Chapter 8

ANALYZING TRANSACTION LOGS FOR EFFECTIVE DAMAGE ASSESSMENT

Prahalad Ragothaman and Brajendra Panda

Abstract In this research, we have proposed to divide the log into several segments based on three different methods with a view to reduce log access time, as a result, expediting recovery. We offer to segment the log based on the number of committed transactions, time and space. A fixed number of transactions will form a segment in the first approach. In the second method, a new segment will be formed with all committed transactions after a set time has elapsed. In the third approach, a segment will be built with all the committed transactions after they have used up a set size of disk space. The three schemes mentioned also vouch for the fact that no segment will grow out of proportion since we are enforcing constraints on their sizes. The algorithms to implement this approach will be relatively simple and easy. Performances of these algorithms have been tested through simulation programs and the results are discussed.

Keywords: Transaction dependency, damage assessment, log segmentation, defensive information warfare

1. Introduction

In this rapidly changing world where everything boils down to time, information sharing plays a vital role. Computers are the most powerful means to share information. With the dawn of Internet technologies, this process has become faster and efficient. But unfortunately, the Internet has also attracted a large number of malicious users who have used it to break into systems and render them inconsistent and unstable. Though there are several protection mechanisms available to stop malicious users from intruding into the system, they are not always successful as savvy hackers find different ways to attack systems. Hence the next best thing would be to detect the attack and bring the system back to a consistent state as soon as possible. Some of the most recent intrusion detection techniques are presented in [4,5,10]. But intrusion

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: 10.1007/978-0-387-35697-6_26

E. Gudes et al. (eds.), *Research Directions in Data and Applications Security*

detection techniques are beyond the scope of this research and hence we shall not discuss them in detail.

Traditional recovery methods [2,3,6] recover databases after a failure. The log is scanned from the end until the last checkpoint and all those active transactions whose effects were saved into the database are undone and those transactions that were committed but whose effects were not saved into the database are redone. Also in the traditional logging mechanism, read operations are not stored and the log is periodically purged to free up disk space. However, for recovery of databases after an attack by a malicious transaction we require that both read and write operations of transactions are stored in the log and the log is never purged. This causes the log to grow to astronomical proportions and scanning such a log will become a very tedious and slow process. This will lead to a prolonged denial-of-service. A solution is to devise methods that read only parts of log and skip as much as possible while guaranteeing that the sections skipped do not contain any affected transactions. This is done by log segmentation. Segmenting the log based on transaction dependency [8] and data dependency [9] achieves this goal. However, these approaches use intricate computation and slow down transaction processing although achieving faster damage assessment and recovery. In this research we intended to strike a balance between log segmentation overhead and time needed for damage assessment and recovery. We offer to segment the log based on the number of committed transactions, time and space. Through simulation we verified the improved performance of our proposed methods.

A major drawback of the approaches presented in [8,9] is that a segment can grow to exorbitant proportions. For example, in case of transaction dependency based log segmentation, if a large number of transactions are dependent on a single transaction, the cluster can become massive and might even become as big as the entire log in the worst case. Therefore, scanning for affected transactions in the clusters will be no different from scanning the entire log, as a result, slowing the process of damage assessment and recovery. A similar analogy can be extended to the data dependency approach.

Another drawback in both these approaches is the amount of system resources used in building the segments. In real time, clustering the log based on transaction dependency and data dependency involves usage of valuable system resources when execution of operations of transactions must be given higher priority. In an ideal scenario, a database is not attacked very frequently. Hence, having simple algorithms to build the segments and designing efficient algorithms to assess damage is necessary.

In this research, we present simpler models to build the log. We keep in mind not to allow the clusters to grow too large by enforcing restrictions on clusters like number of committed transactions, the size of clusters, or a time window for clusters. We have also presented an algorithm that carries out the

process of damage assessment when the log is clustered according to any of the three models that we have developed. We have provided simulation results to show effectiveness of our approaches.

The rest of the paper is organized as follows. The log segmentation models along with their respective algorithms are presented in Section 2. In Section 3, we offer the damage assessment model. Section 4 discusses the analysis of results obtained through simulation. Section 5 concludes the paper.

2. Log Segmentation Model

As stated earlier, we have developed methods of segmenting the log so that the segments are limited from growing out of proportion. A segment is formed after a certain condition such as a fixed number of committed transactions, a specified time window, or the space occupied by the committed transactions is satisfied. Our model is based on the following assumptions: (1) transaction operations are scheduled in accordance with the rigorous two-phase locking protocol as defined in [3], (2) read operations are also recorded in the log file, (3) intrusion is detected using one of the intrusion detection techniques and the id of the attacking transaction is available, (4) the log is never purged, and (5) blind writes are not allowed. Next, we define a few data structures that are used in the algorithms.

Tuft: A tuft is a group of transactions that adheres to any one of the three models presented. The transactions in the tuft are stored in the chronological order of their commit time. A tuft is denoted by Γ_i where i denotes the tuft number.

Read_items: It is a table that consists of two fields: the tuft number and a list of all the data items that were read by all the transactions in that tuft.

Tuft_table: This table lists transaction number and the corresponding tuft in which the transaction's operations are stored.

To implement the algorithms, a temporary log file is maintained. The temporary log file is the system generated log that stores all necessary operations of transactions. The operations of the transactions in the temporary log file are considered to build the tufts. During checkpoint, the following steps take place: no new transactions are considered, all active transactions are completed, the modified database buffers are saved into the stable storage, an end-checkpoint record is added to the last tuft after the last transaction, the temporary log file is deleted, and then execution of new transactions resume. The methods of segmentation are discussed below with the help of examples. Algorithms to implement these approaches are also provided.

2.1 Log Segmentation Based on Number of Committed Transactions

In this approach, a fixed number of committed transactions are grouped together to form a tuft. Consider the following piece of history, H, in the temporary log file. T_i represents the start of transaction i and c_i represents the commit record of transaction i. Individual operations of the transactions are not shown for simplicity.

H: $T_1\ T_5\ c_1\ T_7\ c_7\ T_9\ c_5\ T_{10}\ T_{12}\ c_{12}\ T_{15}\ T_2\ c_{10}\ T_{13}\ c_9\ T_6\ T_8\ c_{15}\ c_2$
$T_4\ c_8\ c_4\ T_{18}\ c_{18}\ c_6\ c_{13}$.

Let us assume that five committed transactions form a tuft. Thus, the tufts are:

$\Gamma_1 = \{T_1, T_7, T_5, T_{12}, T_{10}\}$
$\Gamma_2 = \{T_9, T_{15}, T_2, T_8, T_4\}$
$\Gamma_3 = \{T_{18}, T_6, T_{13}\}$

It has to be observed that the transactions are considered in the order of their commit sequence. Doing so ensures that the partial order among the transactions is maintained in the tufts. A lemma to prove this is provided later. The variables that are used to implement the algorithm for this approach are given below.

Transactions_in_tuft is a variable that holds the number of transactions that make a tuft. *Transaction_count* is a variable that holds the number of transactions that are recorded into the tuft so far.

Algorithm 1:

1. Initialize $i = 1$;
2. Create tuft Γ_i;
   Add a record for (Γ_i) in the *read_items* table;
   transaction_count := 1;
3. if(*transaction_count* != *transactions_in_tuft*);
   3.1. Read next committed transaction, T_j,
   from the temporary log file and store the
   operations of T_j in (Γ_i).
   Add a new record in the *tuft_table* for T_j;
   3.2. Add the read set of T_j in the *read_items*
   table against tuft (Γ_i);
   3.3. Increment *transaction_count*;
   3.4. Goto Step 3;
4. Else
   4.1. Increment i;
   4.2. Goto Step 2;

This process of segmenting the log is much simpler than the log segmentation algorithms based on either transaction dependency or data dependency approach. Also, we do not run the risk of segments growing too large. We can rest assured that a segment, a tuft in this case, will definitely end once a fixed number of transactions commit. If the algorithm, at Step 3.1, does not find any committed transactions to store into the tuft, it will wait until a transaction commits and then proceed with the rest of the steps in the algorithm.

2.2 Log Segmentation Based on a Time Window

For this protocol, we assume that the commit time of each transaction is also stored along with the commit record of the transaction. Using this method, all transactions that committed in a particular window of time will form a tuft. To explain the idea let us consider a piece of log from the temporary log file. Commit time (assume in AM) of each transaction shown next to the commit record of that transaction.

H: T_1 T_5 c_1[9:05] T_7 c_7[9:07] T_9 c_5[9:12] T_{10} T_{12} c_{12}[9:15] T_{15} T_2 c_{10}[9:16] T_{13} c_9[9:16] T_6 T_8 c_{15}[9:23] c_2[9:25] T_4 c_8[9:40] c_4[9:42] T_{18} c_{18}[9:43] c_6[9:43] c_{13}[10:10].

The operations of each individual transaction and aborted transactions are not shown for simplicity. Let us assume that the time allotted to build a tuft is five minutes. Let us also assume that this algorithm started at 9:00 AM. Thus all transactions that committed between 9:00 AM and 9:05 AM, both times inclusive, must form a tuft. The next window of time will begin at 9:06 AM and end at 9:10 AM and so on. If no transaction committed in a particular window of time, that tuft will be used for the committed transactions that occur in the next window of time. The tufts and their transactions for the history given above are shown below.

$(\Gamma_1) = \{T_1\}$
$(\Gamma_2) = \{T_7\}$
$(\Gamma_3) = \{T_5, T_{12}\}$
$(\Gamma_4) = \{T_{10}, T_9\}$
$(\Gamma_5) = \{T_{15}, T_2\}$
$(\Gamma_6) = \{T_8, T_4, T_{18}, T_6\}$
$(\Gamma_7) = \{T_{13}\}$

As in the previous model, here too we do not have to worry about a tuft growing too big. The tuft will certainly end when the time slice ends. The number of transactions in a tuft is not known unlike the previous method. Nor is the size of each tuft. A regulation on the size of the tuft can be enforced and this method is discussed in the next section.

As mentioned before, we assume that tuft Γ_1 will be created at 9:00 AM. Hence Γ_3 will be created at 9:11 AM. With respect to tuft Γ_3 from the above

example, we will define a few necessary data structures that are required in the algorithm to implement this approach.

Tuft_end_time is the variable that holds the time at which the current tuft must end. The variable *Time_of_last_creation* holds the time when the most recent tuft was built. *Current_time* stores the time at that moment. It has to be noted that *Current_time* will be called only once and the same time will be used through one iteration of the algorithm. *Period_of_creation* holds the time period after which the next tuft must be built. *Commit_time*(T_i) is the time at which transaction T_i committed.

When the procedure starts, *tuft_end_time* is initialized to *current_time* plus the *period_of_creation*. *Time_of_last_creation* is initialized to the *current_time*. All those transactions that have their commit time between the *time_of_last_creation* and *tuft_end_time* will form a tuft. When the tuft ends, the window is advanced to the next time slice by adding the *period_of_creation* to the *tuft_end_time* and setting the *time_of_last_creation* to the *tuft_end_time*.

Algorithm 2:

1. Initialize $i = 1$;
   tuft_end_time = *current_time* + *period_of_creation*;
   time_of_last_creation = *current_time*;
2. Create tuft Γ_i;
   add a new record for Γ_i in the *read_items* table;
3. WAIT UNTIL (*current_time* >= *tuft_end_time*)
   3.1. Read all transactions, T_j, from temporary log where
   (*commit_time*(T_j) > *time_of_last_creation*) and
   (*commit_time*(T_j) <= *tuft_end_time*) and
   record their operations into tuft Γ_i.
   Add a new record for T_j in the tuft table;
   3.2. Add the read set of T_j to the *read_items* table against tuft Γ_i;
   3.3. Increment i;
   3.4. *time_of_last_creation* = *tuft_end_time*;
   3.5. *tuft_end_time* = *tuft_end_time* + *period_of_creation*;
   Go to Step 2;

2.3 Log Segmentation Based on Fixed Size Tuft

In this approach, the size of the tuft is kept constant. Operations of committed transactions are added to the tuft until there is no more space for the next committed transaction to fit into it. It is assumed that the size of a tuft is bigger than the largest transaction. If a committed transaction does not fit into the current tuft, we close the tuft and create a new one. We do not allow the transactions to span from one tuft to another. This will result in wastage of disk

space in this approach. But if we do allow the transactions to span, damage assessment will be more complicated. This will be evident when we present the damage assessment model in Section 3. Key variables and data structures used in the algorithm are shown below.

Sizeof(i) returns the size of i. *Tuft_size* is the size of the tuft. Var is a variable that has units as bytes. T_{lar} is the largest transaction in terms of size. *Space_available(i)* contains the space available in i.

Algorithm 3:

1. $i = 1$;
   tuft_size = **sizeof**(T_{lar}) + *var*;
2. Create tuft Γ_i;
   space_available(Γ_i) = *tuft_size*;
   Add a new record for Γ_i in the *read_items* table;
3. Read next committed transaction, *Tk*, from temporary log file;
4. if **sizeof**(*Tk*) <= *space_available*(Γ_i);
   4.1. Store the operations of T_k in Γ_i;
   4.2. Add the read set of T_k to the *read_items* table against tuft Γ_i;
   4.3. Add a record for the transaction T_k in the *tuft_table*;
   4.4. *space_available*(Γ_i) = *space_available*(Γ_i) − **sizeof**(T_k);
   4.5 Go to Step 3;
5. Else
   5.1 Increment *i*;
   5.2 Create tuft Γ_i;
   space_available(Γ_i) = *tuft_size*;
   Add a new record for Γ_i in the *read_items* table ;
   5.3 Go to Step 3.

3. Damage Assessment Model

Damage spreads in a database when valid transactions directly or indirectly read data items written maliciously by attacking transactions and then update other data items. Damage assessment is the process of identifying all those data items written by transactions that read damaged data written either by malicious transactions or other affected transactions. This process is often executed by denying service of the system to other valid transactions. Other valid transaction must not be kept waiting for long while damage assessment is done. Thus, this process has to be done in as efficient a manner as possible and open the system for other transactions. The damage assessment model presented in this section holds good for segmented logs developed by any one of the three methods proposed earlier. It is assumed that the intrusion is detected and all the attacking transactions are known before the start of damage assessment. When

an attack is detected, the following steps take place: (1) no new transactions are accepted (2) all active transactions are completed, any updates made by these transactions are saved to the database and the tufts updated accordingly, and (3) the temporary log file is deleted. Thus the temporary log file can be completely ignored during damage assessment. Some of the variables and data structures used in this algorithm are given below.

affected_items: It is a list containing data items that were written either by a malicious transaction or by a transaction that read a data item written by a malicious transaction.

affected_transactions: This list contains all malicious transactions and those transactions that read malicious data items.

The process of damage assessment starts by identifying all malicious transactions. The tuft in which the first malicious transaction, say T_m, is stored is obtained by looking up in the tuft_table. All transactions that appear in all of the tufts prior to the one in which T_m appears can be safely ignored because they are not affected. No valid transaction that read a data item written by Tm and appearing in one of the tufts prior to the one in which T_m is present will be affected because it would have committed earlier than T_m. The data items that were written by Tm are stored into the *affected_items* list. The same procedure is carried out for any malicious transaction that we encounter further in the process of damage assessment.

Each of the transactions that appear in the same tuft as transaction T_m is scanned to check if it is affected. The read set of the transactions is intersected with the *affected_items*. If the result is not a null set, then that transaction is affected because it has read a data item written by a malicious transaction. The transaction number is added to the *affected_transactions* list and its write set is appended to the *affected_items* list.

After all the transactions in the same tuft are scanned, the subsequent tufts are scanned to look for malicious and affected transactions. If a malicious transaction is present in the subsequent tuft, the write set of the malicious transaction is added into the *affected_items* list. The *affected_items* list is intersected with the *read_items* list of the corresponding tuft. If the result is not a null set, it means that the tuft is affected and one or more of the transactions in that tuft has read data item(s) written by a malicious transaction. All the transactions in that tuft are scanned to determine which of the transactions have read an affected data item. This is done by intersecting the *affected_items* list and the read set of each of the transactions in that tuft. If the result is not a null set, it means that the transaction is affected. The transaction's write set is appended to the *affected_items* list and the transaction number is added to the *affected_transactions* list. This process is done until the process is through with all the tufts.

Algorithm 4:

1. Let the first attacking transaction be T_i;
2. Set *affected_items* = {};
3. Using the tuft table, determine the tuft number, say k, where transaction T_i is stored;
   - 3.1. Append the write set of transaction T_i into the *affected_items* list;
   - 3.2. For each transaction (say T_j) appearing after T_i in Γ_k;
     - 3.2.1. If T_j is a malicious transaction then append the write set of transaction T_j to the *affected_items* list;
     - 3.2.2. Else;
       - 3.2.2.1. Call procedure **assess_damage**(T_j);
   - 3.3. For each tuft l where $l >= k + 1$;
     - 3.3.1. if(*read_items*(l) $\bigcap$ *affected_items* != ϕ);
       - 3.3.1.1. Call procedure **assess_damage**(T_j) on every transaction, say T_j, from the beginning of the tuft;
     - 3.3.2. For each malicious transaction appearing in Γ_l;
       - 3.3.2.1. Append the write set of the next malicious transaction, say T_m, to the *affected_items* list;
       - 3.3.2.2. For each transaction, say T_n, appearing after T_m in tuft Γ_l ;
       - 3.3.2.3. Call procedure assess_damage(T_n);
     - 3.3.3. Increment l;

Procedure **assess_damage**(T_j)

1. Obtain the read set and the write set of the transaction T_j ;
   - 1.2. If T_j is not a malicious transaction;
   - 1.2.1. If(*read_set* (T_j) $\bigcap$ *affected_items* != ϕ);
     - 1.2.1.1. Enter the *write_set* of the transaction T_j into the *affected_items* list;
     - 1.2.1.2. Enter the transaction number in the *affected_transactions* list;
   - 1.3. Else;
     Append the write set of transaction T_j to the *affected_items* list;

In case of multiple attacks, we assume that all malicious transactions have been detected through one of the intrusion detection techniques. These malicious transactions may be present in many tufts. While scanning the tufts for dependencies (Step 3.3.2 of Algorithm 4), we also determine if there are any malicious transactions present in that tuft using the tuft_table. If there are malicious transactions present, we perform the same steps as we did with the

first malicious transaction. While doing so, there are chances that some of the good transactions have read data items written by multiple malicious transactions and those good transactions may have already been appended to the affected_transactions list. We check to see if they are already present in the list and add them only if they are not present.

4. Performance Analysis using Simulation

We simulated the scenario using the C programming language. A sample log was generated in accordance to the rigorous two-phase locking protocol. Inter-leaving of transactions was allowed. Segmentation of the log based on time requires that the time of commit of the transaction be stored too. Programs were written to segment the log based on each of the three methods described before. A program to assess damage assuming an attacker id was also written and affected transactions were obtained. The results of the simulation are shown in the following graphs and the explanation accompanies them.

As depicted in Figure 1, the space accessed by the damage assessment process increases as the number of committed transactions in the tuft increases. Table 1 shows the values of the various parameters that were considered to generate the graph. As more transactions are stored in one tuft, we have to scan the entire tuft if it is affected. Hence we see a decline in performance as the number of transactions in a tuft increases. Nevertheless, this is still better than the performance of the traditional damage assessment process with a non-segmented log as we shall see later.

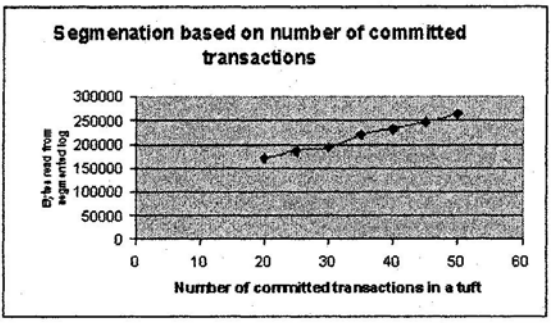

Figure 1. Behaviour of algorithm on a log segmented by number of committed transactions.

Figure 2 shows the performance of the damage assessment process on a log segmented based on the size of the tufts. As before, the performance declines as the size of the tufts increases. When the tuft size increases, more transactions can fit into the tuft and we have to scan one large tuft if we find that tuft affected. The values to construct the chart in Figure 2 are the same as the ones

Table 1. Values of parameters for chart shown in Figure 1.

Total number of Transactions	500
Total number of data items	5000
Maximum data items accessed by a transaction	40
Attacker id	250
Number of committed transactions in a tuft varying with increments of 5	20-50

given in Table 1 except that the size of the tuft varies from 20000 to 50000 bytes with increments of 10000 bytes.

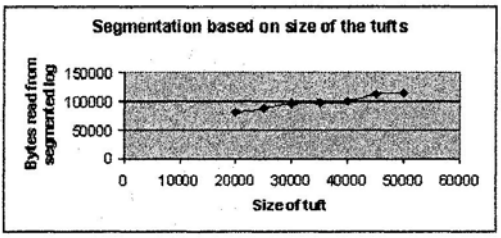

Figure 2. Behaviour of algorithm on a log segmented based on size of tufts.

Figure 3 illustrates performance of time window based segmented log. It has to be observed that the initial part of the curve fluctuates randomly and not so much later on. As the time window for each tuft increases, the space that has to be read during damage assessment also varies uniformly. The commit time of transactions entirely depends on the system resources. For the same seed, the commit time will vary rapidly though the commit sequence remains the same. It is possible that the attacking transactions, and thus the affected transactions, may appear in any of the tufts in the segmented log and not necessarily in the same tuft every time, given the same seed. In consequence, the result varies randomly when the window is smaller. But as the time window increases, things even out and the chances of a transaction occurring in the same tuft also increases. Hence, we see the uniform variation with the increase in the tuft window size. The values to construct the graph shown in Figure 3 is the same as the ones shown in Table 1 except that the time window for each tuft varies from 20000 to 500000 microseconds with increments of 5000 microseconds.

Comparison analysis of damage assessment using unsegmented log and that using a segmented log is shown in Figure 4, which confirms that having a segmented log greatly improves performance. In that, significantly less number of bytes are read during damage assessment. The values to construct the graph shown in the above figure are the same as in Table 1 except that the attacker id was varied from 50 to 450 with increments of 100. The values of number of committed transactions, time window and space occupied by the committed

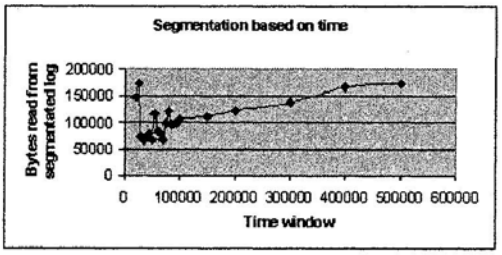

Figure 3. Behaviour of algorithm on a log segmented based on time window.

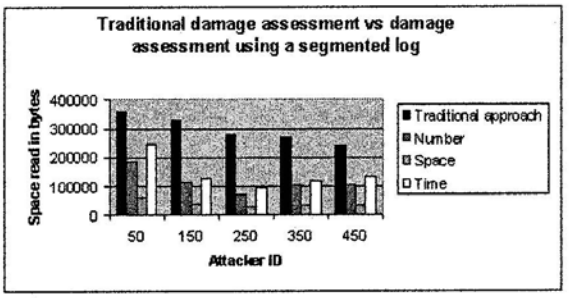

Figure 4. Comparison of traditional approach and approach using a segmented log.

transactions were the same as the ones that were considered to generate the previous graphs.

5. Conclusions

In this paper, we focused on how to segment the log for faster damage assessment in case of an information attack. In the process, we also ensured that the size of each segment was under control and did not grow to humongous proportions. We achieved this by enforcing constraints on the size of the segment such as number of committed transactions, space occupied by committed transactions and a fixed time window for transactions to commit and form a segment. We have provided necessary damage assessment algorithm, which can be used on the log file that was segmented using any of the three approaches described. The result of the damage assessment process is a list of all the malicious and affected transactions. Using this information, we can use any of the previously proposed approaches to carry out the recovery process. We also provided the results of the simulation that was performed to verify efficiency of our methods. The results conclusively proved that damage assessment using a segmented log reads far less disk space than when damage assessment is done with an unsegmented log. We also provided results of damage assessment for varying parameters for log creation. It can be concluded

that damage assessment based on the space occupied by committed transactions is quicker than the other two approaches. It does not read as many bytes of the log file as damage assessment based on number of committed transactions nor does it display the random behavior of the segmented log based on time.

Acknowledgment

We are thankful to Dr. Robert L. Herklotz for his support which made this work possible. This work was partially funded by US 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*, 1997.

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

[3] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems (Third Edition)*, Addison-Wesley, 2000.

[4] S. Forrest, S. Hofmeyr and A. Somayaji, Computer immunology, *Communications of the ACM*, vol. 40(10), pp. 88-96, 1997.

[5] S. Forrest, S. Hofmeyr, A. Somayaji and T. Longstaff, A sense of self for UNIX processes, *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[6] H. F. Korth, A. Silberschatz and S. Sudarshan, *Database System Concepts (Third Edition)*, McGraw-Hill International, 1997.

[7] B. Panda and J. Giordano, Reconstructing the database after electronic attacks, in *Database Security, XII: Status and Prospects*, S. Jajodia (ed.), Kluwer, 1999.

[8] B. Panda and S. Patnaik, A recovery model for defensive information warfare, *Proceedings of the Ninth International Conference on the Management of Data*, pp. 359-368, 1998.

[9] B. Panda and S. Tripathy, Data dependency logging for defensive information warfare, *Proceedings of the ACM Symposium on Applied Computing*, pp. 361-365, 2000.

[10] S. Stolfo et al., JAM: Java agents for meta-learning over distributed databases, *Proceedings of the AAAI Workshop on AI Methods in Fraud and Risk Management*, 1997.