Chapter 7

MULTIPHASE DAMAGE CONFINEMENT SYSTEM FOR DATABASES

Peng Liu and Ying Wang

Abstract Damage confinement is a critical aspect of database survivability. Damaged data items of a database should not be allowed to access until they are repaired. Traditional database damage confinement is *one phase*, that is, a damaged data item is confined only after it is identified as corrupted, and one- phase damage confinement has a serious problem, that is, during damage assessment serious damage spreading can be caused. In this paper, we present the design and implementation of a *multiphase* database damage confinement system, called *DDCS*. The damage confinement process of DDCS has one confining phase, which instantly confines the damage that might have been caused by the intrusion(s) as soon as the intrusion(s) are detected, and one or more later on unconfining phases to unconfine the data items that are mistakenly confined during the confining phase and the items that are repaired. In this way, DDCS ensures no damage spreading during damage assessment. DDCS can confine the damage caused by multiple malicious transactions in a concurrent manner. DDCS is built on top of a commercial database server. DDCS is transparent to end users, and the performance penalty of DDCS is reasonable.

Keywords: Database security, survivability, damage containment

1. Introduction

Recently, more and more people realized that existing secure systems are still vulnerable to a variety of attacks. The inability of existing security mechanisms to prevent every attack is well embodied in several recent large scale Internet attacks such as the DDoS attack in February 2000. These accidents convince the security community that traditional *prevention- centric* security is not enough and the need for *intrusion tolerant* or *survivable* systems is urgent. Intrusion tolerant systems, with characteristics quite different from traditional secure systems, extend traditional secure systems to *survive* or *operate through*

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*

attacks. The focus of intrusion tolerant systems is the ability to continue delivering essential services in face of attacks [1, 4].

Being a critical component of almost every mission critical information system, database products are today a multi-billion dollar industry. Database survivability focuses on the ability to correctly execute transactions (or queries) in face of *data corruption* by attacks. Unfortunately, several studies show that traditional database security mechanisms are very limited in surviving attacks.

Since database attacks can be enforced at multiple levels, including at the hardware, OS, DBMS, and transaction (or application) levels, database survivability in general requires a multi-layer approach. Although several effective low-level survivability mechanisms have been developed to tackle hardware-level attacks [10] and OS-level attacks [2, 8, 9], there are still a lot of challenges to survive malicious transactions, which should be the major security threats to databases according to the fact that most attacks are from insider [3]. In this paper, we focus on how to survive malicious transactions.

A simple transaction level database survivability framework may consist of an *intrusion detector* [5], which identifies malicious transactions, and a *repair manager* [7], which locates and repairs the damage caused by these malicious transactions. We assume that the database continues executing new transactions during intrusion detection and repair. The key challenge of this simple framework is in fact *damage spreading*. In particular, in a database the results of one transaction can affect the execution of some other transactions. When a transaction T_i reads a data object x updated by another transaction T_j, T_i is directly *affected* by T_j. If a third transaction T_k is affected by T_i, but not directly affected by T_j, T_k is indirectly affected by T_j. It is easy to see that when a (relatively old) transaction B_i that updates x is identified malicious, the damage on x has already spread to every object updated by a good transaction that is affected by B_i, directly or indirectly. Moreover, during the repair process of the damage caused by B_i, the damage could further spread to many other data objects.

The main goal of *damage confinement* is to reduce the amount of damage spreading during the repair process. At the first glance, it seems that confining the damage that is already located by the Repair Manager is a good idea. However, in this *one-phase* confinement approach the damage caused on an object will not be confined until the object is identified (by the Repair Manager) as corrupted. And when there is a significant latency for locating a damaged object x, during the latency many new transactions may read x and spread the damage on x to the objects updated by them. As a result, when x is confined many other objects may have already been damaged, and the situation can feed upon itself and become worse because as the damage spreads the damage assessment latency could become even longer.

To overcome the limitation of one-phase damage confinement, we propose a multiphase damage confinement approach. In particular, a multi- phase damage confinement process has one confining phase, which instantly confines the damage that might have been caused by the intrusion(s) as soon as the intrusion(s) are detected, and one or more later on unconfining phases to unconfine the data items that are mistakenly confined during the confining phase and the items that are repaired. In this way, since confined data objects are not readable until they are repaired, multi-phase confinement ensures no damage spreading during the repair process.

In this paper, we present a complete model for multiphase database damage confinement, and we design and implement a multiphase database damage confinement system, called DDCS, that enforces this model. A preliminary version of our model appeared in [6]. In this paper, we extend the model with the ability to handle multiple malicious transactions and we present a query-rewriting technique to transparently implement the extended damage confinement model on top of a commercial DBMS. We also present the detailed design of DDCS and its components. The key features of DDCS are: (1) DDCS enforces multiphase damage confinement, so DDCS can guarantee that there is no damage leakage during the repair process. (2) DDCS can concurrently confine the damage caused by multiple malicious transactions without any damage leakage. (3) DDCS is built on top of a commercial DBMS, so DDCS can be easily transported from one type of DBMS to another type of DBMS. (4) DDCS is transparent to end- users, so database application developers are immunized from the complexity of damage confinement.

2. Multiphase Damage Confinement Elements

For clarity, in this section we only consider how to confine the damage caused by a single malicious transaction B. In next section we will extend the basic model to handle multiple malicious transactions. In our model, a database is a set of data *objects* (or items) which are accessed by transactions. A *transaction* is a partial order of read and write operations that either commits or aborts. A *history* models the concurrent execution of a set of transactions. We assume that there is a transaction *log* that not only keeps the read and write operations of every transaction in the history but also keeps the commit order of these transactions. A *malicious* transaction *corrupts* (or damages) a data object by changing its value to a wrong or misleading value. A corrupted data object is *repaired* after the value of the object is restored to its latest undamaged version. We assume the survivable database system has an Intrusion Detector and that can identify malicious transactions, and a Repair Manager that can locate and repair the set of data objects corrupted. The Repair Manager is triggered by the alarms raised by the Intrusion Detector. Note that since

the detector is not 100% accurate, some damage may not be able to be located and repaired and some undamaged data objects may be mistakenly repaired. Finally, we assume the database continues executing new transactions during intrusion detection and repair.

Definition 1 [One-phase Damage Confinement] The survivable database system enforces *one-phase damage confinement* if all and only the data objects that are identified by the Repair Manager as corrupted will be confined until they are repaired. A confined data object cannot be read or updated by any new transactions.

Definition 2 [Multiphase Damage Confinement] The survivable database system enforces *multiphase damage confinement* if
(1) As soon as a malicious transaction B is detected, a specific set of data objects, denoted S_E, will be instantly confined. S_E is determined in such a way that the set of data objects that are corrupted by B, denoted S_D, is a subset of S_E. This phase is called *initial confinement*. Initial confinement should be quickly done. The set of confined data objects is called a *confinement set*. To ensure no damage spreading after initial confinement, every active transaction should be rolled back before S_E is confined.
(2) The whole multiphase damage confinement process is a sequence of confinement sets, namely, $S_E, S_1, S_2, ..., S_k, ...$, that is converged to the empty set $\emptyset$. S_E is the result of initial confinement and may include a lot of undamaged data objects that are mistakenly confined. S_i (with $i \geq 1$) is the result of a set of unconfining operations that only unconfine the data objects that are mistakenly confined or the data objects that are repaired. As a result, $S_j \subseteq S_i$ for $i < j$. When the sequence is converged to $\emptyset$, all the confined damage is repaired and no object needs to be confined. The unconfining operations are usually grouped into several *unconfining phases*, although these unconfining phases can be concurrent.

The advantage of multiphase containment is no damage leakage during repair and much simpler repair. The drawback is that some undamaged objects could be mistakenly contained temporarily. A multiphase database damage confinement system that enforces this model can work as follows:
(a) The initial confinement can be enforced using *time stamps*. Assume each data object in the database is associated with a specific time stamp that indicates when the object is last updated. As soon as a malicious transaction B is detected, all the objects associated with a time stamp later than the start time of B will be confined as S_E. Since only these objects could be damaged by B, S_D is a subset of S_E. Note that S_E does not include the objects that are updated after initial confinement.
(b) The system can have the following four unconfining phases, which start at the same time but usually proceed with very different speeds.

Phase A: During the process of damage assessment which scans the log (after B commits) to find out which good transactions are affected by B and which data objects are corrupted by B and these affected transactions, the Repair Manager also wants to find out which objects are not damaged and should be unconfined. In particular, when the Repair Manager finds that a transaction G is not affected, all the objects that are updated by G and are associated with a time stamp earlier than the commit time of G will be unconfined. Note that in many cases not all objects updated by G can be unconfined because some of these objects could have been later on damaged.

Phase B exploits the dependency relationship among transaction types or access patterns. In particular, we assume that each transaction belongs to a specific *type* and the profile or code for each transaction is pre-known. We found that the types of data objects that a transaction may read and write could be extracted from the transaction's profile. And we call the read (write) set extracted from a profile the *read (write) set template* of the transaction type. (Note that a data object type can indicate a column or a table in a relational database. A transaction type ty_i is *dependent upon* ty_j if the intersection of ty_j's write set template and ty_i's read set template is not empty. The rational is that in the corresponding type history of the (affected) history, if $(type(T_j), type(B))$ is not in the transitive closure of the dependent upon relation, then T_j is not affected by B, and T_j's write set template should be unconfined. However, since some objects in T_j's write set template could be later on damaged, we cannot unconfine T_j's write set as soon as we find that T_j is not affected. Instead, Phase B puts T_j's write set into a temporary set denoted Q. Q will not be unconfined until every transaction that could be affected by B is analyzed. During this analysis, when we find that a transaction T_k's type is affected by $type(B)$, we will remove the intersection of Q and T_k's write set template out of Q.

Phase C: One limitation of Phase B is that in some cases even if $(type(T_k), type(B))$ is in the transitive closure of the dependent upon relation, T_k could still be unaffected, since the granularity of an object (type) kept in a template is usually very large. To unconfine the writes of such T_k, Phase C *materializes* the read and write set templates of each transaction in the affected history and uses the materialized read and write sets to identify the transactions that are not affected and the objects that should be unconfined. A data object type (kept in a template) is materialized by replacing a variable associated with the object type with an input of a transaction instance. A materialized read (write) set looks no different from a real read (write) set. Phase C does unconfinement in almost the same way as Phase A except that Phase C uses materialized read and write sets.

Phase D: After a damaged object x is repaired by the Repair Manager, x will be unconfined.

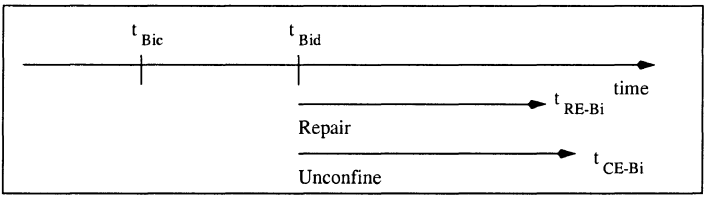

Figure 1. Confinement timeline of B_i.

(c) It is easy to see that Phase A is accurate, that is, every object that is mistakenly confined will be unconfined. However, Phase A is in general very slow. During the damage assessment latency, substantial availability could be lost. Phases B and C can provide a lot more availability, although they may miss some objects that should be unconfined. That is, Phases B and C are in general much quicker than Phase A since they do not need to analyze the read and write operations of transactions.
(d) Finally, it should be noticed that periodically after a repair is done, the time stamp based confinement control should be dismissed.

3. Handling Multiple Malicious Transactions

In many cases, multiple malicious transactions can be detected at different points of time with some intervals between each other. So it is possible that a new malicious transaction is detected before the repair of the set of already detected malicious transactions is done. Under such situations, we found that some data objects that are already unconfined could have been corrupted by the new malicious transaction and should not be unconfined any longer. To illustrate, let's consider a simple example where when malicious transaction B_j is detected B_i is still under repair. If B_j were not detected, the damage confinement and repair process of B_i can be shown by Figure 1. Assume t_{Bic} is the commit time of B_i; t_{Bid} is the time when B_i is detected and known to every component of the survivable database system; t_{RE-Bi} is the time when all the damage caused by B_i is repaired; and t_{CE-Bi} is the time when all the unconfining phases for B_i end. Assume B_j is detected after t_{Bid} but before t_{CE-Bi} and assume that t_{Bic} is before t_{Bjc} , then it is possible that an unconfining phase for B_i has already unconfined a data object x which is updated after t_{Bjc} and not damaged by B_i. However, if x is actually damaged by B_j, then after the initial confinement of B_j is done x should no longer be unconfined. In this section, we propose a multiphase damage confinement algorithm for multiple malicious transactions. Our algorithm can guarantee that on damage will leak out of any confined part of the database.

Algorithm 1 Handling multiple malicious transactions
We assume at one point of time the confined part of the database is specified by a confinement time window denoted $[t_1, t_2]$, and an unconfinement set denoted U_SET. Any data object updated within the confinement time window should be confined except that the object is in U_SET.

// We specify the algorithm by induction

When the system has only one malicious transaction B_i being repaired and a new malicious transaction B_j is detected:

// Note that at this moment, t_1 is the start time of B_i, t_2 is the time when the initial confinement
// for B_i is enforced, and U_SET contains the data objects that are already unconfined by the
// unconfining operations for B_i

Confinement operations:
(a) roll back all the active transactions;
(b) set the value of t_2 to the current time;
(c) set the value of t_1 to $min(t_{Bis}, t_{Bjs})$. Here t_{Bis} and t_{Bjs} are the start times of B_i and B_j, respectively; //Note that B_j could start before B_i
(d) allow new transactions to come in after U_SET is adjusted by the following unconfinement operations;
// This adjustment includes any changes to U_SET

Unconfinement operations:
Case 1 B_j commits before B_i
(a) remove every data object from U_SET;
(b) shut down all the current unconfining phases;
(c) restart unconfining phases A, B, and C by scanning the log from the point where B_j starts. The restarted phases should now handle B_j, B_i instead of only B_i. For example, Phase A should put only the objects that are neither corrupted by B_i nor corrupted by B_j into U_SET;
(d) restart the repair process (and unconfining phase D) by scanning the log from the point where B_j starts. The restarted repair process and Phase D should now handle B_j, B_i;
(e) whenever a data object is unconfined, put the data object into U_SET;
Case 2 B_j commits after B_i
if no unconfining phase has scanned the part of the log that contains the operations that were performed after B_j commit

continue each unconfining phase in such a way that each confining phase is adjusted
to handle B_j, B_i instead of only B_i;

else for each unconfining phase (including the repair process) that has scanned some of the operations that were performed after B_j commit

(a) shut down the unconfining phase (or the repair process);
(b) remove every data object that was updated after B_j commit but is unconfined by this phase (or process) from U_SET;
(c) restart this unconfining phase (or the repair process) by re-scanning the log from the point where B_j starts. The restarted phase (or repair process) should now handle B_j, B_i.

When the system has m malicious transactions $\{B_{i1}, B_{i2}, ..., B_{im}\}$ being repaired concurrently and a new malicious transaction B_j is detected:
[Confinement operations] The difference from the above is:
(d1) set the value of t_1 to $min(t_1', t_{Bjs})$. Here t_1' indicates the current value of t_1 before B_j is detected;
[Unconfinement operations] The difference from the above is:
Case 1 $t_{Bjc} < t_1'$
(d1) restart unconfining phases A, B, and C by scanning the log from the point where B_j starts. The restarted phases should now handle $\{B_{i1}, B_{i2}, ..., B_{im}, B_i\}$;
(d2) restart the repair process (and unconfining phase D) by scanning the log from the point where B_j starts. The restarted repair process and Phase D should now handle $\{B_{i1}, B_{i2}, ..., B_{im}, B_i\}$;
(d3) whenever a data object is unconfined, put the data object into U_SET;
Case 2 $t_{Bjc} > t_1'$
if no unconfining phase has scanned the part of the log that contains the operations that were performed after B_j commit

(d1) continue each unconfining phase in such a way that each confining phase is adjusted
to handle $\{B_{i1}, B_{i2}, ..., B_{im}, B_i\}$;

else for each unconfining phase that has scanned some of the operations performed after B_j commit

(d1) restart this unconfining phase (or the repair process) by re-scanning the log from the point where B_j starts. The restarted phase should now handle $\{B_{i1}, B_{i2}, ..., B_{im}, B_i\}$;

THEOREM 1 In a survivable database system where multiple malicious transactions can be repaired concurrently, Algorithm 1 ensures that: All the damage that is caused by a malicious transaction will be confined as soon as the malicious transaction is detected, and At any point of time, no damage will spread out of the confined part of the database.

4. DDCS

DDCS is a prototype system that implements the multiphase database damage confinement algorithm presented in Section 3. DDCS is built on top of an Oracle DBMS but DDCS is in general not dependent on the specific DBMS. In order to support time-stamp based confinement control, DDCS transparently adds an extra time-stamp column to each table and maintains time stamp information by rewriting SQL statements. The major components of DDCS are shown in Figure 2. In general, the Intrusion Detector informs DDCS which transactions are malicious. The Transaction Proxy proxies user transactions for the purpose of keeping track of the status and the SQL statements of transactions (in the TRANS_LIST table). The triggers and the Read Extractor are responsible for keeping track of the read and write operations of transactions, which are necessary for the unconfining operations. Note that the Read Extractor extracts transaction read information from the SQL statements kept by the Transaction Proxy. The Confinement Executor is responsible for (1) maintaining the confinement time window as new malicious transactions are reported by the Intrusion Detector, (2) enforcing the damage confinement control with the help of the U_SET, and (3) maintaining the time stamp information by rewriting user SQL queries. Unconfining phases B and C are enforced by the Unconfinement Executor. Unconfining phases A and D are enforced by the Repair Manager, which also performs damage assessment and repair. In [7], the Intrusion Detector, the Triggers, the Read Extractor, the Transaction Proxy, and the Repair Manager are described in detail. In this section, we will focus on the Confinement Executor and the Unconfinement Executor.

The key operations of DDCS are triggered by three main events. (1) When a new user transaction T arrives, the Transaction Proxy will proxy the transaction, and the Unconfinement Executor will enforce the confinement control and maintain the time stamps for the data objects that are updated by T. At the same time, the read operations of T will be extracted from T' SQL statement(s) and put into the READ_LOG. (2) When a new malicious transaction B is detected, the Confinement Executor will set a new confinement time window, the Unconfinement Executor will adjust the U_SET and its unconfining operations to cover B, and the Repair Manager will adjust its damage assessment and repair operations to cover B. (3) When the Repair Manager finishes the

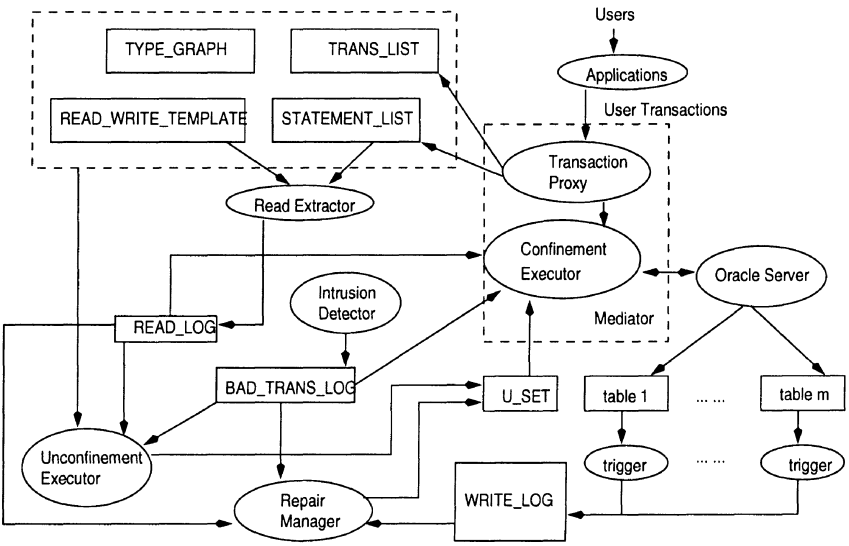

Figure 2. DDCS components.

repair for the set of detected malicious transactions, the Unconfinement Executor will discontinue enforcing the confinement control. (4) When a transaction T commits, the write operations of T will be put into the WRITE_LOG by the triggers.

4.1 Confinement Executor

When the *Confinement Executor* retrieves a new malicious transaction from its message queue, it will perform the confinement operations specified in Algorithm 1. In particular, it will (1) stop executing new transactions, (2) abort all the active transactions, (3) adjust the confinement time window, and (4) allow new transactions to execute after getting a $READY$ message from both the Unconfinement Executor and the Repair Manager saying that the U_SET is adjusted. Since the TRANS_LIST table contains the identifiers of the active transactions, the *Confinement Executor* can ask the DBMS to abort these transactions. Since the start times of transactions are also kept in the TRANS_LIST table, it should be easy to adjust the confinement time window.

When a new user transaction arrives after the above confinement operations are done, the *Confinement Executor* needs to enforce the damage confinement control in such a way that any data object updated within the confinement time window is not allowed to access except the objects U_SET. The confinement control algorithm is as follows. Note that the damage confinement control is enforced in terms of SQL statements instead of transactions since (1) read extraction is also in terms of SQL statements, (2) in some transac-

tions the execution of some later SQL statement may depend on the results of a precious statement, and (3) in this way quicker confinement checking can be achieved. For a transaction with multiple SQL statements, if the first SQL statement wants to read a data object that is confined, we can reject or delay the access of this transaction to the database without checking the reads of any of the other SQL statements.

Algorithm 2 Damage Confinement Control
// Assume the Unconfinement Executor maintains the set of detected malicious
// transactions that are not yet repaired, which is denoted as **B**. For
// simplicity, we assume the write set of a transaction is a part of the read set of the transaction
while TRUE
if a new SQL statement S wants to be executed
if B is not empty
retrieve the read set of this SQL statement from the READ_LOG via Trans_ID and S_Pattern as following: SELECT Table_Name, Record_ID FROM READ_LOG WHERE Trans_ID=S.Trans_ID AND S_Pattern=S.S_Pattern;
// We assume at this moment the Read Extractor has already extracted the reads
// of S, otherwise, the Unconfinement Executor needs to wait for a while
for each read item x in the read set of S, search x in the U_SET table. **If** x is not in the U_SET table
retrieve the timestamp of x from the WRITE_LOG. **If** the timestamp is within the confinement time window
abort or delay the transaction (based on the preference of the user);
jump to the beginning of the loop;
use Algorithm 3 to rewrite the SQL statement;
// At this moment, it is clear that S will not read any data item that is confined
else // B is empty
use Algorithm 3 to rewrite the SQL statement;
end while

DDCS needs the time stamp information to enforce damage confinement control, however, asking the applications to maintain time stamp information is not only not secure because this enables malicious applications to manipulate time stamps, but also not transparent to existing applications which usually do not maintain time stamps. In order to maintain the time stamp information in a transparent and secure manner, DDCS does three things: (1) DDCS transparently adds a TIME_STAMP column to each user table. (2) DDCS rewrites user queries to associate time stamps with each write. (3) DDCS does not allow any (end) user to change time stamp information. The query-rewriting algorithm is as follows. (Due to space limit, only part of the algorithm is presented.)

Algorithm 3 Time Stamp Maintenance by Rewriting Queries
while TRUE
if the SQL statement is a SELECT statement
forward the SQL statement to the Oracle server without any change;
else if the SQL statement is an INSERT statement
if the format is: INSERT INTO tablename (...) VALUES (...)
rewrite to: INSERT INTO tablename (..., timestamp) VALUES (..., SYSDATE);
// The value of SYSDATE, an Oracle system variable, is the current system time.
forward the rewritten SQL statement to the Oracle server;
... ...

```
    else if the SQL statement is an UPDATE statement
        if the format is: UPDATE tablename SET ... WHERE ...
            rewrite to: UPDATE tablename SET ..., timestamp=SYSDATE WHERE ... ;
            forward the rewritten SQL statement to the Oracle server;
    else if the SQL statement is a DELETE statement
        forward the SQL statement to the Oracle server without any change;
end while
```

4.2 Unconfinement Executor

The Unconfinement Executor is responsible for unconfining phases B and C. To enable Phase B, DDCS needs to maintain the dependency relationships among transaction types. In particular, DDCS uses the TYPE_GRAPH table to keep the type dependencies. An example TYPE_GRAPH table is shown in the following table. The first record says that transaction type TA, TE, and TD are affected by TD, directly or indirectly. So the TYPE_GRAPH maintains the transitive closure of the type dependency relationship.

Trans_Type	Affected_Types
TD	TA+TE+TD
TA	TA+TD

When the Unconfinement Executor retrieves a new malicious transaction B_j from its queue, the following two procedures proceed concurrently.

Phase B Procedure. The Unconfinement Executor follows Algorithm 1. To know if this phase has already scanned some of the operations that were performed after B_j commit or not, the Unconfinement Executor compares within the $TRANS\_LIST$ table the location where B_j commit and the location where the current scan is performed. The $TRANS\_LIST$ table keeps the start time and commit time of transactions. Before the Unconfinement Executor scans the $TRANS\_LIST$ to do unconfinement, the Executor first searches the $TYPE\_GRAPH$ table for the types that are affected by $type(\mathbf{B})$ (note that $\mathbf{B}$ is usually a set of malicious transactions). Then for each transaction T, if $type(T)$ is not among the search results, the Executor gets the object types that could have been updated by T from the $READ\_WRITE\_TEMPLATE$ table (i.e., the write set template of $type(T)$), and put these object types in the Q table. To enable the Executor to adjust the U_SET, the Executor associates a timestamp with each object type put into the U_SET in such a way that the timestamp indicates when the object (type) was updated. After the U_SET is adjusted in terms of both Phase B and Phase C, the Executor sends a $READY$ message to the Confinement Executor.

Phase C Procedure. Phase C is similar to Phase B except that (1) the Executor needs to apply the techniques described in [7] to extract the input arguments of transactions from their SQL statements and use the input arguments to materi-

alize both the read set and the write set of each transaction in the history; (2) the Executor needs to use the materialized read and write sets to reason the affecting relationships among transactions in such a way that many transactions that are not affected by **B** can be identified. The timestamps used in Phase C are obtained in the same way as Phase B.

5. Conclusions

In this paper, we present the design and implementation of DDCS, a multiphase database damage confinement system. DDCS enforces multiphase damage confinement to substantially reduce the amount of damage spreading during damage assessment and repair. DDCS can confine the damage caused by multiple malicious transactions in a concurrent manner without any damage leakage. DDCS is built on top of a commercial database server. DDCS is transparent to end users. Our preliminary testing shows that the confinement control process, the unconfinement operations, and the query-rewriting process are pretty efficient, and the performance penalty of DDCS is reasonable.

Acknowledgments

This research was 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, by DARPA/ AFRL under agreement number F20602-02-1-0216, by NSF CCR- 0233324, and by a Department of Energy Early Career Award.

References

[1] Ammann, P., Jajodia, S., McCollum, C. and Blaustein, B. (1997). Surviving information warfare attacks on databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 164-174.

[2] Barbara, D., Goel, R. and Jajodia, S. (2000). Using checksums to detect data corruption. In *Proceedings of the 2000 International Conference on Extending Database Technology.*

[3] Carter and Katz (1996). Computer crime: An emerging challenge for law enforcement. *FBI Law Enforcement Bulletin*, 1(8).

[4] Graubart, R., Schlipper, L. and McCollum, C. (1996). Defending database management systems against information warfare attacks. Technical Report, The MITRE Corporation.

[5] Ingsriswang, S. and Liu, P. (2001). Aaid: An application aware transaction-level database intrusion detection system. Technical Report, Department of Information Systems, University of Maryland, Baltimore County.

[6] Liu, P. and Jajodia, S. (2001). Multi-phase damage confinement in database systems for intrusion tolerance. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop.*

[7] Luenam, P. and Liu, P. (2001). Odar: An on-the-fly damage assessment and repair system for commercial database applications. In *Proceedings of the 2001 IFIP WG 11.3 Working Conference on Database and Applications Security.*

[8] Maheshwari, U., Vingralek, R. and Shapiro, W. (2000). How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating System Design and Implementation.*

[9] McDermott, J. and Goldschlag, D. (1996). Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pp. 176-185.

[10] Smith, S., Palmer, E. and Weingart, S. (1998). Using a high-performance, programmable secure coprocessor. In *Proceedings of the International Conference on Financial Cryptography.*