REORGANIZATION OF THE DATABASE LOG FOR INFORMATION WARFARE DATA RECOVERY

Rumman Sobhan and Brajendra Panda
Computer Science Department, University of North Dakota

Abstract: Using traditional logs that contain only before and after images of data items and record neither any of the read operations nor actual mathematical or logical operations associated with an update, a complete damage assessment and repair of the database is not possible in the post Information Warfare scenario. In this research, a new logging protocol is proposed which records all appropriate information required for the complete repair of a database that was updated by committed but affected transactions. This information includes various predicates and statements used in the transaction. An algorithm to incorporate these in the log is offered. Based on this new log, a method to perform damage assessment and recovery is presented.

1. INTRODUCTION

The process of information sharing has gathered pace with the advent of high-speed networks and Internet technologies. The Internet has made every organization, government or industry, virtually connected to the entire world. This connectivity has created opportunities for intruders to access and possibly damage sensitive information. In many cases, the system cannot distinguish between a hacker and a legitimate user, and makes the malicious updates permanent in the database.

Existing intrusion detection methods cannot always successfully detect an attack immediately [12]. An attack can therefore go unnoticed for several days even months in some situations. In the mean time, valid transactions can read the damaged data and update other items, thus, spreading the damage in the database. This can have a cascading effect over time.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35587-0_24
M. S. Olivier et al. (eds.), *Database and Application Security XV*
© IFIP International Federation for Information Processing 2002

Therefore, during damage assessment and recovery, it is crucial to detect all affected transactions and undo their effects along with the attacking transaction's updates, then, re-execute these affected transactions.

The scope of all conventional recovery protocols, ([2], [3], [5], [8], [11]), for example, are confined only to recovery from system and hardware failures. These mechanisms depend heavily on traditional logs. Traditional logs record only the before and after images of updates, and none of the read operations. The log is also purged from the system from time to time. For recovery in a post information warfare scenario, identities of all read items are required to determine the "read-from" relationships for complete damage assessment [7]. In addition, for complete repair of the database, all affected transactions must be re-executed to reach a final consistent state, which would have been achieved if there were no attack [9]. Therefore, for re-execution purposes, all operations of all transactions need to be recorded in the log. Furthermore, the log must be available to the recovery module in its entirety - no part of it can be purged at any time.

Some researchers have proposed new models and novel recovery techniques to handle electronic attacks on databases. Among them, [1] followed a transaction dependency approach. Panda and Giordano [13] adopted a data dependency approach to recover from malicious attacks. Patnaik and Panda [14] developed a log clustering method to make the recovery process efficient. However, these models are based on the assumption that the log maintains detailed information about transaction operations, which are crucial for carrying out UNDO and REDO activities of committed transactions. On the contrary, no such log model exists as yet.

In this paper, we have presented a model to restructuring the logging protocol. This protocol facilitates storage of all information necessary for complete damage assessment and recovery of an affected database in a convenient and efficient manner. Based on this new approach, we have developed a method to identify damaged transactions and recover the database to a consistent state, which would have been there had there been no attack. The next section outlines our model. Section 3 presents the logging protocol. Damage Assessment and Recovery algorithms are provided in section 4. Section 5 offers the conclusion of the paper.

2. THE MODEL

2.1 Transaction Model

This model agrees with that presented in [4]. In our model, a transaction is visualized as a program. For each transaction, the information will include one or more conditions, statement(s) associated with each condition, and

before and after images of any data item that is updated by the transaction. The model formalizes the transaction program in terms of predicates and statements. A predicate is a precondition that must be evaluated to be true for the statements associated with it to be executed.

Traditionally, a predicate is attached to or followed by a set of statements. The set can be a singleton or otherwise. The statements are, in general, semantically related in some way and are organized to achieve a goal or part of a goal in a unified fashion. The proposed model defines the cluster of one such predicate and its following statement(s) as a *Predicate Statement Block.* A Predicate Statement Block is representative of a "branch" in a multi-branch IF – ELSE IF – ELSE structure where the branch contains a predicate (the condition) and the statements that are associated with it. In any IF – ELSE IF – ELSE structure, multiple Predicate Statement Blocks can be connected to each other in an exclusive conditional fashion. Our model defines such a collection of Predicate Statement Blocks as a *Predicate Block.* A transaction can comprise of one or more Predicate Blocks. It must be noted that an IF block can have one or more nested IF blocks. The proposed model requires that such nested blocks are simplified to a basic multi-branch IF – ELSE IF – ELSE structure by the database system. The model also requires that a LOOP structure in a transaction be decomposed into a combination of IF structure (to check the loop condition) and GOTO statement (to continue looping).

A Predicate Block can be of two kinds: *conditional* or *unconditional.* In a conditional Predicate Block, each of the branches has a predicate. The boolean value of the predicate must be evaluated before the associated statements can be executed. If the value is true then the statements are executed. Otherwise, the predicate of the next branch is evaluated. However, in an unconditional Predicate Block, there is always only one Predicate Statement Block and an explicit predicate is absent. Here, the database system assumes a virtual predicate and it always evaluates to true. Due to the constant true value of the predicate, in an unconditional Predicate block, the statements are always executed, unless, the transaction is aborted before their execution. These concepts are explained in the example 1 below. This example shows a typical transaction, say, T_1 with two Predicate Blocks. Start and end of the blocks are indicated in the example. Block 1 is a conditional Predicate Block. It comprises of three Predicate Statement Blocks. Each Predicate Statement Block has its own predicate. It must be noted that the third Predicate Statement Block of Predicate Block 1 has an implicit predicate of ($x > 5$). The model requires that it be generated by the system by complementing the other predicates of the target Predicate Block. Predicate Block 2 in the above example is an instance of an unconditional Predicate Block. There is only one Predicate Statement Block, and it does

not have an explicit predicate. Here, a true predicate is assumed and therefore, the statements are always executed.

Example 1: T_1 : /* Start of Predicate Block 1 */
If ($x < 5$) then { $a = a + 1$; $b = b + 1$; }
Else If ($x == 5$) then { $a = a - 1$; $b = b - 1$; }
Else { $a = 0$; $b = 0$; } /* Implicit predicate ($x > 5$) */
/* End of Predicate Block 1 */
/* Start of Predicate Block 2 (unconditional) */
{ $p = p - 1$; $q = 100$; $r = p + q$; }
/* End of Predicate Block 2 */

The proposed model requires that each transaction is broken into one or more predicate blocks. Then identifiers are assigned to the corresponding predicates and statements. Here, the predicate and the statement IDs require some explanation. A transaction may have multiple Predicate Blocks. Each block can contain multiple Predicate Statement Blocks. Each of these Predicate Statement Blocks has one predicate (implicit or explicit), and one or more statements. In the rest of the paper, T_i represents the i-th transaction, $T_i.P_{j.k}$ denotes the k-th predicate in the j-th Predicate Block of transaction T_i, and $T_i.P_{j.k}.S_l$ represents l-th statement of predicate k, of j-th Predicate Block, which belongs to transaction T_i. Moreover, $T_i.P_j$ represents Predicate Block j of transaction T_i. The identification scheme is shown with the help of the following example.

Example 2 T_2 : If (($x > 5$) AND ($y < 15$)) then
{ $u = t + u + 1$; $v = t + v - 1$; }
If ($a < 10$) Then {$w = w * 25$;}
Else $w = w / 25$; /* implicit predicate (a >= 10) */

The transaction in Example 2 has two Predicate Blocks. Each of the blocks has only one Predicate Statement Block. The identifier of the transaction is T_2. The identifiers of the predicates and statements are shown in Table 1. It must be noted that the predicate (a >= 10) is generated by the system by complementing the predicate (a < 10). The model defines the terms "Read_Set" and "Write_Set", and provides them with special usage. The Read_Set refers to the set of data items that are read in the execution of a statement. Similarly, the Write_Set refers to the set of data items that are updated in a statement. When used with an argument, these terms can be used as functions. In Example 2, Read_Set($T_2.P_{1.1}$) represents the set $\{x, y\}$ and Read_Set($T_2.P_{1.1}.S_1$) represents the set $\{t, u\}$. The Write_Set($T_2.P_{1.1}.S_1$) refers to the set $\{u\}$. It must be noted that the Read_Set defined in this research is applicable to a predicate or a statement, and not a transaction. The Write_Set applies to only a statement. Naturally, a Read_Set can contain multiple members, while a Write_Set is always a singleton.

A Predicate Statement Block mostly consists of update type statements each of which is basically an assignment statement where zero or more number of data items are read and only one data item is written. It has the general form $x = f(I)$, i.e., the values of the data items in set I are used in calculating the new value of x. Based on the semantic content of the right side of a statement, there can be two cases:

Case I : $I = \phi$. This means that no data items were read to calculate the value of x. That is, the statement is of the form: $x = c$, where c is a numeric or string constant. We define it as a direct update. A simple example is $x = 5$.

Case II : $I \neq \phi$. This means that one or more data items were read to calculate the new value of x. The data item x may or may not belong to the Read_Set of that statement. We define it as a calculated update. An example would be $x = x + y - 10$.

Table 1. Demonstration of the Identifier Scheme

Predicate/Statement	ID
Predicate: $((x > 5)$ AND $(y < 15))$	$T_2.P_{1.1}$
Statement: $u = t + u + 1$;	$T_2.P_{1.1}.S_1$
Statement: $v = t + v - 1$;	$T_2.P_{1.1}.S_2$
Predicate: $(a < 10)$	$T_2.P_{2.1}$
Statement: $w = w * 25$;	$T_2.P_{2.1}.S_1$
Predicate: $(a >= 10)$	$T_2.P_{2.2}$
Statement: $w = w / 25$;	$T_2.P_{2.2}.S_1$

2.2 Database Model

In this research, the database system model presented in [2] is followed. The logging, damage assessment, and recovery models proposed require that the functionality of the transaction manager and recovery manager be enhanced to some degree. Our model requires that the transaction manager accepts the operations, the predicates, and the statements from transaction programs. It is also expected that in addition to delegating identifiers for transactions, the database system issues identifiers for Predicate Statement Blocks and statements as well. In the traditional database model, the transaction manager sends each operation tagged with the transaction identifier to the scheduler. The scheduler either directly sends it to the data manager for immediate execution, delays its execution by putting it on a queue, or rejects it. The recovery manager accepts an operation from the scheduler and records it in the log. The proposed model requires that the predicates and statements, along with their identifiers be sent from the transaction manager, through the scheduler, to the recovery manager.

2.3 Assumptions

Our model uses a log sequence number (LSN) for each log record and follows the write-ahead-logging protocol as described in [10]. The model also assumes that the system follows the STEAL/NO FORCE protocol [6]. As a result, it keeps both Undo and Redo type log records. It also requires that the checkpoint activities are performed periodically and are recorded in the log. We assume the use of cache consistent checkpointing [2] for our proposed model as this does not impose delay on transaction execution and it reduces and simplifies restart activities better than any of the other checkpoint methods. Some other basic assumptions made regarding the operations of the database system are: (1) the scheduler produces rigourous serializable history [3], (2) the log is not modifiable by users, (3) no part of the log can be purged at any time, (4) nested transactions are not allowed, and (5) a transaction writes a data item to the stable database only once. The next section presents the proposed logging protocol.

3. THE LOGGING PROTOCOL

An integral part of our logging method is the different kinds of information that need to be recorded in the log. The proposed protocol requires the recording of the following entries:

1. [Start_Transaction, T_i]: Denotes that transaction T_i started execution.
2. [Record_Predicate $T_i.P_{j.k}$, *Predicate_String*] : Predicate k of Predicate Block j of transaction T_i is being recorded. *Predicate_String* represents the actual predicate.
3. [Record_Statement, $T_i.P_{j.k}.S_l$, *Statement_String*] : Statement l of predicate $j.k$ belonging to transaction T_i is being recorded. *Statement_String* represents the actual statement.
4. [Read_Item, T_i, x] : Records that T_i read the value of database item x.
5. [Write_Item, $T_i.P_{j.k}.S_l$, x, *old_value*, *new_value*] : Records that transaction T_i has updated the value of the data item x of Statement $T_i.P_{j.k}.S_l$. Both before image (*old_value*) and after image (*new_value*) of the data item are recorded.
6. [Undo, $T_i.P_{j.k}.S_l$, x, *undo_value*] : Transaction T_i has aborted (due to transaction failure or system crash) and the data item x of statement $T_i.P_{j.k}.S_l$ was flushed to the stable database. The data item must be reset to the *old_value* of the write operation log record it is attempting to undo. The *undo_value* represents the *old_value*.
7. [Redo, $T_i.P_{j.k}.S_l$, x, *redo_value*] : The system has crashed after Transaction T_i has committed, but the data item x of statement $T_i.P_{j.k}.S_l$

was not flushed to the stable database. The data item must be changed to the *new_value* of the write operation log record it is attempting to redo. The *redo_value* represents the *new_value*.

8. [Flush, x, LastUpdatePtr] : Records that the cache slot containing data item x is flushed to the stable database. The LastUpdatePtr represents the LSN of the log record that last updated x. It is the only log record that the cache manager communicates to the recovery manager.
9. [Commit, T_i] : Records that T_i has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
10. [Abort, T_i] : Records that transaction T_i has been aborted.

It must be noted that the Undo and Redo log entries in this protocol do not hold the traditional connotation of log entries of "update" activity of data items described in [6]. Instead, the Undo and Redo log entries in this protocol refer to actual undo and redo operations that follow a transaction failure or a restart after a system failure. The main goal of the logging protocol is to record all scheduled operations in the log. Obviously, the different operations include start of a transaction, read and write operations on data items, undo and redo operations on updates, flushing of pages to the stable database, commit, and abort operations of transactions. The recovery manager, which is responsible for maintaining the logging protocol must methodically identify each operation and write a log entry for it. If the activity is a start, flush, redo, undo, commit or abort operation of a transaction, then the recovery manager will write Start_Transaction, Flush, Redo, Undo, Commit, or Abort type log record respectively. In addition, one of the principal parts of the logging mechanism is to record a Predicate Block in terms of its predicates and statements. This is performed at the arrival of the first read or write operation of a Predicate Block. A Predicate Block can be recorded in only three specific cases. All these three cases represent the first operation from that Predicate Block.

Case 1: The current operation is a "read" and the data item belongs to a predicate.

Case 2: The current operation is a "read" and the data item belongs to a statement. The Predicate Block is unconditional.

Case 3: The current operation is a "write". The operation represents a direct update from an unconditional Predicate Block.

Whenever a read or write operation is scheduled it must be checked whether the Predicate Block has already been recorded. If not, it is recorded in the log in terms of its predicates and statements. The read or write record is written after that. Next, we present the logging algorithm based on the above-proposed protocol.

3.1 Logging Algorithm

1. If (Operation = = Start transaction T_i) then
   1.1 Write the record [Start_Transaction, T_i] in the log
2. Else If (Operation = = Read the data item x of transaction T_i) then
   2.1 If ($x \in$ Read_Set(Predicate $T_i.P_{j.k}$)) then
   2.1.1 If the Predicate Block j has not yet been recorded then
   2.1.1.1 Repeat until all Predicate Statement Blocks of Predicate Block $T_i.P_j$ have been processed
   2.1.1.1.1 Write the record [Record_Predicate, $T_i.P_{j.k}$, *Predicate_String*] in the log
   2.1.1.1.2 Record each statement associated with predicate $T_i.P_{j.k}$ in the form [Record_Statement, $T_i.P_{j.k}.S_l$, *Statement_String*]
   2.2 If ($x \in$ Read_Set (Statement $T_i.P_{j.k}.S_l$)) then
   2.2.1 If the Predicate Block j has not yet been recorded then
   2.2.1.1 Record each statement associated with Predicate Block j in the form [Record_Statement, $T_i.P_{j.k}.S_l$, *Statement_String*]
   2.3 Write the record [Read_Item, T_i, x]
3. Else If (Operation = = Write the data item x of statement $T_i.P_{j.k}.S_l$) then
   3.1 Perform Step 2.2.1
   3.2 Write the record [Write_Item, $T_i.P_{j.k}.S_l$, x, *old_value*, *new_value*]
4. Else If (Operation = = Undo the update of data item x of statement $T_i.P_{j.k}.S_l$) then
   4.1 Write the record [Undo, $T_i.P_{j.k}.S_l$, x, *undo_value*]
5. Else If (Operation = = Redo the update of data item x of statement $T_i.P_{j.k}.S_l$) then
   5.1 Write the record [Redo, $T_i.P_{j.k}.S_l$, x, *redo_value*]
6. Else If (Operation = = Flush the cache slot containing data item x) then
   6.1 From x's cache slot, obtain the LSN of the log record that last updated data item x
   6.2 Write the record [Flush, x, LastUpdatePtr]
7. Else If (Operation = = Commit the transaction T_i) then
   7.1 Write the record [Commit, T_i]
8. Else If (Operation = = Abort the transaction T_i) then
   8.1 Write the record [Abort, T_i]

4. DAMAGE ASSESSMENT AND RECOVERY

In this section, we propose a method to perform damage assessment and recovery using the new log model. This method scans the log from the point of attack until the end. To provide this direct access an index on each

transaction's start operation could be maintained. While scanning, the mechanism determines the spread of damage and re-executes parts of transactions to repair the database.

4.1 Damage Assessment and Recovery Procedure

When a new affected data item is found, it is added to the list of affected items. However, if the Read_Set of a statement does not contain any bad data, an affected data item is changed to an undamaged item. From that point in the history, the particular data item becomes "fixed". Therefore, it is removed from the list of affected items. Thus the list can "shrink" with the on-going process of recovery. The unit of processing is one log record at a time, and the unit of recovery is one Predicate Block at a time. It checks up on one Predicate Block, determines whether it has accessed any damaged data items, and re-executes that block or part of that block to fix the affected data items. Malicious transactions are never re-executed. While re-executing affected Predicate Blocks, all parts of it may not be executed. There may be one or more statements that did not read or write any damaged data. These statements are not re-executed. When a Predicate Block or part of it is re-executed, some damaged data items are recalculated to new correct values, and they are updated in the damaged item list. When the recovery process is continuing, the data items are not corrected in the stable database. If the database were updated every time an affected item received a new correct value, it would involve a large number of disk access operations. Instead of doing that, the data items and their correct values are simply kept in the list and flushed to the hard disk at the end of the recovery process.

The recovery mechanism declares a Predicate Block affected if it has accessed any damaged data. As mentioned earlier, these affected Predicate Blocks are re-executed for recovery purposes. Based on the source of the bad data, there can be two scenarios.

Scenario 1: The affected data item (one or more) comes *only* from the statement(s) of the Predicate Block. Since only the statement data items are found damaged, it means that the data items in the predicates that were evaluated for the execution of the corresponding statements are all unaffected. Here, the recovery process is relatively simple. Since the data items in the predicates are all "good", there is no need to re-evaluate the predicates. During repair, the same set of statements that executed previously, need to be re-executed. Furthermore, there is no need to undo the data items corresponding to the statements, since the same data items would be updated during re-execution. Based on the semantic content of a statement, there can be two cases. First, the Read_Set of the statement contains one or more damaged data items. Here, first it is checked whether

the data item is already in the affected item list. If it is in there, its value is updated to the newly calculated value. If the data item is not in the list, then it is added to the list with the new value. Secondly, the Read_Set of the statement does not contain any damaged data item. It means that either the statement represents a direct update, or the read data items are all unaffected. Therefore, with the execution of this statement, the affected data item became unaffected. In this case, the statement is not re-executed, and the data item is simply removed from the affected item list, if it is already there.

Scenario 2: The affected data item(s) comes from the predicates or from both statements and predicates of the affected Predicate Block. Since the predicate data item (could be more than one) is affected, during recovery, a completely different Predicate Statement Block may get executed. Therefore, theoretically, the data items that were updated in the first run of this Predicate Block need to be undone. But for all practical purposes, this is not needed at all. The affected item list keeps track of all affected items and their current, correct values. Therefore, during recovery of such a Predicate Block, the values are not updated in the affected item list, since the list already contains good values. If an updated data item is not in the affected item list, it needs to be added in with the before image of the update. For recovery, the Predicate Block is re-executed. Any data item that gets updated by this is first checked if it is in the affected item list. If it is there, its value is updated with the newly calculated value. Otherwise, the data item is added to the list with the newly calculated value.

In both *Scenarios 1* and *2*, for recovery purposes, the current correct value of some of the data items may be required. If the data item is in the affected item list, then its current correct value can be easily obtained from there. But if the data item is not in the list, it means that the item was unaffected at that particular point in history. Its value cannot be obtained from the database because a latter transaction may have updated it. So using the database value would be incorrect. To get its value for that point of time, the log must be searched in the forward direction to find the next update of that data item. The before image (*old_value*) of that update record will reflect the correct value. If such an update is not found, then the stable database is accessed.

4.2 Implementation

The following damage assessment and recovery algorithm uses several data structures to accomplish its tasks. The first such structure is *Malicious Transaction List* (*Malicious_TL*). This list contains the transaction identifiers of all malicious transactions, in the order they committed. The

second structure *Affected Item List* (*Affected_IL*) is basically a collection of data item records that were damaged or affected directly or transitively by the attacking transaction(s). It has two fields, Data_Item and Fresh_Value. The Data_Item field represents the identifier of the data item (e.g., x) and the Fresh_Value field contains the correct value of the data item. The value of the Fresh_Value field for a data item can change as repair continues. The last structure *Affected_Predicate_Block_*T_i (*Affected_PBT*$_i$), contains the identifiers of Predicate Blocks of transaction T_i that have been affected directly or transitively by a malicious transaction. Each entry of this list has two fields, ID and Source. The ID field contains the Identifier of the Predicate Block. The Source field value can be "P", or "S", or "PS" depending on whether a damaged data item is found in the predicate block, or the statement, or both predicate block and statement respectively. The developed algorithm is presented below.

4.3 Damage Assessment and Recovery Algorithm

1. Create *Affected_IL* and initialize it to null, i.e., *Affected_IL* = { }
2. Get the first entry of the *Malicious_TL*, say T_{Mi}
3. Find the log record [Start_Transaction, T_{Mi}] in the log
4. Scan the log until the end. For every committed transaction,
   4.1 If ((Log Record = = [Start_Transaction, T_i]) AND ($T_i \notin$ *Malicious_TL*)) then
   4.1.1 Create *Affected_PBT*$_i$ list and initialize it to null
   4.2 If (Log Record = = [Read_Item, $T_i.P_{j.k}$, x]) then
   4.2.1 If (($T_i \notin$ *Malicious_TL*) AND ($x \in$ *Affected_IL.Data_Item*)) then
   4.2.1.1 If ($T_i.P_j \notin$ *Affected_PBT*$_i$*.ID*) then
   Set *Affected_PBT*$_i$ = *Affected_PBT*$_i \cup [T_i.P_j$, P]
   4.3 Else If (Log Record = = [Read_Item, $T_i.P_{j.k}.S_l$, x]) then
   4.3.1 If (($T_i \notin$ *Malicious_TL*) AND ($x \in$ *Affected_IL.Data_Item*)) then
   4.3.1.1 If ($T_i.P_j \in$ *Affected_PBT*$_i$*.ID*) then
   4.3.1.1.1 If the corresponding item is [$T_i.P_j$, P] then
   Substitute it with [$T_i.P_j$, PS]
   4.3.1.2 If ($T_i.P_j \notin$ *Affected_PBT*$_i$*.ID*) then
   Set *Affected_PBT*$_i$ = *Affected_PBT*$_i \cup [T_i.P_j$, S]
   4.4 Else If (Log Record = = [Write_Item, $T_i.P_{j.k}.S_l$, x, *old_value*, *new_value*]) then
   4.4.1 If ($T_i \in$ *Malicious_TL*) then
   4.4.1.1 If ($x \notin$ *Affected_IL.Data_Item*) then
   Set *Affected_IL* = *Affected_IL* $\cup$ [x, *old_value*]
   4.4.2 If (($T_i \notin$ *Malicious_TL*) AND ($T_i.P_j \notin$ *Affected_PBT*$_i$*.ID*)) then
   4.4.2.1 If ($x \in$ *Affected_IL.Data_Item*) then
   Remove the entry for x from the *Affected_IL*,

4.4.2.2 Else Do nothing
4.4.3 If (($T_i \notin$ *Malicious_TL*) AND ($T_i.P_j \in$ *Affected_PBT*$_i$*.ID*)) then
4.4.3.1 If the Source field of the entry for $T_i.P_j$ in the *Affected_PBT*$_i$ list contains only "S" then
Call Procedure Check_ReExecute($T_i.P_{j.k}.S_l$)
4.4.3.2 Else // Source field of $T_i.P_j$ contains "P" or "PS"
4.4.3.2.1 If ($x \notin$ *Affected_IL.Data_Item*) then
Add an item to the *Affected_IL*
4.4.3.2.2 If the currently scanned update log record is the last update by the Predicate Statement Block *k*, then
Call Procedure ReExecute_PB($T_i.P_j$)
4.5 Else If (Log Record = = [Commit, T_i]) then
4.5.1 If there exists any *Affected_PBT*$_i$ list
4.5.1.1 Delete *Affected_PBT*$_i$ list

5. For each data item in the *Affected_IL*,
   5.1 Bring the data item into the cache from the stable database
   5.2 Update its value to the Fresh_Value of the entry
6. Start and end a checkpoint to flush the updated data items to the stable database
7. Delete *Affected_IL*

Procedure Read_Item (Data_Item : *p*)
{ 1. If ($p \in$ *Affected_IL.Data_Item*) then
   1.1 Read its current correct value from the Fresh_Value field of the corresponding entry
2. Else // ($p \notin$ *Affected_IL.Data_Item*)
   2.1 Scan forward to find the next update record that modified data item *p* and obtain the *old_value* from the log record.
   2.2 If forward scan is unsuccessful, retrieve the value directly from the database }

Procedure Update_AIL (Data_Item : *q*; Data_Value : *new_value*)
{ 1. If ($q \in$ *Affected_IL.Data_Item*) then
   1.1 Substitute the Fresh_Value field of the entry in *Affected_IL* with the newly calculated value (*new_value*)
2. Else // ($q \notin$ *Affected_Item_List.Data_Item*)
   2.1 Add a new entry for *q* in the *Affected_IL* with the newly calculated value }

Procedure Check_ReExecute (Statement Identifier : $T_i.P_{j.k}.S_l$)
{1. Use $T_i.P_{j.k}.S_l$ as a pointer to locate the "Record_Statement" log record for S_l of Predicate Statement Block *k* of Predicate Block *j*
2. Read the log record into a buffer

3. If ((Read_Set(S_l) ∩ *Affected_IL.Data_Item*) == ∅) then
   3.1 For the data item q that is updated by S_l
   3.1.1 If (q ∈ *Affected_IL.Data_Item*) then
   3.1.1.1 Remove the entry q from the *Affected_IL*
4. Else // Read_Set(S_l) contains one or more affected data items
   Re-Execute S_l
   4.1 For every data item P that needs to be read for re-execution,
   4.1.1 Call Procedure Read_Item (p)
   4.2 For the data item Q that gets updated upon re-execution of S_l
   4.2.1 Call Procedure Update_AIL(q, *new_value*), where *new_value* represents the newly calculated value of q }

Procedure ReExecute_PB (Predicate Block Identifier : $T_i.P_j$)

{1. Use $T_i.P_j$ as a pointer and locate the beginning of Predicate Block j in the log.
2. From the log reconstruct the entire string for Predicate Block j in a buffer
3. Re-execute the Predicate Block j
   3.1 For every data item P that needs to be read for re-execution
   3.1.1 Call Procedure Read_Item (p)
   3.2 For every data item q that gets updated due to re-execution,
   3.2.1 Call Procedure Update_AIL(q, *new_value*), where *new_value* represents the newly calculated value of q }

5. CONCLUSION

In case a malicious transaction damages a database and the attack is detected, it is necessary to carry out prompt recovery. All damage assessment and recovery algorithms developed so far require more semantic information about transaction operations than what is available in the log. In this research, a new logging protocol is proposed which records all appropriate information regarding transactions. This information includes various predicates and statements whether they have been executed or not. This helps in accurate damage assessment and fast recovery. We have discussed, in detail, requirements of damage assessment and recovery process and have presented a suitable algorithm. The algorithm performs damage assessment and recovery concurrently. A procedure for recovery from system failures is also described.

ACKNOWLEDGMENT

We are thankful to Dr. Robert L. Herklotz and Capt. Alex Kilpatrick for their support, which made this work possible. This work was partially funded by AFOSR grant F49620-99-1-0235.

REFERENCES

[1] P. Amman, S. Jajodia, C. D. McCollum, and B. Blaustein, Surviving Information Warfare Attacks on Databases, Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 1997.

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

[3] R. Elmasri, and S. B. Navathe, Fundamentals of Database Systems, Third Edition, Addison-Wesley, Menlo Park, CA, 2000.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, The Notions of Consistency and Predicate Locks in a Database System, CACM, 19(11), November 1976.

[5] J. Gray and A. Reuter, Transaction Processing : Concepts and Techniques, Morgan Kauffmann, San Mateo, CA, 1993.

[6] T. Haerder and A. Reuter, Principles of Transaction-Oriented Database Recovery, Computing Surveys, 15(4), December 1983.

[7] S. Jajodia, C. D. McCollum, and P. Amman, Trusted Recovery, Communications of the ACM, 42(7), pp. 71-75, July 1999.

[8] H. F. Korth and A. Silberschatz, Database System Concepts, Second Edition, McGraw-Hill, New York City, NY, 1991.

[9] P. Liu, P. Ammann, and S. Jajodia, Rewriting Histories: Recovering from Malicious Transactions, Distributed and Parallel Databases, 8(1), pp. 7-40, January 2000.

[10] C. Mohan et. al, ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM Transactions on Database Systems, 17(1), March 1982.

[11] M. T. Ozsu and P. Valduriez, Principles of Distributed Database Systems, Second Edition, Prentice-Hall, Upper Saddle River, NJ, 1999.

[12] B. Panda and J. Giordano, Defensive Information Warfare, CACM, July 1999.

[13] B. Panda and J. Giordano, Reconstructing the Database After Electronic Attacks, Database Security XII: Status and Prospects, S. Jajodia (editor), Kluwer Academic Publishers, 1999.

[14] S. Patnaik and B. Panda, Dependency Based Logging for Database Survivability from Hostile Transactions, Proceedings of the 12th International Conference on Computer Applications in Industry and Engineering, November 1999.