ODAM: An On-the-fly Damage Assessment and Repair System for Commercial Database Applications*

Pramote Luenam, Peng Liu

Dept. of Info. Systems, UMBC
Baltimore, MD 21250
pluena1@umbc.edu, pliu@umbc.edu

Abstract

This paper presents the design and implementation of an on-the-fly damage assessment and repair tool for intrusion tolerant commercial database applications, called ODAM. *ODAM is a COTS-DBMS-specific implementation of a general on-the-fly damage assessment and repair approach developed by P. Ammann, S. Jajodia, and P. Liu in [18]. The general approach, given a set of malicious transactions reported by an intrusion detector, locates and repairs the damage caused by each malicious transaction on the database, along with the damage caused by any benign transaction that is affected, directly or indirectly, by a malicious transaction. The general approach locates and repairs damage on-the-fly without the need to periodically halt normal transaction processing. In this paper, the development of the first ODAM prototype, which is for Oracle Server 8.1.6, is discussed. ODAM uses triggers and transaction profiles to keep track of the read and write operations of transactions, locates damage by tracing the affecting relationships among transactions along the history, and repairs damage by composing and executing some specific* UNDO *transactions. ODAM is transparent to on-going user transactions and 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. To our best knowledge, ODAM is the first tool that can do automatic on-the-fly damage assessment and repair for commercial database applications..*

1. Introduction

Database security concerns the confidentiality, integrity, and availability of data stored in a database. A broad span of research from authorization [5,7,14], to inference control [1], to multilevel secure databases [15,17], and to multilevel secure transaction processing [2], 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, air traffic control, 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 resilient database systems that can survive malicious attacks are a significant concern.

One critical step towards attack resilient database systems is intrusion detection, which has attracted many researchers [4,9,13]. 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* [8] or on known patterns of attacks, called *signatures* [6,16].

* Luenam and Liu were supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-00-2-0575.

Intrusion detection can supplement protection of database systems by rejecting the future access of detected attackers and by providing useful hints on how to strengthen the defense. However, intrusion detection makes the system attack-aware but not attack-resilient, that is, intrusion detection itself cannot maintain the integrity and availability of the database in face of attacks.

To overcome the inherent limitation 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 a critical database intrusion tolerance problem beyond intrusion detection, namely *attack recovery*, and presents the design and implementation of an on-the-fly attack recovery tool, called *ODAM*.

The attack recovery problem can be better explained in the context of an intrusion tolerant database system. Database intrusion tolerance can 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 [3,9,10,11,12] 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 discussion will be based on the (conceptual) intrusion tolerant database system architecture shown in Figure 1.

The architecture is built on top of a traditional COTS (Commercial-Off-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 **UNDO** transactions. The *Damage Confinement Manager* restricts the access to the data items that have been identified by the Damage Assessor as damaged, and unconfines a data item after it is cleaned. The *Policy Enforcement Manager* (PEM) (a) functions as a proxy for normal user transactions and those UNDO 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 a user as soon as the Intrusion Detector finds that the user is malicious.

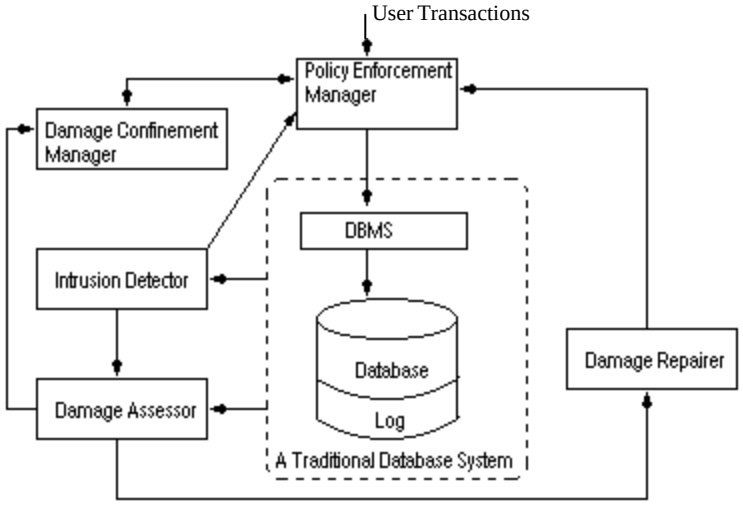

Figure 1: Basic ItDBMS System Architecture

The attack recovery problem has two aspects: *damage assessment* and *damage repair*. The complexity of attack recovery 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 a data item x updated by another transaction T_j (We say T_i *reads* x *from* 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 data item updated by a transaction that is affected by B_i, directly or indirectly.

The goal of attack recovery is to locate each affected transaction and recover the database from the damage caused on the data items updated by every malicious or affected transaction. And the goal of ItDBMS is to identify each malicious transaction, and to assess and repair the damage caused by these malicious transactions in a timely manner such that the database will not be damaged to such a degree that is unacceptable or useless.

In some cases, the attacker's goal may be to reduce availability by attacking integrity. In these cases the attacker's goal not only introduces damage to certain data items and uncertainty about which good transactions can be trusted, but also achieves the goal of bringing the system down while repair efforts are being made. To address the availability threat, on-the-fly attack recovery without the need to halt normal transaction processing is needed. It is clear that the job of attack recovery gets more difficult as use of the database continues because the damage can spread to new transactions and cleaned data items can be re-damaged by new transactions.

ODAM is a COTS-DBMS-specific implementation of a general on-the-fly damage assessment and repair approach developed by P. Ammann, S. Jajodia, and P. Liu in [18]. The general approach, given a set of malicious transactions reported by an intrusion detector, locates and repairs the damage caused by each malicious transaction on the database, along with the damage caused by any benign transaction that is affected, directly or indirectly, by a malicious transactions. In this paper, the development of the first ODAM prototype, which is for Oracle Server 8.1.6, is discussed. ODAM uses triggers and transaction profiles to keep track of the read and write operations of transactions, locates damage by tracing the affecting relationships among transactions along the history, and repairs damage by composing and executing some specific UNDO transactions. ODAM is transparent to on-going user transactions and 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. To our best knowledge, ODAM is the first tool that can do automatic on-the-fly damage assessment and repair for commercial database applications. We have tested the ODAM prototype using simulated data. The results show that ODAM causes very little performance penalty, is practical, and can be used to effectively help provide “data integrity” guarantees to arbitrary commercial database applications in face of attacks.

The rest of the paper is organized as follows. Section 2 gives an introduction to the on-the-fly attack recovery approach. In Section 3, we present the design and implementation of ODAM. Section 4 shows the results of various tests performed on different aspects of ODAM. In Section 5, we conclude the paper.

2. Introduction to the On-the-fly Attack Recovery Approach

This section provides a simple introduction to the general on-the-fly damage assessment and repair approach developed by P. Ammann, S. Jajodia, and P. Liu in [18].

2.1 Affecting Relationships among Transactions

In this approach, a *database* is a set of data items handled by *transactions*. A transaction is a partial order of *read* and *write* operations that either *commits* or *aborts*. The execution of a set of transactions is modeled by a structure called a *history*. In a history, a transaction T_i is dependency upon another transaction T_j if there exists a data item x such that T_i reads x after T_j updates it, and there is no transaction that updates x between the time T_j updates x and T_i reads x. The the size of the intersection of T_j ’s write set and T_i ’s read set is called the *dependency degree* of T_i upon T_j . The *dependent upon* relation indicates the path along which damage spreads. In particular, if a transaction which updates an item x is dependent upon a malicious transaction which updates an item y, we say the damage on y *spreads* to x, and we say x is *damaged*. Moreover, a transaction T_u *affects* transaction T_v if the ordered pair (T_v ,T_u) is in the transitive closure of the *dependent upon* relation. It is clear if malicious transaction *Bi* affects an innocent transaction G_j , the damage on B_i ’s write set will spread to G_j ’s write set. If an innocent transaction G_j is not affected by a malicious transaction, G_j ’s write set is not damaged. Therefore, the damage will be repaired if we back out every malicious or affected transaction.

To illustrate, consider the history as follows. B_1 affects G_2 but does not affect G_1. x and y are damaged but z is not. So if B_1 is malicious then backing out B_1 and G_2 can repair the damage caused on x and y. Note that G_2 need not be backed out.
H: (B_1 reads x) (B_1 writes x) (B_1 commits) (G_1 reads z) (G_2 reads x) (G_1 writes z) (G_2 reads y) (G_1 commits) (G_2 writes y) (G_2 commits)

2.2 The Algorithm

The approach, given a set of malicious transactions (denoted **B**) and a log that records every read and write operation, scans the log to identify every innocent transaction that is affected by **B**. When a malicious transaction is scanned, its write set will be marked *dirty*. When a committed innocent transaction is scanned, if it reads a *dirty* item, then its write set will be marked *dirty* and a specific UNDO transaction will be composed and executed. When a data item is *cleaned* by an UNDO transaction, it will not be marked dirty anymore. UNDO transactions are composed as follows. For every item that is updated by a transaction to be undone, if the item is not cleaned by any previous UNDO transaction, then a write operation that restores the item to the value before it was damaged will be added to the UNDO transaction.

2.3 Termination Detection

Since the algorithm allows users to continuously execute new transactions as the damaged items are identified and cleaned, new transactions can spread damage if they read a damaged but still unidentified item. So we face two critical questions: (1) Will the repair process terminate? (2) If the repair process terminates, can we detect the terminations? The general approach clearly answers these two questions. First, when the damage spreading speed is quicker than the damage repair speed, the repair process may never terminate. When the damage repair speed is quicker, the repair process will terminate. Second, under the following conditions we can ensure that the repair process will terminate: (1) every malicious transaction is repaired; (2) no item is marked dirty; (3) further scans will not identify any new damage.

3. ODAM

In this section, we will present the design of ODAM. ODAM is a tool that can do automatic on-the-fly damage assessment and repair for commercial database application. One of its main features is to locate and repair the damage caused by malicious but committed transactions or any transaction that is affected by a malicious transaction without the need to periodically halt normal transaction processing. ODAM is designed to be a general approach and therefore can be applied to many other database platforms for a wide variety of applications. The design of ODAM and the connectivity of the components are illustrate in Figure 2.

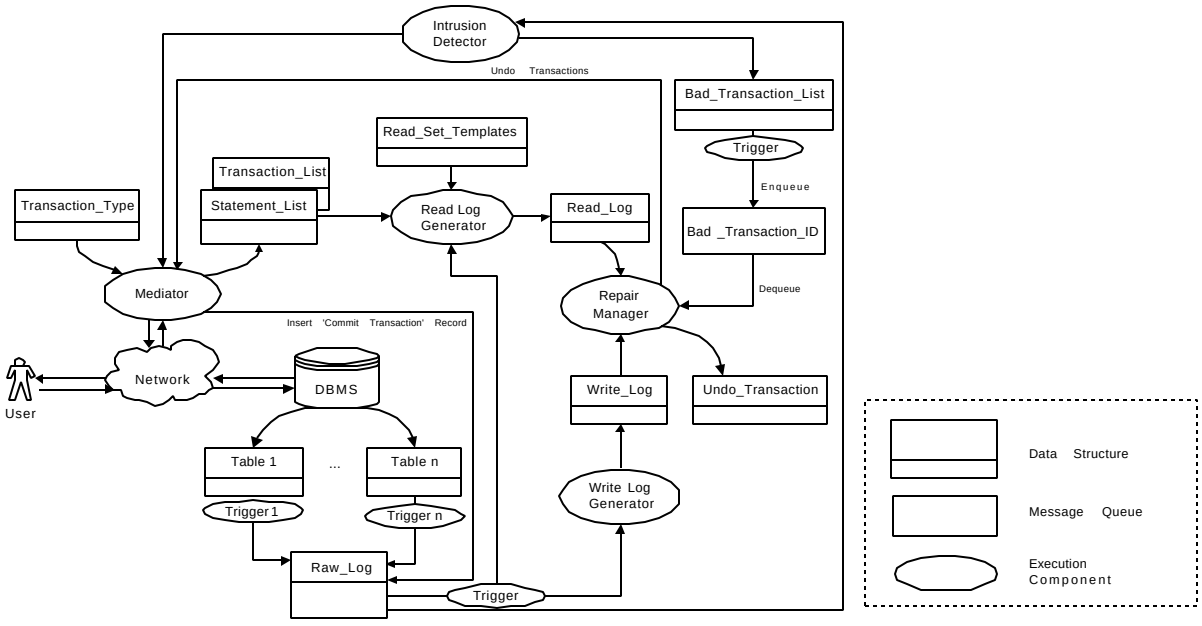

Figure 2: Design of ODAM

ODAM consists of following major components.

- The Triggers
- The Intrusion Detector
- The Mediator
- The Read Log Generator
- The Write Log Generator
- The Repair Manager

Each component performs a specific task and they typically need to interact with each other. In this section, the functionality, the procedures, and the relevant data structures of each component are discussed in detail.

3.1 Triggers

3.1.1 Functionality

Since Oracle Redo log structure is confidential and difficult to handle, we maintain read and write information by ourselves. In particular, we add a trigger to each user table to collect the information about the write operations on that table. When a record (of a user table) is updated, deleted, or inserted, the corresponding trigger is executed and all the write operations are recorded into the Raw_Log table. Besides user tables, we also associate triggers with the Raw_Log table and the Bad_Transaction_List table, which we will address shortly.

The trigger for the Raw_Log table is used to invoke the Read_Log Generator and the Write_Log Generator. Whenever the Mediator inserts a 'Commit' operation into the Raw_Log table, the trigger is executed. The trigger invokes the Read Log Generator and the Write Log Generator to generate the Read_Log and Write_Log data.

The trigger for the Bad_Transaction_List table is used to perform an 'enqueue' operation. When a new bad (or malicious) transaction is recorded in the Bad_Transaction_List table, the trigger will put the Transaction_ID of the bad transaction into the Bad_Transaction_ID queue.

3.1.2 Data Structure

- **Raw_Log Table**

The Raw_Log table keeps records of every write operation. Each record of the table has five fields: (1) Transaction_ID that uniquely identifies a transaction; (2) Item_ID. This is a composite field which consists of the Table_Name, Record_ID, and Field_Name; (3) Old_Value keeps the item's value before it was modified; (4) New_Value keeps the new item's value (5) Op_Type that indicates the categories of the command. The possible values include 'Insert', 'Update', 'Abort', and 'Commit'.

3.1.3 An Example

To illustrate, we give an example to show how the trigger associated with each user table works. In this example, ODAM is applied to a banking application. Suppose we have a transaction for updating the customer account balance. This transaction contains three types of operations: read, update and insert. The read operation is used to retrieve the account information of the customer. The update operation is used to change customer account balance in the Accounts table from $1,500.00 to $2000.00. The insert operation is used to record user transaction into the Transactions table. This information includes the ID of the user transaction, Account_ID, Amount, Post_Date, and Teller_ID. As described earlier, the Raw_Log stores information about write operations only. Thus, the trigger does not have a responsibility to operate the read operation. At the beginning of the process, we assume that the Raw_Log table is empty. After the user transaction is committed, the trigger for the user table assigns the Transaction_ID to this transaction and inserts the writes into the Raw_Log table. Results in Figure 3 show that each row contains write operation information for each item. Therefore, if the insert operation operates on many items, it will be recorded by many rows. The output in the Raw_Log, generated by the trigger, is shown in Figure 3.

Transaction_ID	Item_ID			New_Value	Old_Value	Op_Type
4.91.6240	Accounts	1174639205	Acc_Balance	1500.00	2000.00	Update
4.91.6240	Transactions	2833	ID		2833	Insert
4.91.6240	Transactions	2833	Account_ID		1174639205	Insert
4.91.6240	Transactions	2833	Amount		500.00	Insert
4.91.6240	Transactions	2833	Post_Date		21-Mar-00	Insert
4.91.6240	Transactions	2833	Teller_ID		Teller-1	Insert

Figure 3: Records in the Raw_Log generated by the trigger

3.2 Intrusion Detector

3.2.1 Functionality

The Intrusion Detector has a responsibility to detect and report malicious transactions to the Repair Manager. It uses the trails kept in the Raw_Log and some other relevant proofs to identify bad transactions. If a bad transaction is active when being identified, the transaction will be aborted and any further malicious transactions submitted by the malicious user will be rejected. However, if a bad transaction is already committed when being identified, Intrusion Detector will put the bad transaction's identifier into the Bad_Transaction_List table.

3.2.2 Data Structures

- **Bad_Transaction_List Table**

This table records the list of transactions that are detected as a malicious transaction. Each record contains the Transaction_ID filed which identifies the bad transaction.

- **Bad_Transaction_ID Queue**

This message queue stores information about bad transactions for the repair process. Messages are put into the queue by the trigger on the Bad_Transaction_List table. When a new Transaction_ID is recorded in the Bad_Transaction_List, the trigger will put the Transaction_ID of the bad transaction into the queue. These messages are retrieved by the Repair Manager to perform the repair operation.

3.3 Mediator

3.3.1 Functionality

In order to maintain read and write information, we developed an intermediate component, called the Mediator, to mediate every user transaction, and some specific triggers to log write operations. The main functions of the Mediator include:

(1) Provide services to clients for accessing the database.
Each of the client programs sends all SQL commands, including packages and procedure calls, via the Mediator to access the database. The Mediator proxies every user transaction.

(2) Capture most of the information about transactions' behavior.
The Mediator captures all the useful information about each command sent to the server. The information is then kept in the Statement_List table and the Transaction_List table.

(3) Identify the type of each in-coming transaction.
This function provides support for capturing read operations. The Mediator identifies the type of each in-coming transaction by matching the transaction's statements with the transaction patterns (or profiles) kept in the Transaction_Type table.

(4) Insert a 'Commit' record into the Raw_Log table.
Once a transaction commits, the Mediator inserts a 'Commit' record into the Raw_Log table to indicate the ending point of the transaction.

3.3.2 Data Structures

- **Transaction_List**

As described earlier, this table is maintained by the Mediator. It contains some important information

about transactions. The Mediator records log into the table when specific events occur in the transaction. The events include 'Start', 'Commit' and 'Abort'. Additionally, whenever the 'Start' event record is inserted, the record that indicates type of the transaction is inserted into the Transaction_List as well.

Each record of Transaction_List is composed of four major fields: (1) Transaction_ID; (2) Sesssion_No that keeps the identification number of the session used by a transaction; (3) Transaction_Status that keeps the status of events that occurs in a transaction; (4) Transaction_Type that identifies the type of transaction.

- **Statement_List**

This table records every client's SQL statement which is sent to the database. Each record of the Statement_List is composed of four major fields: (1) Transaction_ID; (2) SQL_Type that indicates the particular type of the statement; (3) Statement_Txt that keeps a completed content of the statement; (4) Statement_Seq that indicates the sequence of the statement in a transaction.

- **Transaction_Type**

The table keeps patterns and types of transactions. Each record of the Transaction_Type is composed of three major fields: (1) Pattern_Txt. This field contains a statement in a transaction that is used for matching operation; (2) Type. This field indicates the particular type of the transaction; (3) Compare_Len that specifies length of text in Pattern_txt to be compared when performing a matching operation.

3.3.3 An Example

We use an example to show how the Mediator works. As described earlier, the Mediator has a responsibility for recording the information of read and write operations into the Transaction_List and Statement_List. In addition, after a transaction is committed, the Mediator inserts a 'Commit' record into the Raw_Log. Figures 5, 6, and 7 show the data output by the Mediator in this example.

Pattern_Txt	Type	Compare_Len
SELECT Aac_balance FROM accounts WHERE Account_ID = :ID	A	50
SELECT Cust_name, Cust_addr FROM customers WHERE Cust_ID = :ID	B	50
SELECT Teller_name FROM tellers WHERE Teller_ID = :ID	C	50

Figure 4: Example of records in the Transaction_Type

Transaction_ID	Session_No	Transaction_Status	Transaction_Type
4.91.6240	21253	Start	
4.91.6240	21253	Type	A
4.91.6240	21253	Commit	

Figure 5: Example of records in the Transaction_List maintained by the Mediator

Transaction_ID	SQL_Type	Statement_Text	Statement_Seq
4.91.6240	SELECT	SELECT Acc_balance FROM accounts WHERE Account_ID = 1174639205	1
4.91.6240	UPDATE	UPDATE accounts SET Acc_balance = Acc_balance+500	2
4.91.6240	INSERT	INSERT INTO transaction (ID, Account_ID, Amount, Post_Date, Teller_ID) values (2833, 1174639205, 500.00, '21-Mar-00', 'Teller-1')	3

Figure 6: Example of records in the Statement_List maintained by the Mediator

Transaction_ID	Item_ID			New_Value	Old_Value	Op_Type
4.91.6240	Accounts	1174639205	Acc_Balance	1500.00	2000.00	Update
4.91.6240	Transactions	2833	ID		2833	Insert
4.91.6240	Transactions	2833	Account_ID		1174639205	Insert
4.91.6240	Transactions	2833	Amount		500.00	Insert
4.91.6240	Transactions	2833	Post_Date		21-Mar-00	Insert
4.91.6240	Transactions	2833	Teller_ID		Teller-1	Insert
4.91.6240						Commit

Figure 7: Records in the Raw_Log after the 'Commit' operation is inserted

Suppose ODAM has three types of transaction identified by 'A', 'B', and 'C' (see Figure 4) and the operation continues from the example in section 3.1.3. After the user performs read and write operations, all SQL statements in the user transaction, including SELECT, UPDATE, and INSERT, will be recorded into the Statement_List (see Figure 6). After that, the Mediator inserts three records about events of the transaction, 'Start', 'Commit' and the type of transaction, into the Transaction_List table (see Figure 5). The type of transaction is obtained by matching the first SQL statement in the user transaction (or any

other specific statement in the transaction) with the pattern stored in Pattern_Txt field in the Transaction_Type table. Before performing the pattern matching, the Mediator has to substitute the variable in Pattern_Txt (a variable is denoted by a colon (:) immediately followed by a name) with the value extracted from statements in the transaction. For example, the variable :ID, in the first row of Transaction_Type table, will be substituted with '1174639205' (Account_ID value). The value is extracted from the SELECT statement in the transaction. After the Mediator successfully performs a matching, the type of transaction will be searched out. In this example, the type of transaction, obtained from the matching process, is 'A'.

3.4 Read Log Generator

3.4.1 Functionality

The Read Log Generator maintains the Read_Log table to keep track of all the read operations issued by a SELECT statement. Once a 'Commit' operation is inserted into the Raw_Log table, the trigger on the Raw_Log will invoke the Read Log Generator. To capture read operations, since triggers can capture write operations only, we take the approach of extracting read sets from SELECT statements instead. To support this, first, we let the Mediator identify the type of each in-coming transaction. This can be done by matching each statement in a transaction with the patterns kept in the Transaction_Type table. Second, we let the Mediator record each SELECT statement and the relevant arguments in the Statement_List table. The Mediator also keeps the type, status and other information of the transaction in the Transaction_List table. Third, we let the Read Log Generator use the transaction type information and the arguments to materialize the relevant templates maintained in the Read_Set_Templates table. Finally, Read Log Generator retrieves read information from the Statement_List, and generates them into the Read_Log table.

3.4.2 Data Structures

- **Read_Log Table**

This table logs read operations of transactions. Each record has three fields: (1) Transaction_ID that uniquely identifies a transaction; (2) Item_ID. (3) Time_Stamp that indicates the time when the record is inserted into the Read_Log table.

- **Read_Set_Templates**

This table keeps the read operation patterns of transactions. Each record has four fields: (1) Transaction_Type that indicates the particular type of the transaction (2) Statement_Seq. A transaction may consist of many SELECT statements. Therefore, this field is used to identify the sequence number of SELECT statement in the transaction. (3) Field_Seq that is used to identify the sequence number of field in the SELECT statement. (4) Field_Name that identifies name of the field in the SELECT statement.

3.4.3 An Example

We use an example to show how the Read Log Generator works. Suppose this operation continues from the example in section 3.3.3. As described earlier, whenever a 'Commit' operation is inserted into the Raw_Log table, the Read Log Generator is invoked by the trigger on the Raw_Log. In this example, the Transaction_ID, which contains the value '4.91.6240', is passed as a parameter. By using the Transaction_ID, the Read Log Generator searches the LSN of the transaction from the Write_Log. However, the Write_Log is empty now. Therefore, the Read Log Generator has to generate the LSN value itself. Suppose the generated value is 100. With the LSN = '100' and the Op_type value = 'Start', the Read Log Generator inserts the first record of the transaction into the Write_Log table (see Figure 9).

In order to log the read operation into the Read_Log, we need to know the value of LSN, and Item_ID. As described earlier, the Item_ID is composed of Table_Name, Record_ID, and Field_Name. The Table_Name and Record_ID can be extracted from the user SELECT statement but the Field_Name can be retrieved from the Read_Set_Templates by identifying the Transaction_Type and the Statement_Seq. For the Transaction_Type, we can obtain it from the Transaction_List by searching with Transaction_ID = '4.91.6240' and Transaction_Status = 'Type'. As a result, Transaction_Type = 'A' is returned (see Figure

5). For the Statement_Seq, we obtain the value from the Statement_Seq field in the Statement_List table. In this example, the Statement_Seq of the read operation = '1'. With Transaction_Type = 'A' and Statement_Seq = '1', the Read Log Generator gets all field name specified in the read operation from the Read_Set_Templates (see Figure 8). After combining the value of Table_Name, Record_ID, and Field_Name into the Item_ID, the Read Log Generator will set Time_Stamp to current time. Finally, the reads operations are record into the Read_Log as shown in Figure 10.

Transaction_Type	Statement_Seq	Field_Seq	Field_Name
A	1	1	Acc_Balance
B	1	1	Cust_Name
B	1	2	Cust_Addr
C	1	1	Teller_Name

Figure 8: Example of records in the Read_Set_Templates

LSN	Transaction_ID	Item_ID	New_Value	Old_Value	Op_Type	Time_Stamp
100	4.91.6240				Start	21-Mar-00

Figure 9: Records in the Write_Log generated by the Read Log Generator

LSN	Transaction_ID	Item_ID			Time_Stamp
101	4.91.6240	Accounts	1174639205	Acc_Balance	21-Mar-00

Figure 10: Records in the Read_Log generated by the Read Log Generator

3.5 Write Log Generator

3.5.1 Functionality

The Write Log Generator maintains the Write_Log table to capture write operations involved in an INSERT, UPDATE, or DELETE command. Similar to the Read Log Generator, the Write Log Generator is invoked by the database trigger associated with the Raw_Log table whenever a 'Commit' record is inserted into the Raw_Log table.

3.5.2 Data Structures

- **Write_Log Table**

This table logs write operations. Each record has seven fields: (1) LSN that denotes the log serial number of a write record in a transaction; (2) Transaction_ID; (3) Item_ID; (4) Old_Value that keeps the item's value before it was modified; (5) New_Value that keeps the new item's value (6) Op_Type that indicates the categories of the statement. The possible values include: 'Start, 'Insert', 'Update', 'Abort', and 'Commit'. (7) Time_Stamp that indicates the time when the record is inserted into the Write_Log table.

3.6 Repair Manager

3.6.1 Functionality

The Repair Manager has a responsibility to perform on-the-fly damage assessment and repair. To do the job, first, the Repair Manager retrieves a message from the Bad_Transaction_ID queue. The message contains the Transaction_ID that is used to identify the bad transaction. Then, the Repair Manager scans the growing logs of on-the-fly histories to mark any bad as well as suspect transactions. After that, the Repair Manager rollbacks all changes made by the bad or suspect transaction by preparing undo commands and sending them to the Oracle Scheduler via the Mediator. After a 'success' return code is sent back from the Mediator, the Repair Manager records the information of all undo transactions and its Time_Stamp into the Undo_Transaction Table.

3.6.2 Data Structure

The Repair Manager uses the following major data structures:

- *tmp_item_set*, an array of records that is used to keep information about an item that may have been damaged. Each record of the array contains the Item_ID field. Item_ID is a composite field that consists of the Table_Name, Record_ID, and the Field_Name.
- *cleaned_item_set*, an array of records that is used to keep information about items that have been cleaned. Each record of the array has two fields: (1) LSN that denotes log serial number of the bottom record of the log at the time when the item is cleaned; (2) Item_ID.

- *dirty_item_set.* This array keeps information of every data item that is marked as dirty. Each record of the array contains the Item_ID field.
- *tmp_undo_list.* This array keeps information about every in-repair transaction that has read some data item in the dirty_item_set or satisfy others criteria which will be discuss in detail later. Each record of the array contains the Transaction_ID field.
- *B.* This array stores Transaction_ID which is retrieved from the Bad_Transaction_ID queue. The Transaction_ID will be removed from B after finishing repair the bad transaction.
- *Undo_Transaction* Table. This table records the list of transactions that have been undone. Each record contains the Transaction_ID field.

3.6.3 Procedures

The Repair Manager is composed of five sub-procedures: Main, On-the-Fly Repair, Undo-Transaction, Read Operation, and Write Operation procedure. Details of each procedure are describes below.

- **Main Procedure**

The Main procedure is utilized by the Repair Manager to perform various functions including invoking other sub-procedures. Such functions include:

(1) Initialize values of variables including tmp_undo_list, cleaned_item_set, dirty_item_set, tmp_item_set by setting these variables to null.
(2) Scan the Bad_Transaction_ID Message Queue. The Main procedure is responsible for periodically polling the Message Queue for incoming message that contains the bad Transaction_ID. After receiving the message, the Main procedure adds the Transaction_ID to B.
(3) Invoke On-the-Fly Repair procedure to perform repair operations. In order to do this task, the Main procedure must pass an output parameter specifying the bad Transaction_ID to the On- the-Fly Repair procedure. This task is performed repeatedly until the termination conditions are satisfied. The conditions are: (a) every bad transaction (stored in B) has been undone; and (b) the dirty_item_set is empty; and (c) the tmp_undo_list is empty; and (d) the LSN of each item stored in the cleaned_item_set is not less than the LSN of the next Write_Log entry to scan.

- **On-the-Fly Repair Procedure**

The goal of this procedure is to mark any bad as well as suspect transactions on the growing logs of on-the-fly histories. The procedure accepts the Transaction_ID, passed from the Main procedure, as an input parameter. The parameter is used to specify records of a transaction in the Write_Log and the Read_Log table. Once receiving a call, the On-the-Fly Repair procedure retrieves all records in the Write_Log table starting from the record that has the Transaction_ID equal to the input parameter. For each data item in those records, the procedure performs the following operations:

(1) Invoke the Write Operation procedure to process each write entry in the transaction, This operation will be performed, if the item is not marked as the dirty item and the entry is an ‘Insert’ or ‘Update’ operation.
(2) Retrieve all records from the Read_Log and invoke the Read Operation procedure to process all read entries in the transaction. This operation will be performed whenever the following criteria are satisfied:
    2.1. The item is marked as the dirty item and the operation type of the entry is a ‘Commit’ operation.
    2.2. The item is not marked as the dirty item and the entry is an ‘Update’ operation.
(3) Perform the undo transaction operation by calling the Undo Transaction Procedure. The procedure will be called, if the operation type of the entry is a ‘Commit’ operation and the following criteria are satisfied:
    3.1. The item is marked as the dirty item.
    3.2. The item is not marked as the dirty item but it is found in the tmp_undo_list.
(4) Maintain the dirty_item_set to keep information of every data item that is marked as dirty. The item will be added to the dirty_item_set, if the operation type of the item is ‘Update’ or ‘Insert’ and it is marked as the dirty item but it has not been cleaned.

(5) Maintain the tmp_item_set and the tmp_undo_list. If the item in a transaction is not marked as the dirty item and the operation type of the entry is 'Abort', all data items of the transaction will be deleted from the tmp_item_set and the Transaction_ID is also deleted from the tmp_undo_list.

- **Undo Transaction Procedure**

The goal of this procedure is to undo all changes made by a specific transaction. To perform the operation, all write records within the transaction are retrieved from the Write_Log table. The 'Insert' operation can be undone by deleting the corresponding record from the application table. While the 'Update' operation can be undone by rollback the current value in the application table to the old value that are kept in the Write_Log table. The Undo Transaction procedure also has a responsibility to maintain the submitted_item_set by recording some details of every data item whose undo operation has been submitted to the Mediator but the item still has not been cleaned.

- **Read Operation Procedure**

This procedure maintains data in the tmp_undo_list that is used to keep some information about in-repair transactions that has read some data item in the dirty_item_set, or has read an item in the cleaned_item_set. The data item will be added to the tmp_undo_list, if the log serial number of the last current record in the Write_Log is greater than the log serial number of the read record. Data in the tmp_undo_list are used by On-the-Fly Repair procedure to determine whether a transaction has to be performed the undo operation or not.

- **Write Operation Procedure**

This procedure maintains the tmp_item_set to temporally keep suspect data items that might be marked as dirty items later. The data item will be added to the tmp_item_set, if it is found in the cleaned_item_set, and the log serial number of the item is greater than the log serial number of the last current record in the Write_Log table.

4. Testing ODAM

In this section, we present the testing results of three experiments designed to examine the performance of On-the-Fly Repair process, presented in the previous section. In Experiment 1 we look at the effects of varying the average repair time under several dependency degree. In Experiment 2 we look at the effects of varying the average repair time under several number of transactions. In Experiment 3 we compare throughput of the system when the Mediator and the Repair Manager are run and not run. Additionally, in order to have an idea about the processing speed of the Repair Manager when there are other processes running on the same system, in each experiment, we measure the performance at two stages of databases -- busy and not busy. In busy status, database was rarely idle. We simulated the situation by running ten concurrent users. Each user continuously accessed the database at the same time. Each transaction made by a user is composed of several UPDATE and SELECT statements. However, these transactions only increase workload for database and do not generate any logs into our system.

We performed the experiment by simulating transactions. Transaction data was generated by a simulator process called Transaction Simulator. Using the process, we can define our parameters used in the test. They are size of transaction, the number of transactions, dependency degree, the number of read and write operations, the number of concurrent users. We set the first simulating transaction to be a malicious one. Then, the next transaction is launched and so on. The Transaction_ID of the first transaction is inserted into the Message Queue table. Then the Repair Manager is notified and the repair process starts.

4.1 Test Environment

For our testbed, we use Oracle 8.1.6 Server to be the underlying DBMS (Note that ODAM is in general DBMS independent, thus ODAM can be easily adapted to tolerate the intrusions on a database managed by almost every "off-the-shelf" relational DBMS such as Microsoft SQL server, Informix, and Sybase). The Oracle Server, the Mediator, and the Intrusion Detector are running on a DELL dual Pentium 800MHZ CPU PC with 512MB memory. The Repair Manager and the Transaction Simulator are running on a Gateway PC with a 600MHZ Pentium CPU and 128MB memory. These two PCs run Windows NT

4.0 and are connected by a 10/100Mbps switch LAN, however, the speed to the DELL Database Server is only 10Mbps. The current version of ODAM has around 20,000 lines of (multi-threaded) C++ code and Oracle PL/SQL code. Each component of ODAM is implemented as a set of C++ objects that have a couple of CORBA calling interfaces through which other components can interact with the component and the reconfiguration can be done. We use ORBacus V4.0.3 as the ORB. Finally, ODAM assumes applications use OCI calls, a standard interface for Oracle, to access the database, and ODAM proxies transactions at the OCI call level. The reason that ODAM does not proxy transactions at the TCP/IP or the SQL*NET level, which is more general, is because the exact packet structure of SQL*NET is confidential.

4.2 Average Repair Time vs. Dependency Degree

The objective of this experiment is to study the effects of changing the value of dependency degree to the average delay for cleaning damaged items. There are two kinds of repair time delay in our system:

1. The time between an item being damaged and the item being repaired by the Repair Manager.
2. The time between a damaged-item being reported by the Intrusion Detector and the item being repaired.

Figure 11 shows details of both time delays. The first time delay is used for measuring performance of both the Intrusion Detector and the Repair Manager while the second time delay is used for measuring performance of the Repair Manager only. In this experiment, we focus on the second time delay and we measure the time delay in seconds. In addition to such factors as database size and the number of transactions, the average repair time is heavily dependent on the dependency degree between transactions.

In this test, we simulate a sequence of transactions with a specific probabilistic dependency degree (between each pair of adjacent transactions). Each dependency degree number denotes size of the intersection of a transaction's write set and its previous adjacent transaction's read set.

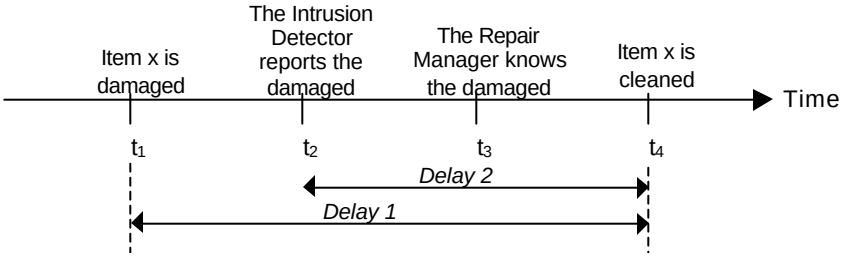

Figure 11: Time delays occur in the repair process

4.2.1 Average Repair Time vs. Dependency Degree in Not Busy Database

In this first experiment, we tested the Repair Manager's performance while no other transactions were processing except transactions created by the Transaction Simulator. Figure 12 summarizes the parameters and show the values used in this experiment. We simulate five concurrent users to run the transactions. The database contains 10,000 records. Each record contains 140 bytes of data. We simulate three type of transaction – large, medium and small. A large size transaction is composed of 15 read operations and 10 write operations. A medium size transaction is composed of 8 read operations and 4 write operations. A small size transaction is composed of 5 read operations and 2 write operations. Each type of transaction is simulated three times. The average values of the average repair time over dependency degree are plot in graphs shown in Figure 13.

The number of concurrent users: 5
Database size (records): 10,000
Record size (bytes): 140
The number of read operations (per transaction): 15 (Large), 10 (Medium), 5 (Small)
The number of write operations (per transaction): 10 (Large), 4 (Medium), 2 (Small)

Figure 12: Simulation parameter and settings

Results from graphs in Figure 13 show that dependency degree affects the performance of the system. For all three size of transaction, high dependency degree leads to high average repair time. This can be explained that higher dependency degree increases the possibility of affected transactions. Consequently, the increase in the number of affected transactions increases the average repair time.

Additionally, at the same dependency degree, performance suffers least when processing the small size transactions. Comparing the slope between three transaction sizes, we can see that the slope of the large size transaction is steepest among three size of transaction. The main reason is that more statements are needed to repair large size transactions. In addition, high average repair time for large size transactions may be due to data concurrency. Since we simulate this experiment with five concurrent users. There will be more chance for large size transactions that some records are simultaneous accessed by many transactions.

4.2.2 Average Repair Time vs. Dependency Degree in Busy Database

In this experiment, we measure the performance over dependency degree while the database is busy. Parameters used in this experiment are set to the same value as the previous one. Results are shown in Figure 13.

Compared to the previous experiment in not busy database, this one takes more average repair time. Average repair time increases 70-150%, 130-160%, and 145-165% for the large size transaction, medium size transactions, and small size transactions respectively. From the results, the Repair Manager's performance is affected most for the small size transactions.

Similar to the previous experiment, the slope of the large size transactions is steepest among three size of transaction. The results support the previous finding that the higher dependency degree increases more possibility of affected transactions in large size transactions than that in medium and small size transactions.

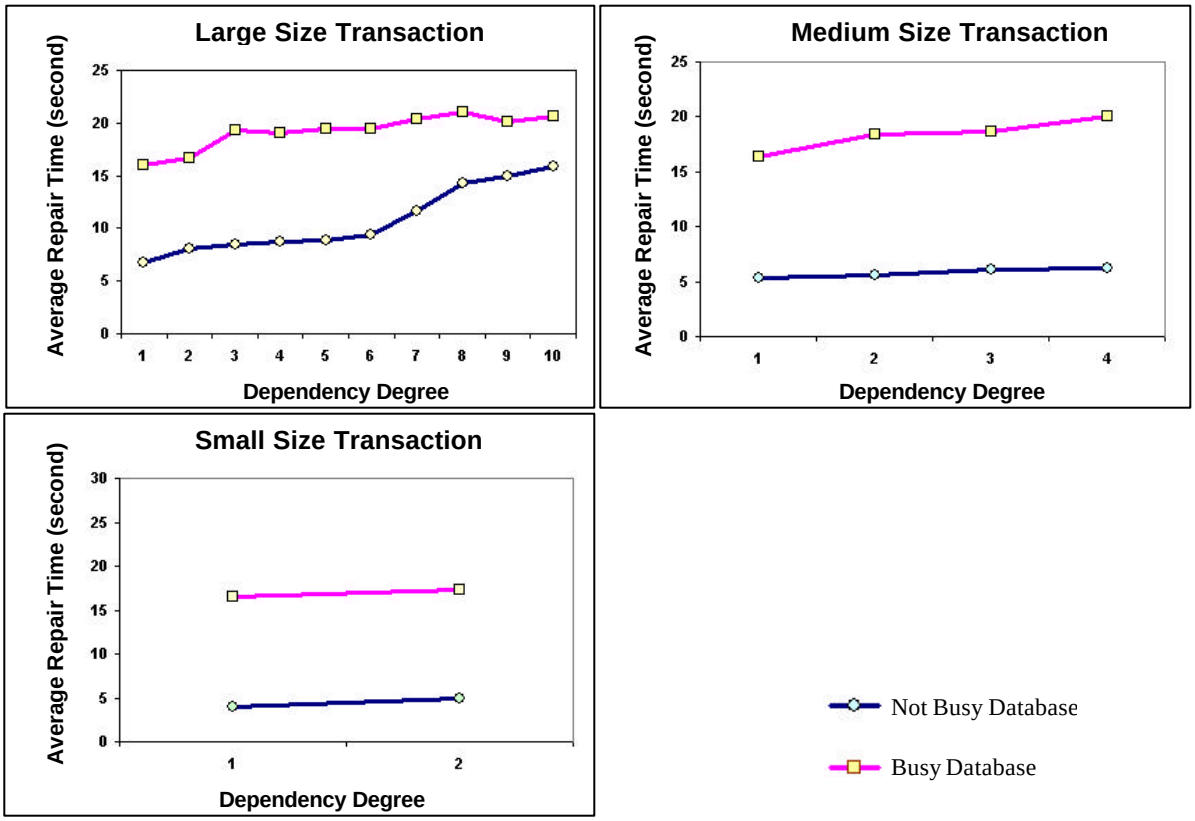

Figure 13: Average Repaired Time vs. Dependency Degree

4.3 Average Repair Time vs. the Total Number of Transactions

The objective of this experiment is to test the Repair Manager's performance by measuring average of time used for cleaning damaged items regarding several numbers of transactions. We measure the performance in two status of database – busy and not busy.

4.3.1 Average Repair Time vs. the Total Number of Transactions in not Busy Database

The simulation parameters used in this experiment are shown in Figure 14. These parameters are set to the same value as the previous experiment except we simulate only one concurrent user and we set value of dependency degree to 2. Graphs in Figure 15 displays average time used for repairing the damaged item over the total number of transactions.

The number of concurrent users: 1
Database size (records): 10,000
Record size (bytes): 140
The number of read operations (per transaction): 15 (Large), 10 (Medium), 5 (Small)
The number of write operations (per transaction): 10 (Large), 4 (Medium), 2 (Small)
Dependency Degree: 2

Figure 14: Simulation parameter and settings

Results from graphs in Figure 15 show that the performance of the repair process is reduced as the total number of transactions is increased. For all size of transaction, high total number of transactions leads to high average repair time. This is because higher the total number of transactions increases workload of the Repair Manager since the number of statements to be processed is increased.

Additionally, at the same total number of transactions, performance suffers least when processing the small size transactions. When comparing the slope between three transaction sizes, we can see that the slope of large size transactions is steepest among three size of transaction. This can be explained that the higher number of transactions increases more possibility of affected transactions in large size transactions than that in medium and small size transactions.

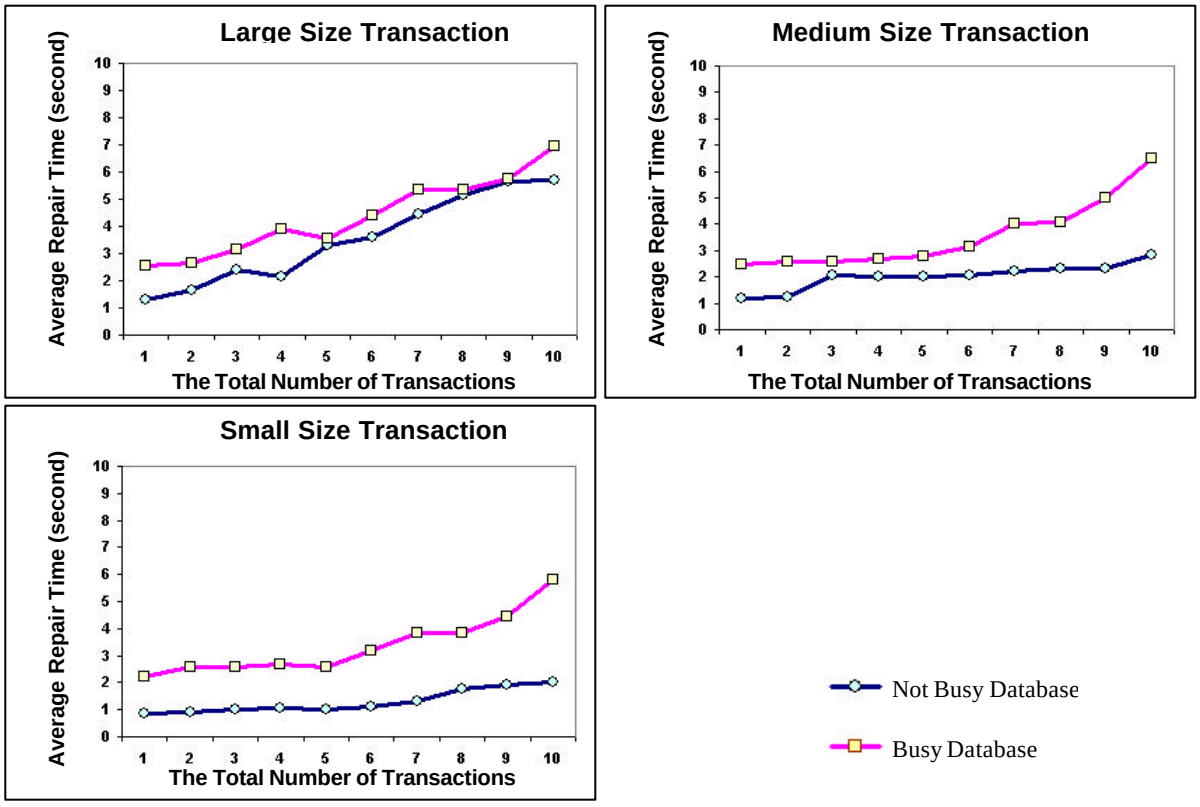

Figure 15: Average Repair Time vs. the Total Number of Transactions

4.3.2 Average Repair Time vs. the Total Number of Transactions in Busy Database

In this experiment, we measure the performance over the total number of transactions when the database is rarely idle. All parameters are set the same value as the previous experiment. Results, as shown in Figure 15, show that the repair process, in busy database status, takes more average repair time than not busy database status. Average repair time increases 2-95%, 25-130% and 120-190% for large size transactions, medium size transactions, and small size transactions respectively. Similar to the experiment in section 4.2.2, the Repair Manager's performance is affected most for small size transactions.

4.4 Impact on Throughput

The objective of this experiment is to evaluate the impact of ODAM on the throughput of transaction processing. This test is performed by comparing the average time used for executing a single transaction when ODAM is deployed with the average time when ODAM is not enforced. Note that the average time used for executing a transaction represents the *throughput* of a database server. Two tests are performed when the database is busy and when the database is not busy, respectively.

4.4.1 Throughput in not Busy Database

In this experiment, we measure the average repair time while no other transactions are processed except the transactions created by the Transaction Simulator. We simulate two type of transaction – large and small. Figure 16 summarizes the parameters and show the values used in this experiment. The average values of the average repair time when running and not running the Mediator and the Repair Manager over dependency degree are plot in graphs shown in Figure 17.

The number of concurrent users: 1
Database size (records): 10,000
Record size (bytes): 140
The number of read operations (per transaction): 15 (Large), 5 (Small)
The number of write operations (per transaction): 10 (Large), 2 (Small)

Figure 16: Simulation parameter and settings

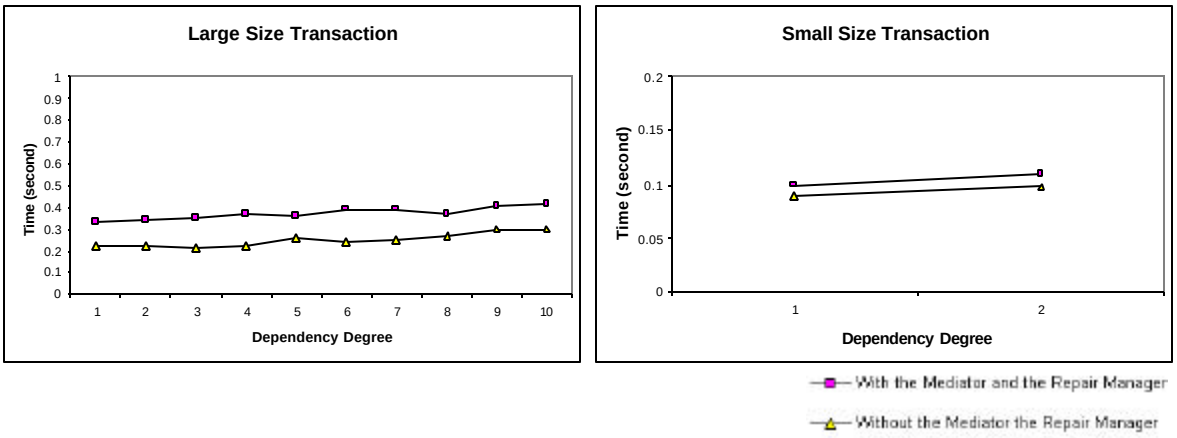

Figure 17: Throughput in not Busy Database

Results from graphs in Figure 17 indicate that the Mediator and the Repair Manage affects performance of the system. The average time used for executing a transaction increases between 37-68% for large size transactions, and 11-20% for small size transactions. This is because large size transactions have much more read and write statements than small size transactions and all statements have to be processed by the Mediator and the Repair Manager. Therefore, large size transactions have more performance penalty than small size transactions have. However, in real world applications where several read and write statements are usually combined into a single SQL statement, the response time delay should be much smaller than the results here.

4.4.2 Throughput in Busy Database

In this experiment, we study the impact of ODAM on the average transaction execution time. All parameters are set to the same value as the previous experiment. Results are shown in Figure 18.

Similar to the pervious experiment, performance of the system drops when running the Mediator and the Repair Manager. Results show that, in busy database, the average time used for creating a transaction increases for all sizes of transaction. It increases 30-50% for large size transactions, and 10-30% for small size transactions. The results support the previous finding that large size transactions have more performance penalty than small size transactions have.

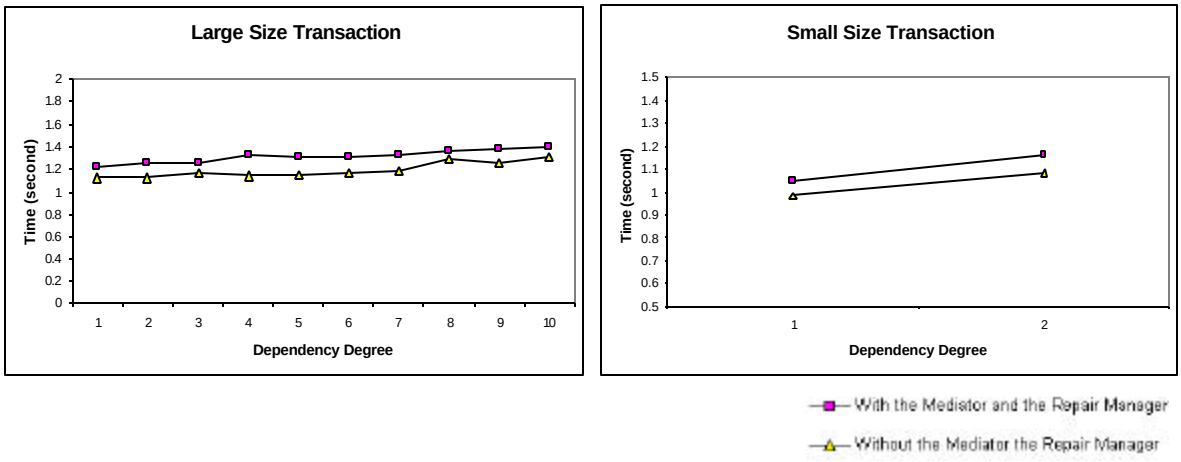

Figure 18: Throughput in Busy Database

5. Conclusion

In this paper, we present the design and implementation of ODAM, an on-the-fly damage assessment and repair tool for intrusion tolerant commercial database applications. Given a set of malicious transactions reported by an intrusion detector, ODAM can locate and repair the damage caused by the malicious transactions and every benign transaction that is affected by a malicious transaction. ODAM locates and repairs damage on-the-fly without the need to periodically halt normal transaction processing. Moreover, ODAM is transparent to on-going user transactions. The first prototype of ODAM, which is for Oracle Server 8.1.6, is tested using simulated data. The results show that the impact of ODAM on the performance of normal transaction processing is very small, which indicates that ODAM can be a practical solution towards providing “data integrity” guarantees to arbitrary commercial database applications in face of attacks. Finally, by implementing ODAM the conceptual soundness, the functional capabilities, and the practicality of the general on-the-fly damage assessment and repair approach have been validated.

There are some future works for ODAM. First, we want to re-test the prototype using real world data such as real world (fraud) credit card transaction data and inventory management data. Although the way we generate the simulated data is based on people’s experiences in real world database applications, we believe that using real world data can give us more insights about the effectiveness and performance overhead of ODAM. Second, currently after the ODAM tool is started, there is no other input to ODAM except the set of malicious transactions (from the Intrusion Detector) and the trails of transactions. Another module can be added to ODAM to provide more interactions between the SSO and ODAM. One desirable feature of this interface would be the capability of showing the SSO the transactions that are being repaired and the data items that are found damaged. Third, the security of ODAM is not addressed in the prototype implementation. However, successful attacks on ODAM can cause some undamaged data items to be mistakenly ‘cleaned’, and can cause some damaged data items to not be identified. For one example, if the triggers are disabled then ODAM is disabled. For another example, if the transaction profiles are maliciously modified then some unaffected benign transactions can be backed out. Therefore, the security of ODAM 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] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[3] 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.

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

[5] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Transaction on Database Systems*, 1(3):242-255, September 1976.

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

[7] 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.

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

[9] T. F. Lunt. A survey of intrusion detection techniques. *Computers & Security*, 12(4):405-418, June 1993.

[10] T. Lunt and C. McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corporation, McLean, VA, 1998.

[11] 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.

[12] 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.

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

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

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

[16] 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.

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

[18] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. Technical report, George Mason University, Fairfax, VA. Under Review for Journal publication.