].
- When a trustworthy transaction T wants to read or update x, x[MAIN] is given to T.
- When a transaction submitted by a suspicious user S_i wants to update x,
  - If x has only the MAIN version, then first an additional version of x, which is associated with a unique version number, e.g., the time stamp (denoted t_i) generated when S_i was found suspicious, is created by copying the value of x[MAIN]. The t_i version is then given to S_i to do updates.
  - Otherwise, the t_i version must exist, and it is given.
- When a transaction submitted by a suspicious user S_i wants to read x, if there is a t_i version of x, then the t_i version is given. Otherwise, the MAIN version is given.

When an isolated user is proved malicious, all the suspicious versions maintained for the user are discarded and the real database is not harmed. When an isolated user is proved innocent, we need to merge the updates of the user back into the real database. Since a data item can be independently updated by both a trustworthy transaction and a suspicious transaction, the real database and the isolated database can be inconsistent. The conflict identification and resolution algorithm, which is specified as follows, uses a specific graph, denoted *precedence graph*, to identify and resolve the inconsistencies. [19] shows that if the precedence graph is acyclic, then the real database and the isolated database are consistent.

Part II: Conflict Identification and Resolution Algorithm

- Assume the history of the isolated user is H_s; assume the suffix of the real database history after the user is isolated is H_m
- The precedence graph, denoted $G(H_m, H_s)$, is built as follows
  - Let T_i and T_j be two suspicious transactions or two trustworthy transactions that perform conflicting operations on a data item. There is a directed edge $T_i \rightarrow T_j$ if T_i precedes T_j.
  - If an update of a trustworthy transaction T_g was disclosed to a suspicious transaction T_s during the isolation, then there is a directed edge $T_g \rightarrow T_s$. This type of edge is called a *read edge*. We add read edges to the traditional precedence graph to support one-way isolation.
  - Let T_g be a trustworthy transaction that reads a data item that has been updated by a suspicious transaction T_s, and there is no path from T_g to T_s, then there is a directed edge $T_g \rightarrow T_s$.
  - Let T_s be a suspicious transaction that reads a data item that has been updated by a trustworthy transaction T_g, and there is no path from T_g and T_s that includes a read edge, then there is a directed edge $T_s \rightarrow T_g$.
- If $G(H_m, H_s)$ is acyclic, then the algorithm ends. If $G(H_m, H_s)$ has cycles, then first break all the cycle by backing out some transactions, then end the algorithm. Although it is shown in [6] that just finding the optimal back out strategy is NP-complete, the simulation results of [6] show that in many cases, several back out strategies, in particular *breaking two-cycles optimally*, can achieve good performance.
- For each transaction T_g that is backed out from H_m, locate every active suspicious history which has a read edge from T_g, and for each such read edge, denoted $T_g \rightarrow T_s$, back out T_s and every transaction that is affected by T_s.

After the inconsistency between the real database and the isolated database is resolved, we can use the following algorithm to merge the (left) updates of the isolated user back into the database. [19] shows that given H_g and H_s, the precedence graph $G(H_m, H_s)$ is acyclic if and only if there is an equivalent merged serializable history H that can generate the same database state as generated by the merging algorithm.

Part III: Merging Algorithm

- For every data item that has a suspicious version maintained for the isolated user, replace its trustworthy version (value) with its suspicious version (value), and then remove the suspicious version.

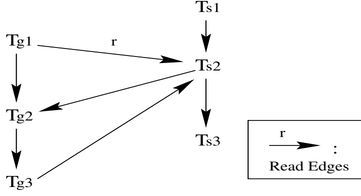

Figure 2. Precedence Graph for Example 1

2.3 An Example

We illustrate the algorithm by an example.

Example 1 *Consider the five transactions given below:*

$READSET(T_{g1}) = WRITESET(T_{g1}) = \{d_1, d_3\}$
$READSET(T_{g2}) = WRITESET(T_{g2}) = \{d_1, d_4\}$
$READSET(T_{g3}) = \{d_2, d_4, d_5\}$, $WRITESET(T_{g3}) = \{d_5\}$
$READSET(T_{s1}) = WRITESET(T_{s1}) = \{d_7\}$
$READSET(T_{s2}) = \{d_1, d_2, d_3, d_7\}$, $WRITESET(T_{s2}) = \{d_2, d_3\}$
$READSET(T_{s3}) = \{d_3, d_6\}$, $WRITESET(T_{s3}) = \{d_6\}$

Assume that the real database history is $H_g = T_{g1}\ T_{g2}\ T_{g3}$ and the suspicious history is $H_s = T_{s1}\ T_{s2}\ T_{s3}$. Assume the commit time order is $T_{s1}\ T_{g1}\ T_{s2}\ T_{g2}\ T_{s3}\ T_{g3}$. It is clear the first version for the isolated user is created when T_{s1} updates d_7. T_{s2} will read the updates of T_{g1} on d_1 and d_3, and read the update of T_{s1} on d_7. After T_{s2} is executed there are three suspicious versions for the isolated user, namely, d_2, d_3, and d_7. However, T_{g3} cannot read the update of T_{s2} on d_2 according to the one-way isolation strategy. After T_{s3} is executed there is one more suspicious version (for d_6). Since every item has a trustworthy version, any corruption on these suspicious versions will not harm the real database.

If the isolated user is proved innocent, the precedence graph $G(H_g, H_s)$ shown in Figure 2 can be used to identify and resolve the conflicts between H_g and H_s. Since the graph has a cycle, conflicts exist among the transactions. For example, since T_{s2} reads item d_1, which is updated by T_{g2}, T_{s2} should precede T_{g2}; since T_{g2} should precede T_{g3}, T_{s2} should precede T_{g3}; however, since T_{g3} reads item d_2, which is updated by T_{s2}, T_{g3} should precede T_{s2}, yielding a contradiction.

The inconsistency between the real database and the isolated database can be resolved by breaking this cycle. For example, if T_{g3} is backed out, then the merged database can be generated by equivalent history $H = T_{s1}\ T_{g1}\ T_{s2}\ T_{g2}\ T_{s3}$. If T_{s2} is backed out, then T_{s3} has to be backed out because it is affected by T_{s2}, and the merged database can be generated by equivalent history $H = T_{s1}\ T_{g1}\ T_{g2}\ T_{g3}$.

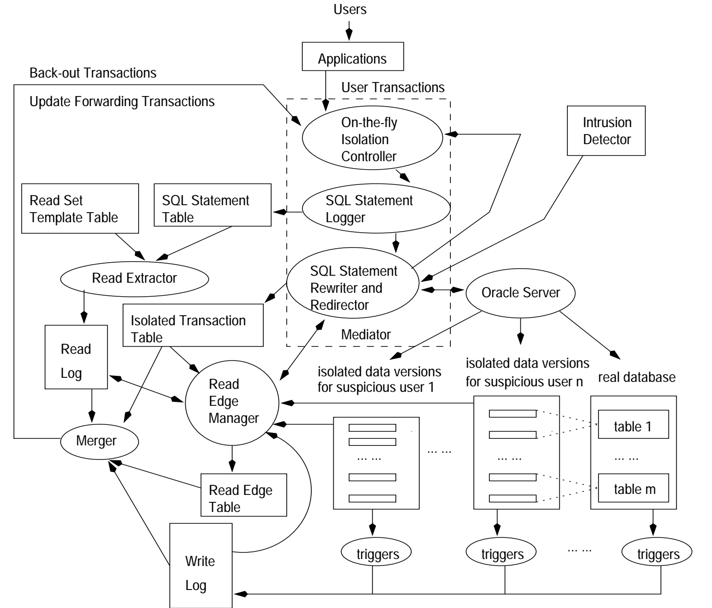

Figure 3. System Model of DAIS

3 DAIS

The major components of DAIS are shown in Figure 3. In general, the *Intrusion Detector* informs DAIS which users are suspicious and should be isolated. The *Mediator*, which has three components, *proxies* every user transaction and SQL statement (or command). The *triggers*, the *SQL Statement Logger*, and the *Read Extractor* are responsible for keeping track of the read and write operations of transactions, which are necessary to build the precedence graph when a merging should be done. The *SQL Statement Rewriter and Redirector* (SRR) is responsible for enforcing the Version Control Algorithm. The *Read Edge Manager* helps to maintain read edges. The *Merger* is responsible for enforcing Conflict Identification and Resolution Algorithm and the Merging Algorithm. The *On-the-fly Isolation Controller* enables new user transactions to continue executing without jeopardizing the correctness of merging processes.

3.1 Intrusion Detector

The Intrusion Detector informs DAIS which users should be isolated. The Intrusion Detector does anomaly detection in terms of each transaction and each user. For each transaction of a user, the Intrusion Detector will generate an *anomaly degree* (based on the transaction's behavior), which indicates how abnormal (or suspicious) the transaction is. Then the anomaly degrees of the transactions submitted by the user (within a session) will be synthesized to generate the anomaly degree of the user. If the anomaly de-

gree is above a specific threshold, the user will be reported to the SRR as *suspicious*. The threshold is adaptively determined by the SSO and the Intrusion Detector based on the probability that a suspicious transaction turns out to be malicious, which can be indicated, to some extent, by the detection history. Readers can refer to [29] for more details about this detector.

3.2 Triggers

Oracle Redo logs record every *write* operation, however, unfortunately its structure is confidential, so getting information about writes from Oracle Redo logs is very difficult. This is also why we use triggers to keep track of writes.

Triggers can capture the write operations associated with a Update, Delete, or Insert statement. We use triggers to keep track of the writes of both trustworthy and suspicious transactions. To illustrate, consider a simple banking database that has two tables: the the Account table that keeps the current balance of each customer account, and the Money_Transaction table that keeps the trails of every deposit or withdraw. The Account table has two fields: Account_ID that is the primary key, and Balance. The Money_Transaction table has four fields: (1) Money_Transaction_ID, the primary key; (2) Account_ID; (3) Amount; (4) Date; and (5) Teller_ID. When a user deposits $500 to his account, we assume the database transaction inserts a new money transaction record into the Money_Transaction table and updates his account balance. After this transaction, the new records added by the triggers associated with these two tables to the Write Log are shown in the following table:

Transaction_ID	Table_Name	Key	OP_Type
4.91.6240	Money_Transaction	2833	Insert
4.91.6240	Account	1591766	Update

Note that Oracle uses a unique seven digit Transaction_ID to identify each (database) transaction. Note also that here for simplicity we log writes at *record* level where each data item represents a record. Logging writes at *field* level can achieve finer isolation control, however, it causes more resources.

3.3 SQL Statement Logger

Although triggers can capture every write, triggers are generally not able to capture the reads by SQL statements. Although every record that is updated or deleted is also a record that is read, a Delete, Insert, or Update statement can cause more reads especially when they contain nested Select statements, which we will show shortly.

In order to capture reads, we take an approach to extract reads from SQL statement profiles and transaction input arguments. The job of the SQL Statement Logger is to keep every SQL statement as well as its arguments in the SQL Statement Table. To illustrate, Table 1 shows the corresponding SQL statements kept by this component for the example transaction introduced in Section 3.2. Here the Seq_No fields indicate the execution order of SQL statements within a transaction.

3.4 Read Extractor

Triggers can capture every write operation, however, no COTS DBMS logs read operations and triggers cannot capture reads. DAIS takes the approach of extracting reads from SQL statement texts. In particular, the Read Extractor uses the SQL Statement Table and the Read Set Template Table to extract read operations of transactions. We assume each user transaction belongs to a specific *transaction type*, and the transaction *profile* (or program) for each transaction type is known. DAIS cannot automatically extract reads from ad-hoc transactions without a known profile. The Read Set Template Table, which keeps the read set *template* for each transaction type, is extracted from the profiles of these transaction types. The reads of a specific transaction T is restored by *materializing* the read set template of T's transaction type with the input arguments of T, which are kept in the SQL Statement Table. We use a simple example to show the idea. Consider again the example transaction introduced in Section 3.2, assume it belongs to the Deposit transaction type, then the read set template of Deposit transactions is shown in the following table:

Trans_Type	Table_Name	Identifier
Deposit	Account	AccID

Here the Identifier field sometimes may contain only a part of a primary key (In this case, an identifier could identify multiple records). Note that the Insert statement causes no reads.

The read set template tells us which table is read, but does not tell us which records are read. In order to figure out which records are read, we need to materialize the Identifier fields. For this purpose, we use the following input argument extraction table:

Trans_Type	Identifier Var	S_Pattern	After	Before
Deposit	AccID	'UPDATE'	'Account_ID ='	';'

This table says that for a Deposit transaction the value of the Identifier variable AccID is the number after the string "Account_ID =" and before the char ";" within the SQL statement that starts with "UPDATE". Using this table and Table 1, we can easily figure out that for transaction 4.91.6240 the Identifier variable AccID is 1591766 (Note that since a transaction could execute different sets of statements based on different input arguments and database states, Seq_No sometimes can identify a wrong statement). Then based on the previous Template table, we can know the transaction has read only one record, which can be denoted as Account.1591766. Then this read operation will be inserted into the Read Log, as shown in the following table:

Transaction_ID	Table_Name	Primary Key
4.91.6240	Account	1591766

In the above example, we assume the type of transaction 4.91.6240 is known. However, the Read Extractor in fact

Trans_ID	SQL_Type	Statement	Seq_No
4.91.6240	INSERT	INSERT INTO Money_Transaction (Money_Transaction_ID, Account_ID, Amount, Date, Teller_ID) VALUES (2833, 1591766, 500, '21-Mar-00', 'Teller-1');	1
4.91.6240	UPDATE	UPDATE Account SET Balance = Balance+500 WHERE Account_ID = 1591766;	2

Table 1. Example SQL Statements Kept in the SQL Statement Table

does not know the type of transaction 4.91.6240 when Table 1 is scanned. DAIS uses an additional table (called the Transaction Pattern Table), which stores the profile pattern of each transaction type, to identify the type of a transaction. For example, the profile pattern for Deposit transactions can say that each transaction has two statements and the first one starts with "INSERT INTO Money_Transaction". By matching the real statements executed by a transaction with the Transaction Pattern Table, we can figure out the type of a transaction.

Our study has shown that for most real world Select, Insert, and Delete statements that have a simple structure, the above method can work very well. However, it should be noticed that for many complex SQL statements that contain nested Select statements, extracting reads is not a easy job. In [2], some general guidelines are given for extracting reads from nested statements. Moreover, it should be noticed that the method of extracting reads from transaction profiles sometimes may only restore an approximation of the real read set of a transaction because the complexities of transactions can result in such read set templates that make exact materialization impossible. In order to guarantee that approximate read sets will not affect the correctness of DAIS, we need to enforce the following constraint during the process of extracting read set templates:

Containment Rule: Whenever a read set template is extracted from a transaction profile, any transaction T of that type, when executed, must have a real read set contained by the materialized read set template.

3.5 SQL Statement Rewriter and Redirector

To save resources, DAIS does not use completely replicated databases to do isolation, instead, DAIS maintains extra *data versions* only for the items that are updated (including item creation) by an isolated transaction. In particular, when a suspicious user is isolated, for each (real database) table R_i that the user wants to write, DAIS maintains one extra table with the same structure, called a *suspicious version* of R_i (denoted S_i), only during the isolation period. S_i keeps all and only the writes of the user on R_i. Hence, the size of S_i should be much smaller than R_i in most cases.

In addition to maintaining suspicious data versions, the SQL Statement Rewriter and Redirector (SRR) maintains the information about each suspicious transaction in the Isolated Transaction Table whose structure is shown as follows.

User_ID	Trans_ID	Status

Here the Status field of a record indicates the corresponding transaction *commits*, *aborts*, or is still *active*.

The key challenge of DAIS is how to enforce one-way isolation. When an isolated user issues a SQL statement to access a table R_i, according to one-way isolation, a record r in R_i can be read only if r does not have a version in S_i, and only S_i can be modified. To achieve this, changing the way SQL statements are executed, i.e., using templates, can cause substantial overhead, however, executing the SQL statement solely on S_i can generate invalid results, so the only practical way is to rewrite the SQL statement.

To rewrite and redirect user SQL statements, we have to proxy user transactions. DAIS assumes every Application uses *OCI calls*, a standard interface for Oracle, to access the database. To proxy user transactions, DAIS proxies every OCI call by providing the Applications with a *pseudo* OCI interface, which forwards each OCI call to the Mediator instead of the Oracle Server. Note that although currently DAIS only supports the applications that use OCI calls, the Mediator can be extended to support a variety of other Oracle interfaces such as ODBC, JDBC, Pro*C, and SQL*NET.

In Oracle database applications, a user can read a table through four kinds of SQL statements, namely, Select, Insert, Update, or Delete (Note that when a Delete or Insert statement has one or more Select statements nested, it can cause reads), and a user can write a table through the four kinds of SQL statements except Selects. Hence, in order to enforce the Version Control Algorithm, DAIS may need to rewrite and redirect every kind of SQL statements. The SQL statement rewriting and redirecting algorithm is specified as follows. In general, the rewriting algorithm for each kind of SQL statements has three steps: (1) build the virtual database for the isolated user; (2) execute the statement; (3) restore the real database.

Algorithm 1 SQL Statement Rewriting and Redirecting Algorithm
while TRUE
  if a new SQL statement wants to be executed
    if the SQL statement is for a trustworthy user
      ▷ forward the SQL statement to the Oracle server

without any changes;

else if the SQL statement is a Select and assume it wants to query tables R_{j1}, R_{j2}, ..., R_{jn}

▷ for each R_{jk} that has some records that are deleted from S_{jk}, delete these records from R_{jk} and keeps these records in memory. We can query the Write Log to know which records have been deleted from S_{jk};

▷ for each S_{jk} that has some records that are not in R_{jk}, insert these records into R_{jk};

▷ rewrite the SQL statement (denoted $orig_stat$) as the following sequence of statements: (sel_{j1}, upd_{j1}, sel_{j2}, upd_{j2}, ..., sel_{jn}, upd_{jn}, $orig_stat$, res_{j1}, res_{j2}, ..., res_{jn}). Within the sequence, sel_{jk} is "SELECT * INTO :k FROM R_{jk} WHERE R_{jk}.primary_key IN (SELECT S_{jk}.primary_key FROM S_{jk}) FOR UPDATE;". Here :k is a bind array to tentatively keep all the rows that are selected. The FOR UPDATE clause ensures that no other users can lock or update these rows until this transaction ends. upd_{jk} is "UPDATE R_{jk} a SET (all fields) = (SELECT * FROM S_{jk} b WHERE a.primary_key = b.primary_key) WHERE R_{jk}.primary_key IN (SELECT S_{jk}.primary_key FROM S_{jk});". res_{jk} is "UPDATE R_{jk} SET (all fields) = (the corresponding record of :k) WHERE R_{jk}.primary_key IN :k;";

/* **See Comment A***/

▷ forward these statements to the Oracle server for execution;

▷ delete every record that has been inserted into R_{j1}, ..., R_{jn};

▷ for the records that have been deleted from R_{j1}, ..., R_{jn}, re-insert them into these tables;

else if the SQL statement is a Delete and assume it does not handle any nested tables and its WHERE clause accesses tables R_{j1}, ..., R_{jn}

▷ for each R_{jk} that has some records that are deleted from S_{jk}, delete these records from R_{jk} and keeps these records in memory;

▷ for each S_{jk} that has some records that are not in R_{jk}, insert these records into R_{jk};

▷ rewrite the Delete statement in such a way that the key word "DELETE" is replaced with "SELECT *";

▷ rewrite the Select statement (generated from the previous step) as (sel_{j1}, upd_{j1}, sel_{j2}, upd_{j2}, ..., sel_{jn}, upd_{jn}, $orig_stat$, res_{j1}, res_{j2}, ..., res_{jn}). Here $orig_stat$ is the Select statement. Every other statement is composed in the same way as we (previously) did for Select statements;

▷ **if** the statement is the first statement of the transaction it belongs to **and** the transaction is not the first transaction of the user that submits it

▷ wait until a *FINISH* message for the user arrives;

/* **See Comment B***/

▷ forward the sequence to the Oracle server for execution, assume the result is denoted To_Delete;

▷ delete every record that has been inserted into R_{j1}, ..., R_{jn};

▷ for the records that have been deleted from R_{j1}, ..., R_{jn}, re-insert them into these tables;

▷ for each record in To_Delete, if it is in the corresponding S_{jk}, then delete it from S_{jk}. If it is not in the corresponding S_{jk}, insert a Delete operation to the Write Log which says this record is deleted from S_{jk}, although this operation is not really done;

else if the SQL statement is an Insert

if the table (denoted R_i) the SQL statement wants to insert data into has not a suspicious version (denoted S_i) for the suspicious user

▷ create S_i;

if the SQL statement has no subqueries in the values_clause

▷ rewrite the Insert statement in such a way that R_i is replaced by S_i;

▷ **if** the statement is the first statement of the transaction it belongs to **and** the transaction is not the first transaction of the user that submits it

▷ wait until a *FINISH* message for the user arrives;

▷ forward the rewritten Insert statement to the Oracle Server;

else assume the subquery of the SQL statement accesses tables R_{j1}, ..., R_{jn}

▷ for each R_{jk} that has some records that are deleted from S_{jk}, delete these records from R_{jk} and keeps these records in memory;

▷ for each S_{jk} that has some records that are not in R_{jk}, insert these records into R_{jk};

▷ rewrite the Insert statement in such a way that the string "INSERT INTO table_name" is removed;

▷ rewrite the Select statement (generated from the previous step) as (sel_{j1}, upd_{j1}, sel_{j2}, upd_{j2}, ..., sel_{jn}, upd_{jn}, $orig_stat$, res_{j1}, res_{j2}, ..., res_{jn}). Here $orig_stat$ is the Select statement. Every other statement is composed in the same way as we (previously) did for Select statements;

▷ **if** the statement is the first statement of the transaction it belongs to **and** the transaction is not the first transaction of the user that submits it

▷ wait until a *FINISH* message for the user arrives;

▷ forward the sequence to the Oracle server for execution, assume the result is denoted To_Insert;

▷ delete every record that has been inserted into R_{j1}, ..., R_{jn};

▷ for the records that have been deleted from R_{j1}, ..., R_{jn}, re-insert them into these tables;

▷ for each record in To_Insert, insert it into the corresponding S_{jk};

else if the SQL statement is an Update

▷ assume the nested queries of the Update statement access tables R_{j1}, ..., R_{jn}

▷ for each R_{jk} that has some records that are deleted from S_{jk}, delete these records from R_{jk} and keeps these records in memory;

▷ for each S_{jk} that has some records that are not in R_{jk}, insert these records into R_{jk};

▷ rewrite the Update statement as (sel_{j1}, upd_{j1}, sel_{j2}, upd_{j2}, ..., sel_{jn}, upd_{jn}, $orig_stat$, res_{j1}, res_{j2}, ..., res_{jn}). Here

$orig_stat$ is the Update statement with the string "UPDATE R_i" rewritten as the string "UPDATE S_i".
Every other statement is composed in the same way as we (previously) did for Select statements;
▷ **if** the Update statement is the first statement of the transaction it belongs to **and** the transaction is not the first transaction of the user that submits it
▷ wait until a *FINISH* message for the user arrives;
▷ forward the sequence to the Oracle server for execution;
▷ delete every record that has been inserted into R_{j1}, ..., R_{jn};
▷ for the records that have been deleted from R_{j1}, ..., R_{jn}, re-insert them into these tables;
end while

Comments

A. A record kept in a trustworthy table R_{jk} can be independently updated by the isolated user within S_{jk}. For such records the isolated user should read the versions kept in S_{jk} instead of R_{jk}. For this purpose, for each such table R_{jk}, the sel_{jk} statement keeps the trustworthy versions of such records, the upd_{jk} statement replaces these trustworthy versions with the corresponding suspicious versions kept in S_{jk}, and the res_{jk} statement restores these trustworthy versions after the SELECT statement is done.

B. In order to ensure the correctness of the Read Edge Maintenance Algorithm presented in the next section, namely, Algorithm 2, we need some synchronization between the SRR and the Read Edge Manager (REM). In particular, the REM gathers read edges caused by a suspicious transaction T_i by checking whether or not a data item is read by the transaction from the real database instead of a suspicious table. To enable the checking, the REM has to query the suspicious tables accessed by the transaction. However, if before the checking another transaction T_j of the user that submits T_i is executed, then the writes of T_j could make the checking report wrong information.

3.6 Read Edge Manager

The Read Edge Manager (REM) uses the Read Edge Table, whose structure is shown as follows, to keep read edges.

User_ID	From	To

In particular, each record of the table represents a read edge, the From and To fields represent the identifiers of the two transactions that comprise the read edge. The algorithm of the REM is as follows. Here we assume that whenever a suspicious transaction commits, the SRR will send an *ISO-TRANS-COMMIT* message to the REM.

Algorithm 2 Read Edge Maintenance Algorithm
while TRUE
if a new *ISO-TRANS-COMMIT* message arrives
▷ query the Read Log to get the read set of the transaction (indicated by the message);
▷ for each record in the read set:
if the record is in the corresponding suspicious table S_{jk}
▷ update the Read Log table such that the Table_Name field of the record is replaced by the name of S_{jk};
else if the record is deleted by the transaction from S_{jk}
▷ update the Read Log table such that the Table_Name field of the record is replaced by the name of S_{jk};
else
▷ search the Write Log table to find the latest trustworthy transaction that updates the record;
if the result is empty
▷ do nothing;
else
▷ insert into the Read Edge Table the read edge from the isolated transaction to the trustworthy transaction if the same read edge has not been inserted;
▷ send a *FINISH* message, which contains the identifier of the transaction, to the SRR;
end while

3.7 Merger

The information kept in the Read Log, the Write Log, the Read Edge Table, and the Isolated Transaction Table is enough for the Merger to perform the Conflict Identification and Resolution Algorithm in almost the same way as we specified in Section 2. In particular, when a suspicious user turns out to be innocent, from the Isolated Transaction Table we know the history of the user, from the Isolated Transaction Table and the Write Log we know the history of trustworthy transactions, from the Read Log and the Write Log we know the read and write sets of each transaction, from the Read Edge Table we know all the relevant read edges. In addition, to back out a transaction T, the Merger will compose a specific Back-Out transaction, which restores every data item x updated by T to the latest value before x was updated, and submit the Back-Out transaction to the Mediator. To forward updates, at the end of a merging, the Merger will compose a specific Update-Forward transaction, which forwards all the remained updates (of the once isolated user) to the real database, and submit the Update-Forward transaction to the Mediator.

One key challenge of the Merger is how to enable multiple users to be simultaneously isolated, and how to handle the impact of one merging on other isolated histories. For this purpose, the Merger does the following things:

- For simplicity, the Merger never back out trustworthy transactions when doing a merging. Although in this way more back-out cost could be caused, after each

merging we need not to check the impact of a back-out trustworthy transaction on each (still) isolated history and back out each isolated transaction that is affected by the trustworthy transaction, which could cost more.

- After a merging, if there are still some users being isolated, the merged history will be maintained so that the next (possible) merging can be done based on the merged history. The merged history can be maintained in the form of a *dependency graph* [2].
- After a merging, if there is no user being isolated, the maintained merged history will be discarded, and when a new user is later on isolated, the part of the trustworthy history that will participate in each further merging will be respecified, which will only cover the trustworthy transactions that commit after the user is isolated. In this way, the processing load on the Merger for the next merging can be dramatically decreased.

3.8 On-the-fly Isolation Controller

Doing a merging can take a while. Disallowing every user transaction to run during a merging could cause serious denial-of-service. However, allowing user transactions to continue being executed without any restriction as the merging is processed can jeopardize the correctness of the merging. To ensure that the correctness of merging will not be jeopardized, the On-the-fly Isolation Controller (OIC) works as follows. Note that the drawback (of the OIC algorithm) is that (1) during a merging, (almost) no new transaction of the once isolated user can be executed, and (2) the locked part of the real database can block the execution of some new user transactions, although many other user transactions that do not access the locked tables can still be executed.

- When the Intrusion Detector tells the SRR that an isolated user turns out to be innocent, the SRR sends an *INNOCENT* message to the OIC.
- When the OIC receives an *INNOCENT* message, the OIC will *lock* every suspicious table S_{jk} that is created for the user and the corresponding R_{jk} table.

4 Performance Issues

The merit of DAIS is that it causes no extra delay for trustworthy transactions. Although the Mediator does cause some delay for every transaction, this delay is unavoidable even if we do not do isolation because we still need to mediate (or proxy) transactions for the purpose of damage assessment and repair. The limitation of DAIS is that rewriting SQL statements could make the isolated user suffer some delay, although in most cases this delay can be significantly reduced by concurrent SQL rewriting and redirecting. For example, for a deposit transaction, rewriting of the UPDATE statement could be finished during the period of time when the INSERT statement is executed, thus the delay for the UPDATE statement is reduced to zero (if the UPDATE statement should be executed after the INSERT statement).

DAIS can be further optimized to improve performance in the following ways:

- Although extracting reads from transaction profile is efficient, this method can only get an approximate read set in some cases. This could cause many innocent transactions to be mistakenly backed out. To reduce the number of mistakenly backed-out transactions, we can let the Mediator to inform the Read Extractor some results of some SQL statements. Since these results indicate exactly which items are read, they can help the Read Extractor to get more accurate read sets.
- Although Algorithm 1 can handle any kind of SQL statements, for the SQL statements that only access the records kept in the real database, the Algorithm can cause too much extra cost. To reduce the extra cost, we could optimize Algorithm 1 as follows: for a suspicious SQL statement, for every table R_{jk} that the statement wants to access, we first check if the intersection of its (intended) reads on R_{jk} and S_{jk} is *empty* (Note that here the reads can be extracted from SQL statement texts). If the intersection is empty, we can execute the SQL statement directly on R_{jk} without losing the correctness.

5 Conclusion

In this paper, we present the design of DAIS, a practical real-time data attack isolation system for intrusion tolerant commercial database applications. By transparently isolating suspicious transactions, DAIS allows suspicious activities to be kept under surveillance without risking further harm to the system or causing denial-of-service to benign transactions. Built on top of Oracle Server 8.1.6, DAIS uses triggers and transaction profiles to keep track of the data items read and written by transactions, and isolates attacks by rewriting and redirecting user SQL statements. Compared with fishbowling, the extra resources consumed by DAIS are much less. To our best knowledge, DAIS is the first tool which can do real-time cost-effective data attack isolation for commercial database applications. DAIS design is very general. In addition to Oracle, it can be easily adapted to support many other database application platforms such as Microsoft SQL Server, Sybase, and Informix. By designing DAIS the conceptual soundness, the functional capabilities, the feasibility, and the practicality of the general isolation approach can be better validated.

There are some future works for DAIS. First, currently after the DAIS tool is started, there is no other input to DAIS except the set of suspicious users (from the Intrusion

Detector) and the trails of transactions. Another module can be added to DAIS to provide more interactions between the SSO and DAIS. One desirable feature of this interface would be the capability of showing the SSO the users that are being isolated and the execution of their transactions. Second, the security of DAIS is not addressed in the prototype design. However, successful attacks on DAIS can cause some benign users to be isolated, can cause some malicious users to stay at large, and can cause incorrect merging, which will jeopardize the consistency of the database. For one example, if the messages from the Intrusion Detector to the SRR can be successfully faked, DAIS can isolated benign users. For another example, if the read set templates are maliciously modified then some cycles in a precedence graph may not be able to be identified, and some transactions that cause inconsistency may be mistakenly merged. Therefore, the security of DAIS is a significant concern.

References

[1] M. R. Adam. Security-Control Methods for Statistical Database: A Comparative Study. *ACM Computing Surveys*, 21(4), 1989.

[2] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 2001. To appear.

[3] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[4] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Proceedings of the 2000 International Conference on Extending Data Base Technology*, Mar 2000.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[6] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–581, September 1984.

[7] D.E.Denning. An intrusion-detection model. *IEEE Trans. on Software Engineering*, SE-13:222–232, February 1987.

[8] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

[9] P. P. Griffiths and B. W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.

[10] P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, 1993.

[11] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1993.

[12] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[13] R. Jagannathan and T. Lunt. System design document: Next generation intrusion detection expert system (nides). Technical report, SRI International, Menlo Park, California, 1993.

[14] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 474–485, May 1997.

[15] H. S. Javitz and A. Valdes. The sri ides statistical anomaly detector. In *Proceedings IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 1991.

[16] H. S. Javitz and A. Valdes. The nides statistical component description and justification. Technical Report A010, SRI International, March 1994.

[17] T. Lane and C.E. Brodley. Temporal sequence learning and data reduction for anomaly detection. In *Proc. 5th ACM Conference on Computer and Communications Security*, San Francisco, CA, Nov 1998.

[18] Wenke Lee, Sal Stolfo, and Kui Mok. A data mining framework for building intrusion detection models. In *Proc. 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[19] P. Liu, S. Jajodia, and C.D. McCollum. Intrusion confinement by isolation in information systems. *Journal of Computer Security*, 8(4):243–279, 2000.

[20] P. Luenam and P. Liu. Odam: An on-the-fly damage assessment and repair system for commercial database applications. In *Proc. 15th IFIP WFG11.3 Working Conference on Database and Application Security*, Ontario, Canada, July 2001.

[21] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H. S. Javitz, A. Valdes, P. G. Neumann, and T. D. Garvey. A real time intrusion detection expert system (ides). Technical report, SRI International, Menlo Park, California, 1992.

[22] Teresa Lunt and Catherine McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corporation, McLean, VA, 1998.

[23] T.F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[24] J. McDermott and D. Goldschlag. Storage jamming. In D.L. Spooner, S.A. Demurjian, and J.E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 365–381. Chapman & Hall, London, 1996.

[25] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.

[26] B. Mukherjee, L. T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.

[27] P.A. Porras and R.A. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the 8th Annual Computer Security Applications Conference*, San Antonio, Texas, December 1992.

[28] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, 1994.

[29] P. Liu S. Ingsriswang. Aaid: An application aware transaction-level database intrusion detection system. Technical report, Department of Information Systems, UMBC, Baltimore, MD, 2001.

[30] D. Samfat and R. Molva. Idamn: An intrusion detection architecture for mibile networks. *IEEE Journal of Selected Areas in Communications*, 15(7):1373–1380, 1997.

[31] R. Sandhu and F. Chen. The multilevel relational (mlr) data model. *ACM Transactions on Information and Systems Security*, 1(1), 1998.

[32] S.-P. Shieh and V.D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):661–667, 1997.

[33] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, 1994.