Multi-Phase Damage Confinement in Database Systems for Intrusion Tolerance

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

Sushil Jajodia
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
jajodia@gmu.edu

Abstract

Preventive measures sometimes fail to defect malicious attacks. With cyber attacks on data-intensive applications becoming an ever more serious threat, intrusion tolerant database systems are a significant concern. Intrusion detectors are a key component of an intrusion tolerant database system. However, a relatively long detection latency is usually unavoidable for detection accuracy, especially in anomaly detection, and it can cause ineffective - to some degree at least - damage confinement. In a busy database ineffective confinement can make the database too damaged to be useful. In this paper, we present an innovative multi-phase damage confinement approach to solve this problem. In contract to a traditional one-phase confinement approach our approach has one confining phase to quickly confine the damage, and one or more later on unconfining phases to unconfine the objects that are mistakenly confined during the first phase. Our approach can ensure no damage spreading after the detection time, although some availability can be temporarily lost. Our approach can be easily extended to support flexible control of damage spreading and multiple confinement policies. Our approach is practical, effective, efficient, and to a large extent assessment independent.

Keywords: Damage Confinement, Intrusion Tolerance, Database Security

1 Introduction

Database security concerns the confidentiality, integrity, and availability of data stored in a database. A broad span of research from authorization [GW76, RBKW94, JSSB97], to inference control [Ada89], to multilevel secure databases [WSQ94, SC98], and to multilevel secure transaction processing [AJG99], addresses primarily how to protect the security of a database, especially its confidentiality. However, very limited research has been done on how to survive successful database attacks, which can seriously impair the integrity and availability of a database. Experience with data-intensive applications such as credit card billing, banking, air traffic control, logistics management, inventory tracking, and online stock trading, has shown that a variety of attacks do succeed to fool traditional database protection mechanisms. In fact, we must recognize that not all attacks – even obvious ones – can be averted at their outset. Attacks that succeed, to some degree at least, are unavoidable. With cyber attacks on data-intensive internet applications, i.e., e-commerce systems, becoming an ever more serious threat to our economy, society, and everyday lives, attack resistant database systems that can survive malicious attacks are a significant concern.

One critical step towards attack resistant database systems is intrusion detection, which has attracted many researchers [D.E87, LTG+92, JL93, HL93, Lun93, MHL94, LM98, LB98, LSM99]. Intrusion detection systems monitor system or network activity to discover attempts to disrupt or gain illicit access to systems. The methodology of intrusion detection can be roughly classed as being either based on *statistical profiles* [JV91, JV94, SM97] or on known patterns of attacks, called *signatures* [Ilg93, GL91, PK92, IKP95, SG97]. Intrusion detection can supplement protection of network and information systems by rejecting the future access of detected attackers and by providing useful hints on how to strengthen the defense. However, intrusion detection has several inherent limitations: (a) Intrusion detection makes the system attack-aware but not attack-resistant, that is, intrusion detection itself cannot maintain the integrity and availability of the database in face of attacks. (b) Achieving accurate detection is usually difficult or expensive. The *false alarm rate* is high in many cases. (c) The average detection latency in many cases is too long to effectively confine the damage.

To overcome the limitations of intrusion detection, a broader perspective is introduced, saying that in addition to

detecting attacks, countermeasures to these successful attacks should be planned and deployed in advance. In the literature, this is referred to as *survivability* or *intrusion tolerance*. In this paper, we will study a critical database intrusion tolerance problem beyond intrusion detection, namely *damage confinement*, and present a set of innovative algorithms to solve the problem.

1.1 The Problem

The damage confinement problem can only be clearly identified in the context of an intrusion tolerant database system. Database intrusion tolerance can be enforced at two possible levels: *operating system (OS) level* and *transaction level*. Although transaction level methods cannot handle OS level attacks, it is shown that in many applications where attacks are enforced mainly through malicious transactions transaction level methods can tolerate intrusions in a much more effective and efficient way. Moreover, it is shown that OS level intrusion tolerance techniques such as those proposed in [Lun93, LM98, MG96a, MG96b, BGJ00], can be directly integrated into a transaction level intrusion tolerance framework to complement it with the ability to tolerate OS level attacks. This paper will focus on transaction level intrusion tolerance, and our presentation will be based on the intrusion tolerant database system architecture shown in Figure 1.

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

The complexity of the framework is mainly caused by a phenomenon denoted *damage spreading*. In a database, the results of one transaction can affect the execution of some other transactions. Informally, when a transaction T_i reads an object x updated by another transaction T_j, T_i is directly *affected* by T_j. If a third transaction T_k is affected by T_i,

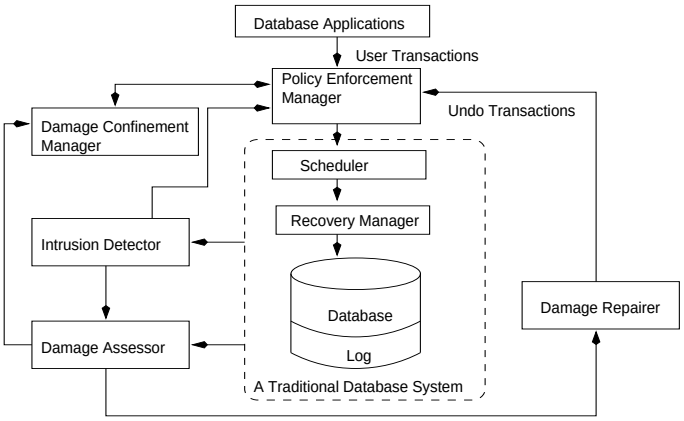


Figure 1. An Intrusion Tolerant Database System Architecture

but not directly affected by T_j, T_k is indirectly affected by T_j. It is easy to see that when a (relatively old) transaction B_i that updates x is identified malicious, the damage on x can spread to every object updated by a transaction that is affected by B_i, directly or indirectly. The job of the *Damage Assessor* and the *Damage Repairer* is to locate each affected transaction and recover the database from the damage caused on the objects updated by the transaction. In particular, when an affected transaction is located, the *Damage Repairer* builds a specific UNDO transaction to *clean* each object updated by the transaction (and not cleaned yet). Cleaning an object is simply done by restoring the value of the object to its latest undamaged version. This job gets even more difficult as the execution of new transactions continues because the damage can spread to new transactions and cleaned objects can be re-damaged by new transactions. Therefore, the main task of this framework is to guarantee that damage spreading is (dynamically) controlled in such a way that the database will not be damaged to a degree that is unacceptable or useless.

The limitation of this architecture is mainly due to the fact that the effectiveness of the framework is heavily dependent on the performance of the *Intrusion Detector* (For example, the framework cannot handle the malicious transactions not captured by the *Intrusion Detector*), and the fact that for accuracy a (relatively) long *detection latency* is usually caused, especially when *anomaly detection* based on statistical profiles is enforced. One main reason for the latency is that intrusion detection has to make a tradeoff between meeting the requirement of reporting an intrusion accurately and the requirement of detecting as many intrusions as possible, which can often result in conflicting design goals. For example, in anomaly detection, for detection

accuracy the anomaly threshold for reporting must be high, thus many intrusions with gradual anomaly cannot be identified; on the other hand, in order to capture more intrusions, the threshold should instead be lower, thus the false alarm rate would increase and many legitimate transactions can be mistaken for malicious and suffer denial-of-service. To resolve this dilemma, extending the monitoring time window is one feasible solution. By collecting and investigating more proofs about a suspicious activity, usually more accuracy can be achieved. However, it should be noticed that even with a fair latency usually no intrusion detector can identify every malicious transaction, so it is possible that some objects are left damaged without being located and repaired. Cleaning these objects is out of the scope of this paper. It should also be noticed that since the false alarm rate is usually not zero, our framework can clean some actually undamaged objects. This issue is also out of the scope of the paper.

An instant impact of the detection latency is that a significant *assessment latency* can be caused. It is easy to see that (a) if every malicious transaction can be identified before it commits, then aborting the transaction prevents the database from being damaged and no assessment is needed; (b) if every malicious transaction can be identified just after it commits, then very little damage can spread, so damage assessment could be done quickly. However, since there is usually a significant detection latency, when a malicious transaction B_i is identified, in the history there can be already a lot of transactions following B_i and many of them may have already been affected, directly or indirectly, by B_i. Damage assessment at this situation will certainly cause a significant delay. In addition to detection delay, another important reason for the assessment latency is the computing time required for assessment. As shown in [LAJ00, AJL98], damage assessment itself can spend substantial computation time, although proportional to the length of the history. The computation needs to mine the history log for the set of affected transactions and locate the objects updated by these transactions.

The problem we want to solve in this paper is not about how to reduce the detection latency or assessment latency, instead, the problem is that significant assessment latency can cause ineffective - to some degree at least - damage confinement. In the framework, damage will not be confined until an object is reported by the *Damage Assessor* as damaged. Hence damage confinement depends on damage assessment. However, since there is usually 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 assessment latency could become even longer. This clearly contradicts with our original motivation of damage confinement. Moreover, in a heavily accessed database this can make the database become too damaged to be useful. Therefore, how to effectively confine damage is a significant concern.

1.2 Our Approach and Contribution

In this paper, we present an innovative multi-phase damage confinement approach to solve the above problem. In contract to prior work where (a) an object x is confined by an one-phase operation; and (b) damage confinement depends on assessment, our approach has one confining phase, denoted *initial confinement*, and one or more later on unconfining phases, denoted *confinement relaxation*, to unconfine the objects that are mistakenly confined during the first phase. Our approach has the following properties:

- Although the initial confinement phase can mistakenly confine some undamaged objects, thus it can cause some availability loss, our approach can guarantee that no damage will spread after the first phase which can be instantly done. Hence the database integrity level can be easily stabilized and the cost of assessment and repair can be dramatically reduced.

- Except for the final unconfining phase, all previous confinement phases, including the confining phase and other unconfining phases, are assessment independent. In this way, confinement and assessment turn from two serial processes to two concurrent and collaborative processes. Hence the negative impact of the assessment latency can be avoided.

- Our approach is efficient and effective, and can be easily extended to support flexible control of damage spreading. To one extreme, no damage spreading is allowed; some computing resources are saved but some availability is lost. To the other extreme, the one-phase approach is taken; maximum availability is got but substantial computing resources can be cost.

- Our approach is practical. It can be seamlessly integrated with the existing damage assessment and repair mechanisms.

The rest of the paper is organized as follows. In Section 2, we formalize the problem. We present our approach in Section 3. In Section 4, we present some alternatives. We address some implementation issues in Section 5. In Section 6, we discuss related work. We conclude the paper in Section 7.

2 Multi-Phase Damage Confinement: The Model

In our model, a *database* is a set of *data objects* (objects for short). Objects are handled by *transactions*. A transaction is a partial order of read and write operations that either *commits* or *aborts*. Since aborted transactions have nothing to do with intrusion tolerance, for simplicity we assume every transaction commits. Two transactions *conflict* if they both have an operation on the same object and one of them is write. The (usually concurrent) execution of a set of transactions is modeled by a structure called a *history*. The correctness of a history is typically captured by the notion of *serializability* [BHG87]. We assume *strict two-phase locking* (2PL) is used to produce serializable histories where the commit order indicates the *serial order* among transactions. Moreover, we denote the set of objects read by a transaction T_i as T_i's *read set*, and we denote the set of objects written by T_i as T_i's *write set*.

In a history, a transaction T_i is *dependent upon* another transaction T_j, if there exists an object x such that T_i reads x after T_j updates it, and there is no transaction that updates x between the time T_j updates x and T_i reads x. The *dependent upon* relation indicates the path along which damage spreads. In particular, if a transaction which updates an object x is dependent upon a malicious transaction which updates an object y, we say the damage on y *spreads to* x, and we say x is *damaged*. If a transaction which updates an object x reads a damaged object y, we also say that the damage on y spreads to x. Moreover, a transaction T_u *affects* transaction T_v if the ordered pair (T_v, T_u) is in the transitive closure of the *dependent upon* relation. Therefore if a malicious transaction B_i affects an innocent transaction G_j, the damage on B_i's write set will spread to G_j's write set, or every object in G_j's write set will be damaged.

Damage assessment can be done by computing which transactions are affected when a malicious transaction B_i is identified because only the write sets of affected transactions will be damaged and such write sets are easy to get from the log. In order to compute the set of affected transactions, we assume a specific graph, denoted *dependency graph*, is used [AJL98]. We define a dependency graph for a set of transactions S in a history as $DG(S) = (V, E)$ in which V is the union of S and the set of transactions that are affected by S. There is an edge, $T_i \rightarrow T_j$, in E if $T_i \in V, T_j \in (V - S)$, and T_j is dependent upon T_i. Since besides S $DG(S)$ includes all and only the transactions affected by a transaction in S, computing $DG(S)$ does the job of assessing the damage caused by S. To illustrate, consider the following history over $(B_1, G_1, G_2, G_3, G_4)$ where B_1 is malicious and others are innocent. $DG(B_1)$ is shown in Figure 2(a). Since the write sets of B_1, G_1, G_2, and G_4 are $\{x, u\}$, $\{x, y\}$, $\{y, v\}$, and $\{u, y, z\}$, perspectively, the damaged part of the database is $\{x, y, u, v, z\}$.

$$
\begin{aligned}
H_1 : & r_{B_1}[x] w_{B_1}[x] r_{B_1}[u] w_{B_1}[u] c_{B_1} r_{G_1}[x] w_{G_1}[x] r_{G_3}[z] \\
& w_{G_3}[z] c_{G_3} r_{G_1}[y] w_{G_1}[y] c_{G_1} r_{G_2}[y] w_{G_2}[y] r_{G_2}[v] \\
& w_{G_2}[v] c_{G_2} r_{G_4}[u] w_{G_4}[u] r_{G_4}[y] w_{G_4}[y] r_{G_4}[z] w_{G_4}[z] c_{G_4}
\end{aligned}
$$

We model the initial confinement phase as a simple estimation problem. When a malicious transaction B_i is detected, the initial confinement phase gets the set of objects to confine by 'estimating' which objects in the database may have been damaged. This set is called a *confinement set*, denoted S_E. Accurate damage assessment is not possible in this phase, since our approach requires the confining phase be finished in a very short period of time so that the damage can be confined before any new transaction is executed (to possibly spread the damage), and the *Damage Assessor* usually needs a much longer period of time to do the assessment. Assume at this moment the set of objects that are really damaged is S_D, then the relation between S_E and S_D is of four types: (1) $S_E = S_D$ (exact estimation); (2) $S_E \supset S_D$ (over estimation); (3) $S_E \subset S_D$ (not enough estimation); (4) $S_E \cap S_D \neq S_E$, and $S_E \cap S_D \neq S_D$ (approximate estimation).

Exact estimation is our goal, but it is almost impossible to achieve. Over estimation is a reasonable strategy in most cases because it can guarantee that after S_E is confined no damage will spread. Not enough estimation can only support partial confinement, and it is too restrict for an estimation. When approximate estimation is enforced, confined data objects may not be damaged and damaged objects may not be confined. Over estimation and approximate estimation are two better strategies. However, as we will mention in Section 4, none of these two strategies is strictly better than the other. In this paper, we will focus on over estimation, although in Section 4 some approximate estimation methods are discussed.

We model each later on unconfining phase as a process to transform one confinement set to another. Hence the whole multi-phase confinement process can be modeled by a sequence of confinement sets, denoted $S_E, S_2, S_3, ...S_n, S_D$. S_E indicates the result of the initial confinement phase. S_i, $2 \leq i \leq n$, indicates the result of the ith phase. The goal of the whole confinement phase is to converge the sequence to S_D. The sequence can be converged to S_D in many different ways. The way our approach will take is that $S_E \supseteq S_1$, $S_i \supseteq S_j$ for $i < j$, and $S_n \supseteq S_D$. It should be noticed that the above discussion does not take into account the objects that are cleaned by the *Damage Repairer* during the confinement process. Typically the objects cleaned during a confinement phase i should be removed from S_i.

Integrity and availability of a database are measured in different ways in different contexts. In our model, *availability* is measured by the percentage of unconfined objects, and *integrity* is measured by the percentage of undamaged

or clean data objects. Note that our measurement of integrity is very different from traditional database systems where integrity is measured based on integrity constraints.

It is helpful to reconsider the different kinds of multi-phase damage confinement processes from another angle, namely *damage leakages*. In our model, *damage leakages* of the initial confinement phase are measured by the set of objects that are not put into S_E. Damage leakages of an unconfining phase are measured by the set of objects that are damaged but are mistakenly unconfined. Multi-phase damage confinement mechanisms can be broken down into two categories based on whether or not they might cause damage leakages during the whole confinement process. A multi-phase damage confinement mechanism is *strict* if no damage leakages are caused in either the confining phase or some later on unconfining phase, otherwise, it is *unstrict*. When a strict mechanism is enforced, no damage will spread, and the damage assessment process can be terminated after the last transaction that commits before the detection of B_i is scanned because no transaction executed after the detection of B_i will be affected. However, when a mechanism is unstrict, the *Damage Assessor* needs to continue scanning new transactions until all the damage spread to these new transactions is repaired. More comparison of these two categories will be discussed in Section 4.

3 Multi-Phase Damage Confinement: The Approach

In this section, we will present a strict multi-phase confinement approach. The techniques for approximate or unstrict confinement are addressed in Section 4. We start with a basic scheme using time stamps. However, Scheme I can cause damage leakages. We then show how to extend Scheme I to achieve strict confinement and less availability loss. At last, we describe our final scheme which satisfies all the properties we mentioned earlier. For clarity, we assume after a malicious transaction B_i is identified no other malicious transactions will be identified until the damage caused by B_i is repaired. Our approach can be easily extended to support multiple malicious transactions.

3.1 Data Structures

Our approach uses the following major data structures:

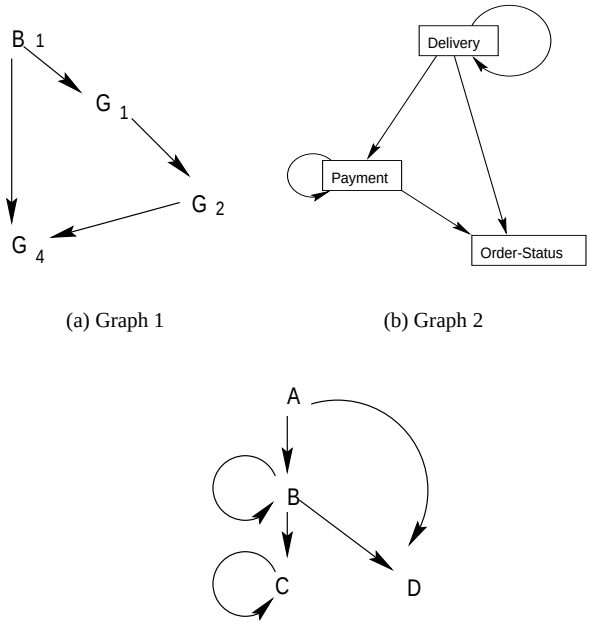
- each object x is associated with a *time stamp*, denote ts_x, which indicates when x is updated. Time stamps are supported by many DBMS, for example, Oracle supports time stamps by maintaining them in a specific field of a record.
- a *Committed Transaction Table* (CTT) that is an array of records. Each record has four fields: (1) a transaction ID that uniquely identifies a transaction; (2) a commit time that indicates when the transaction commits; (3) a transaction type that indicates the category of the transaction (Note that ad-hoc transactions have no transaction types); and (4) a list of input arguments of the transaction.
- an unconfinement set, denoted S_U, which maintains the set of objects that should be unconfined. S_U is changed from time to time.
- a dependency graph for the identified malicious transaction B_i, denoted $DG(B_i)$.
- a dependency graph for the type of B_i, denoted $DG(type(B_i))$. This graph is built in a way sightly different from a normal dependency graph.
- a materialized version of $DG(type(B_i))$.
- the log that records the history. We assume every read or write operation of the history is recorded in the log (COTS DBMS usually do not record read operations, techniques for getting read information are addressed in [AJL98]).

3.2 Scheme I: Time Stamp Based Damage Confinement

When a malicious transaction B_i is identified:

Initial confinement: The initial confinement is done by rejecting the access from users to B_i's write set and every object x associated with a time stamp later than the commit time of B_i, denoted ct_{B_i}, which is obtained from the CTT table (Note that all the objects updated by B_i must be associated with a time stamp earlier than ct_{B_i}). The rational is that any object x which has not been updated after B_i commits, except those updated by B_i, will not be damaged, because if we suppose such a x is damaged, then since damage can only spread along the *affect* relation, x must be updated by a transaction G_j which is affected by B_i. However, according to strict 2PL the fact that G_j is affected by B_i implies that x is updated after B_i commits. This contradicts the assumption.

Although when the initial confinement is enforced no new transaction can access confined objects, continuous execution of the set of *active* transactions (if any) that had read some confined objects before the initial confinement (but will not read any confined object after the initial confinement) can certainly spread the damage. Hence these active transactions must be aborted. Since only the objects that are updated after B_i commits could be damaged, and since no active (or new) transaction will spread the damage, the initial confinement phase causes no damage leakages.



(a) Graph 1 (b) Graph 2

(c) Graph 3

Figure 2. Example Dependency Graphs

Confinement Relaxation: We assume the *Damage Assessor* uses $DG(B_i)$ to do damage assessment. $DG(B_i)$ is built as the history log is scanned by the *Damage Assessor*. The scanning starts at B_i. When a transaction G_j is scanned, if it is not in $DG(B_i)$ then its write set will be put into S_U, that is, G_j's write set are unconfined and open to access again. The rational is that since G_j is not affected, G_j's updates should cause no damage. If G_j is in $DG(B_i)$, then a specific UNDO transaction is executed to clean the damage on G_j's write set. Since every transaction will be logged, an UNDO transaction can also be scanned. When an UNDO transaction U_k is scanned, U_k's write set will be put into S_U. The rational is that after a damaged object is cleaned it should be unconfined.

In order to enable confinement relaxation, when a transaction asks to access an object x we let the PEM first check if $ts_x \leq ct\_B_i$. If so the access is allowed, if not we let the PEM check if x is in S_U. If x is in S_U then the access should be allowed (subject to other DBMS access control rules). Moreover, in order to support UNDO transactions, we let the PEM allow UNDO transactions to access objects updated after B_i commits.

3.3 Scheme II: Handling Damage Leakages during Assessment and Repair

Scheme I has a serious problem, that is, it can cause damage leakages. Reconsider history H_1 presented in Section 2. Here we assume B_1 is identified after G_4 commits. Since G_3 is not in the graph (shown in Figure 2(a)) when it is scanned, according to Scheme I G_3's write set, namely $\{z\}$, will be unconfined. However, the fact that z is not damaged after G_3 is executed does not guarantee that z will not be damaged later. In fact, when z is later on updated by G_4, the damage spreads from B_1's write set and G_2's write set to z. Since z is damaged before the initial confinement, and when z is unconfined it is impossible that z is cleaned by an UNDO transaction (because the UNDO transaction that cleans z will only be executed when G_4 is later on scanned), so z is damaged when it is unconfined. Although when G_4 is found affected z can be re-confined, till then the leaked damage may have already spread to many other objects.

The goal of Scheme II is to avoid the damage leakages caused by Scheme I. The algorithm is as follows. Note that here only the differences from Scheme I are presented.

- When a transaction G_j is scanned, if it is not in $DG(B_i)$ then instead of putting the whole write set of G_j into S_U, the PEM checks the time stamp of each object x updated by G_j. If $ts_x \leq ct_{G_j}$, then the PEM puts x into S_U; otherwise, the PEM keeps x confined.

As shown in Theorem 1, Scheme II will cause no damage leakages. Therefore Scheme II bounds the assessment and repair process within the part of the history that begins with B_i and ends when the initial confinement phase is done, thus Scheme II can significantly simplify the intrusion tolerance process.

Theorem 1 Scheme II causes no damage leakages.

Proof: During the confinement relaxation phase, when a transaction G_k is scanned and found not in $DG(B_i)$, for each object x updated by G_k, if $ts_x \leq ct_{G_k}$, we are sure that x is latestly updated by G_k. Since G_k is not affected, x is not damaged. If $ts_x > ct_{G_k}$, then there must be at least one transaction updating x after G_k commits. If all these transactions are unaffected by B_i, then x is undamaged; otherwise, x will be damaged by at least one of these transactions. Hence keeping x confined when $ts_x > ct_{G_k}$ avoids all the possible damage leakages. Note that it is impossible for ts_x to be later than the initial confinement phase, because if so then x must have been updated after the initial confinement phase. Since x is updated by G_k after B_i commits, the initial confinement phase will confine x until G_k is scanned, so no transactions executed after the initial confinement phase can update x before G_k is scanned. This contradicts with the assumption. □

3.4 Scheme III: Exploiting Transaction Access Patterns

Schemes I and II enforce the initial confinement based on time stamps and confine every object updated after B_i commits. However, in most cases only a small portion of the objects updated after B_i commits are damaged. Therefore, Schemes I and II can confine a large number of undamaged objects during the confining phase. Although later on unconfining phases can unconfine these undamaged objects, the *relaxation latency*, which is measured by the time window between the time an undamaged object x is mistakenly confined and the time x is unconfined, is usually long because Schemes I and II depend on a slow assessment-triggered unconfining process. As a result, substantial availability can be lost during the relaxation latency.

The goal of scheme III is to reduce the relaxation latency. The idea is to exploit transaction profiles to insert one or two assessment independent, much quicker unconfining phases between the confining phase and the assessment triggered unconfining phase. The advantage is that with a much shorter relaxation latency a large number of undamaged but confined objects could be unconfined by these assessment independent phases. It should be noticed that this approach is only good for canned applications where (1) the code of each transaction is fixed and pre-known, and (2) transactions running the same code belong to the same *type*. For ad-hoc transactions, since transaction codes are known only at run time, no pre-computations can be done to reduce the relaxation latency. In the following, we assume each transaction belongs to a specific type and the *profile* or code for each transaction type is pre-known.

The assessment independent unconfining phases still use a dependency graph to compute which objects are not damaged. However, the dependency graphs used here are built not based on the log, but based on transaction profiles. The first step to build the graphs is to extract the read and write sets of a transaction from its profile. Here we use an example to show the idea. We call the read and write sets extracted from transaction profiles *templates* because they are not captured from real transaction executions.

We start with the transaction profile of TPC-A, a well known database benchmark [Gra93], as an example. TPC-A is stated in terms of a hypothetical bank. The bank has one or more branches. Each branch has multiple tellers. The bank has many customers , each with an account. The database represents the cash position of each entity (branch, teller, and account). TPC-A has only one type of transactions, which represents the work done when a customer makes a deposit or a withdrawal against his account. The transaction profile is specified as follows:

Input: Aid, Tid, Bid, Delta
BEGIN TRANSACTION
  Update Account_Balance where Account_ID = Aid:
    Read Account_Balance from Account
    Set Account_Balance = Account_Balance + Delta
    Write Account_Balance to Account
  Write to History:
    Aid, Tid, Bid, Delta, Time_stamp
  Update Teller where Teller_ID = Tid:
    Set Teller_Balance = Teller_Balance + Delta
    Write Teller_Balance to Teller
  Update Branch where Branch_ID = Bid:
    Set Branch_Balance = Branch_Balance + Delta
    Write Branch_Balance to Branch
COMMIT TRANSACTION

Here, Aid(Account_ID), Tid(Teller_ID), and Bid(Branch_ID) are keys to the relevant records. The read and write set templates for this type of transactions are specified as follows. Note that they are specified at both the tuple (record) level where each object denotes a record, and the element (field) level where each object denotes a field of a record (Finer confinement is achieved at the element level).

At the tuple level:
  Read_Set= {Account.Aid, Teller.Tid, Branch.Bid}
  Write_Set= {Account.Aid, Account.Tid, Branch.Bid}
At the element level:
  Read_Set= { Account.Aid.Account_Balance,
  Teller.Tid.Teller_Balance, Branch.Bid.Branch_Balance }
  Write_Set= { Account.Aid.Account_Balance,
  Teller.Tid.Teller_Balance, Branch.Bid.Branch_Balance }

Getting a template is generally not a easy job. First, in many applications, a transaction can run several SQL statements and each statement can be very complex. Second, many transactions have conditional branches in their control flows. Which branch will be chosen in a real execution depends on the input arguments and the current database state. In [AJL98], a general approach to extract read set templates, which can be easily extended to extract write set templates, is proposed. Several specific guidelines (or rules) are provided to deal with the complexities we have mentioned above. In this paper, we justify the feasibility of this approach using a practical inventory management database application which handles millions of records. In particular, we investigate how to extract templates from TPC-C transaction profiles. TPC-C benchmark [Gra93] simulates a practical inventory management database application. The results of our study, namely the read and write set templates of TPC-C profiles, are shown in Appendix B. Our study shows that good templates can be got from real world applications such as TPC-C.

Templates can tell us which kind of data objects (i.e., which table or which column) will be read or written by

a transaction, but cannot tell us which objects are actually read or written by the transaction. In order to get a concrete read or write set of a transaction T, we have to *materialize* T's templates using T's input arguments. Materialization is usually not a complicated process (A general approach is proposed in [AJL98]). For example, for a TPC-A transaction instance with the input Aid='A1591749', Tid='T0002', Bid='BGMU001', Delta=$1000, the read and write sets of the transaction at the element level are as follows. It is clear that materialized templates are much more concrete than unmaterialized templates.

Read_Set = Write_Set= {
Account.'A1591746'.Account_Balance,
Teller.'T0002'.Teller_Balance,
Branch.'BGMU001'.Branch_Balance }

Templates are useful for damage confinement in two ways: first, a dependency graph built from materialized templates, called a *materialized dependency graph*, can be used to do quicker confinement relaxation. A materialized template of a transaction T looks just like a normal read or write set. So a materialized dependency graph can be built in generally the same way as a normal dependency graph. The only difference is that the materialized read (write) set of T may be just an approximation of the real read (write) set of T because the complexities of transactions can result in such templates that make exact materialization impossible (For example, some templates of TPC-C do not support exact materialization). Since materialization is much quicker a process than damage assessment, using materialized dependency graphs can significantly reduce the relaxation latency for the set of objects that can be unconfined by materialized dependency graphs.

Second, a dependency graph built from unmaterialized templates, called a *type dependency graph*, can be used to do even quicker confinement relaxation, although using type dependency graphs can only unconfine a much smaller number of objects than using materialized dependency graphs in many cases. In our approach, a type dependency graph is built in a way slightly different from a normal dependency graph. In particular, (1) the *dependent upon* relation is defined in a way that is execution order irrelevant. A transaction type Y_i is *dependent upon* another transaction type Y_j if the intersection of Y_j's write set template and Y_i's read set template is not empty; (2) a transaction type can be dependent upon itself. The type dependency graph for $type(T)$ captures all and only the types of transactions whose execution (after T) could be affected by T. To illustrate, the type dependency graph for TPC-C **Delivery** transactions is shown in Figure 2(b). Based on this graph, it is easy to see that if B_i is a delivery transaction, then any **New Order** or **Stock Level** transaction will not be affected.

The advantage of using type dependency graphs is that since all type dependency graphs can be pre-computed and since checking whether or not a transaction G_j is affected can (basically) be done by checking whether or not $type(G_j)$ is in the type dependency graph for $type(B_i)$, this method typically causes much less relaxation delay than other methods. The limitation of using type dependency graphs is that since materialized templates are much more concrete than unmaterialized templates, using type dependency graphs can only unconfine a much smaller number of objects than using materialized dependency graphs in many cases. For example, one TPC-A transaction and another TPC-A transaction are always dependent upon each other if we use templates directly. However, if these two transactions are for different Account IDs, Teller IDs, and Branch IDs, then they are independent upon each other.

In order to ensure no damage leakages when exploiting transaction access patterns, we need to enforce the following constraint on the process of extracting templates:

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

To end this section, we present an integrated algorithm, which is as follows, to exploit transaction access patterns to do quicker confinement relaxation. The algorithm is built on top of Scheme II. The algorithm uses both materialized dependency graphs and type dependency graphs to do confinement relaxation, although in two different consecutive unconfining phases. Note that these unconfining phases could overlap with each other during some period of time. Note also that damage assessment is still necessary because it can achieve exact unconfinement.

- The initial confinement is done in the same way as Scheme II.

- The first unconfining phase uses type dependency graphs. We scan the transactions kept in the *CTT* table after B_i (following the commit order) until a transaction that commits after the initial confinement is scanned. During the scan process, we use an extra set, denoted Q_U, to temporarily contain the objects that could be unconfined later on. For each transaction G_k that is scanned, if $type(G_k)$ is not in the type dependency graph for $type(B_i)$, then we put each object type $type(x)$, which is kept in G_k's write set template, into Q_U. If $type(G_k)$ is in the type dependency graph for $type(B_i)$, then we remove the intersection of Q_U and G_k's write set template out of Q_U. When the scan

process ends, we move every object type kept in Q_U to S_U.

During this unconfining phase, we need Q_U because an object type which is found undamaged during a previous scan could be damaged by a transaction scanned later on. Although an object type can be put into S_U only after this unconfining phase is finished, since this unconfining phase does not need to scan the log or materialize templates, it is still (in general) much quicker than other unconfining phases. Finally, it should be noticed that unconfining an object type is equivalent to unconfining every object of this type. For example, in TPC-A object type Account.Aid.Account_Balance indicates the whole Account_Balance column of the Account table.

- The second unconfining phase which uses materialized dependency graphs starts to build the materialized dependency graph for B_i, denoted $MDG(B_i)$, instantly after B_i is identified. When a transaction G_k is found not in $MDG(B_i)$, for each object x kept in G_k's materialized writeset, if $ts_x \leq ct_{G_k}$, then we put x into S_U.

- The third unconfining phase is just the confinement relaxation phase of Scheme II. The only difference is that when an object x is asked to be unconfined it may have already been unconfined during the first or the second unconfining phase. At this situation, the unconfining operation is not needed.

- When a new transaction G_j wants to access an object x, since S_U can contain object types, we may need to match x with an object type at this stage.

The correctness of Scheme III is shown in the following theorem. It can be proved in a way similar to Theorem 1.

Theorem 2 Scheme III causes no damage leakages, if the Containment Rule is not violated.

Scheme III does not start to built $MDG(B_i)$ until B_i is identified. In some cases materialized dependency graphs can be maintained on-the-fly to further reduce the relaxation delay. However, since we cannot know which transaction is malicious in advance, we need to maintain a graph for each transaction, or maintain the dependency upon relationships among all the transactions.

3.5 Scheme IV: Stateful Damage Confinement

Scheme III uses the type dependency graph for $type(B_i)$) in a *stateless* fashion, that is, when a transaction G_k is scanned, Scheme III simply checks if $type(G_k)$ is in the graph or not, without using the state information about previously scanned transactions. As a result, Scheme III can prevent many undamaged objects from being unconfined. The reason is that many transactions of such transaction types that are indirectly affected by $type(B_i)$ may actually not be affected by B_i. Consider the following history where each transaction is associated with a transaction type:

$$H = (B_i, A)\ (G_2, C)\ (G_3, C)\ (G_4, B)$$

Assume the type dependency graph for $type(B_i)$ is as shown in Figure 2(c), when G_2 and G_3 are scanned, since $type(G_2)$ and $type(G_3)$ are in the type dependency graph, according to Scheme III their writesets will not be unconfined. However, since type C is not directly affected by type A, and between B_i and G_2 there is no transaction of type B executed, so neither G_2 nor G_3 has been affected by B_i, so their writesets should be (at least partially) unconfined.

In order to solve this problem, Scheme IV works as follows. Note that here only the difference from Scheme III is described. Note also that the way we use $MDG(B_i)$ cannot be optimized because $MDG(B_i)$ is built based on the real history along the commit order, thus the above problem will not be caused when using materialized dependency graphs. It is clear that Scheme IV will unconfine G_2 and G_3 in the above example.

- During the first unconfining phase, we dynamically maintain a type list which is initialized with $type(B_i)$. As a transaction G_k is scanned, if $type(G_k)$ is not directly affected by any type kept in the list, then for each object type kept in G_k's write set template, we put the object type into Q_U. Otherwise, we add $type(G_k)$ to the type list (and remove the intersection of Q_U and G_k's write set template out of Q_U).

4 Additional Considerations

4.1 Strict vs. Unstrict Damage Confinement

As we mentioned in Section 2, strict damage confinement mechanisms have the merit that a lot of damage assessment and repair work can be saved. The limitation is that compared with unstrict confinement usually less availability is provided. Schemes II, III, and IV are strict damage confinement mechanisms.

Although unstrict damage confinement mechanisms have the advantage of providing more availability (via looser initial confinement), it faces a critical challenge, that is, since the leaked damage can keep on spreading as new transactions are executed and old damage is repaired, it is possible that the assessment and repair process can never be finished if the assessment and repair speed is slower than

the damage spreading speed. Although we can make the recovery process be terminated in a limited amount of time by reducing the speed of damage spreading through some administration interference, i.e., slowing down the process of new transactions, we still face the following problem: how can the system detect the termination? Without this ability, the recovery process will continue even if there is no damage to repair. We denote this problem the *termination detection problem*. Fortunately, in [AJL98] it is found that the termination detection problem is solvable, and a solution is proposed. The basic idea is to use current repair state information to reason whether or not new damage is possible to be caused.

In summary, both strict confinement and unstrict confinement are feasible. The basic relation between them is a trade-off between availability and computing resources. Strict confinement mechanisms trade availability for resources. Unstrict confinement mechanisms trade resources for availability. In real applications, it is the job of the system security officer to do the trade-off based on the semantics of the application. It should be noticed that the trade-off is highly application dependent.

4.2 Approximate Damage Confinement Methods

In this section, we discuss several ideas to do unstrict damage confinement that allows damage leakages, the advantage is that more availability can be provided. As shown above, the damage leakages can be located and repaired by the Damage Assessor and the Damage Repairer. Unstrict or approximate damage confinement can be done based on time stamps, transaction access patterns, or object access patterns. Moreover, approximate damage confinement can happen during the initial confining phase and each following unconfining phase.

- During the initial confining phase, we can break down the time after ct_{B_i} into several time windows or slots, and cluster the objects updated within each time slot with a set. As a result, we can get a sequence of sets, denoted S_1, S_2, ..., S_n. Then we can estimate the probability that an object x in S_i is damaged, and use the probability to do approximate confinement. For example, we can confine a set S_i only if the probability associated with S_i is above a specific threshold. The probability can be estimated based on previous attack experiences. For example, if among 10 previous attacks S_i (relative to each attack) is affected 6 times, and on average 20% of S_i are damaged each time, then the probability can be measured by $0.6 * 0.20 = 0.12$. Note that here (1) we assume that each object is equally important; (2) we treat an object damaged multiple times and an object damaged only once equally. More advanced methods are certainly possible.

  Time stamp based approximate confinement is typically more effective when the application is (relatively) time dependent. For example, for a bank at the end of each month there is usually a burst of summarizing and reporting transactions; but at the beginning of each month, there are very few such transactions. However, if the distribution of transactions is not relevant to time, then the effectiveness of this method will be very poor. Therefore, this method is heavily dependent on application semantics.

- Object access patterns can be used to achieve application independent unstrict damage confinement. Moreover, this method does not require the application to use only canned transactions. The idea is do confinement based on a probabilistic object affecting pattern. For each object x, based on the previous object access history, we can estimate the probability that another object y is affected by x, directly or indirectly. Then we can construct a *susceptible object set* for x, which contains every object whose probability of being affected by x is above a specific threshold. When a malicious transaction B_i is identified, we can just confine the objects in the union of the susceptible object sets for each object updated by B_i.

- The first unconfining phase can be adapted to support approximate confinement. One idea is that we can measure how serious one type B can be affected by another type A and use the measurements to do approximate confinement. For example, when a malicious transaction B_i is identified, we can confine only the transactions whose types will probably be seriously affected by $type(B_i)$.

- The second and the third unconfining phases can also be adapted to support approximate confinement. For example, when a transaction G_k is scanned and found not in $DG(B_i)$ or $MDG(B_i)$, then for each object x in G_k's writeset, if $ts_x > ct_{G_k}$ and the probability that x has been damaged is above a specific threshold, then we confine x. Otherwise, we do not confine x. Here the probability is estimated based on the previous access history.

5 Implementation Issues

A multi-phase damage confinement system (prototype) is being developed in the context of the intrusion tolerant database system architecture shown in Figure 1. The prototype is being implemented on top of an Oracle 8i DBMS. The main components of the prototype are shown in Figure 3 (For clarity, the inputs to the Intrusion Detector are not shown). In general, the Intrusion Detector is responsible

for reporting malicious transactions; the PEM is responsible for the initial confinement phase and confinement enforcement; the Damage Confinement Manager is responsible for the first and the second unconfining phases; the Damage Assessor is responsible for the third unconfining phase.

We maintain the time stamp of an object in a specific field of the table where the object is stored. Time stamps can be maintained at either the record level or the element level. Here we maintain a time stamp for each record. Since our system is transparent to user applications, if a user application does not keep time stamps, we let the PEM, which proxies every transaction, rewrite each INSERT or UPDATE SQL statement in a slightly different way such that whenever a record is inserted or updated, the corresponding time stamp will be kept in the TIME_STAMP field of the record. Moreover, the CTT table is maintained by the PEM. S_U is maintained by both the Damage Confinement Manager and the Damage Assessor.

One key challenge is how to get read and write operation information. Oracle Redo logs record every write operation, however, unfortunately its structure is confidential, so getting information about writes from Oracle Redo logs is very difficult. This prototype uses triggers to capture the write operations associated with a DELETE, UPDATE, or INSERT statement. Moreover, this prototype uses read set templates to gather information about reads through materialization (Note that Oracle does not maintain read information). For this purpose, the *Transaction Proxy* module proxies every SQL statement. We assume user applications use *OCI calls*, a standard interface for Oracle, to access the database. To proxy user transactions, the Transaction Proxy proxies every OCI call by providing the Applications with a *pseudo* OCI interface, which forwards each OCI call to the Proxy instead of the Oracle Server. In this way, every SQL statement together with its input arguments can be captured by the Proxy, and the CTT table can also be easily maintained. Using these input arguments and the read set templates extracted from transaction profiles, the *Read Extractor* can trace the reads of transactions. Note that in addition to OCI calls, the prototype can be easily extended to support a variety of other Oracle interfaces such as ODBC, JDBC, and SQL*NET.

Using the read and write operation information kept in the Write Log and the Read Log, the Damage Assessor can handle the third unconfining phase when a malicious transaction is reported. Using the read and write set templates kept in the Read and Write Set Template Table, the Read Log, the input arguments kept in the SQL Statement Table, and the CTT table, the Damage Confinement Manager (1) can build $MDG(B_i)$ and handle the second unconfining phase, and (2) can build the dependency graph for $type(B_i)$ and handle the first unconfining phase.

Another key challenge is confinement enforcement,

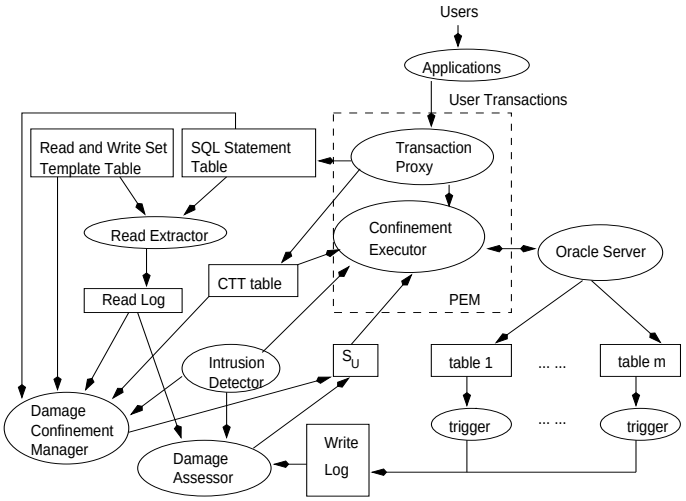


Figure 3. Components of the Multi-Phase Damage Confinement System

which is handled by the PEM based on S_U, the CTT table, and the time stamps associated with each record. In particular, first, when a SQL statement of a transaction is proxied by the PEM, how can the PEM know which objects the statement will access (so that the PEM can check whether or not these objects are in S_U)? Second, since Oracle 8i can only grant and revoke table-specific and column-specific privileges, and cannot handle row-specific or field-specific privileges, the DBMS is unable to ensure that a confined record (or field) will not be accessed, and the PEM has to enforce this kind of confinement by itself. Fortunately, we have figured out some ad-hoc solutions. For example, templates can be used to help the PEM estimate the set of objects that a transaction wants to access, and SQL statement rewriting can be used to do time stamp based access control.

One significant concern about the prototype is its performance. We found that the major impact of the multi-phase damage confinement system on normal transaction processing is the service delay caused by the PEM. Our preliminary testing using simulated data shows that the average response time delay caused (only) by the Proxy is between 10% and 30%, which is reasonable. Although adding the *Confinement Executor* module will cause more delay, the extra delay can still be reasonable since for a transaction with multiple SQL statements, the confinement checking on previous statements and the execution of later on statements can be concurrently processed. Finally, it should be noticed that the performance can be significantly improved if the damage confinement mechanism is integrated into the DBMS

kernel.

6 Related Work

Although database intrusion tolerance is a new topic, there are already some work in this field. In [AJMB97], a fault tolerant approach is taken to survive database attacks where (a) several phases are suggested to be useful, such as attack detection, confinement, damage assessment, attack recovery, and fault treatment; (b) a color scheme for marking damage and repair in databases and a notion of integrity suitable for databases that are partially damaged are used to develop a mechanism by which databases under attack could still be safely used. Their mechanism can be viewed as a confinement mechanism. However, their mechanism assumes that each data object has an (accurate) initial damage mark, our approach does not. In fact, our approach focuses on how to automatically mark (and confine) the damage, and how to deal with the negative impact of inaccurate damage marks. These issues are not addressed in [AJMB97].

In [LJM], an interesting technique called *isolation* is proposed. The idea is setting up a separate environment for allowing suspicious transactions to be executed under surveillance without risking further harm to the system. Isolation can immunize the database from the damage caused by suspicious transactions without suffering denial-of-service. Isolation can confine the damage caused by suspicious transactions, however, isolation cannot confine the damage caused by the transactions that are not isolated. Moreover, for cost reasons, many suspicious transactions will not be isolated. Therefore, although isolation cannot be used to replace multi-phase damage confinement mechanisms, isolation can effectively complement multi-phase damage confinement.

Damage recovery has been investigated in several works. In [JMA99], a general model of damage recovery, and some general principles of trusted recovery are proposed. Damage recovery techniques have been developed from two angles: (1) Since damage is caused and spread by transactions, so damage can be recovered in terms of transactions. In [AJL98, LAJ00], a set of transaction oriented damage recovery algorithms are proposed. The damage assessment and repair algorithms used in our framework are adapted from [AJL98]. (2) Since the subjectives of attacks are data objects, so damage can be recovered in terms of objects. In [PG98], a data object oriented framework is presented where transactions are used to identify damage spreading, and a specific piece of code is constructed to repair each corrupted object. Data object oriented approaches can exploit blind-write transactions to achieve more accurate damage locating in an easier way, however, constructing specific pieces of code to do repair is not only expensive, but also prone to errors.

Our framework cannot (directly) handle OS level attacks. Some other techniques can. In [MG96a, MG96b] a technique is proposed to detect *storage jamming*, malicious modification of data, using a set of special *detect objects* which are indistinguishable to the jammer from normal objects. Modification on detect objects indicates a storage jamming attack. In [BGJ00], checksums are smartly used to detect data corruption. Both *detect objects* and checksums can be used to make our framework resistant to OS level attacks.

7 Conclusion

With cyber attacks on data-intensive applications becoming an ever more serious threat, intrusion tolerant database systems are a significant concern. Intrusion detectors are a key component of such systems. However, the detection latency, which is usually significant, can cause ineffective - to some degree at least - damage confinement. In this paper, we present an innovative multi-phase damage confinement approach to solve this problem. Our approach can ensure no damage spreading after the detection time, although some availability can be temporarily lost. Our approach can be easily extended to support flexible control of damage spreading and multiple confinement policies. Our approach is practical, effective, efficient, and to a large extent assessment independent.

As mentioned earlier in the paper, in many cases approximate methods can be very useful when availability is the most wanted thing. Some general ideas of approximate confinement are mentiond in Section 4, detailed study of these approximate methods could be a good topic for future research.

Acknowledgments

Peng Liu is 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. In particular, our work is supported by DARPA/ISO's Organically Assured and Survivable Information Systems (OASIS) Program. We also thank the anonymous referees for many helpful comments on an earlier version of this paper.

References

[Ada89] M. R. Adam. Security-Control Methods for Statistical Database: A Comparative Study. *ACM Computing Surveys*, 21(4), 1989.

[AJG99] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[AJL98] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. Technical report, George Mason University, Fairfax, VA, 1998.

[AJMB97] P. Ammann, S. Jajodia, C.D. McCollum, and B.T. Blaustein. Surviving information warfare attacks on databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–174, Oakland, CA, May 1997.

[BGJ00] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Proceedings of the 2000 International Conference on Extending Data Base Technology*, Mar 2000.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[D.E87] D.E.Denning. An intrusion-detection model. *IEEE Trans. on Software Engineering*, SE-13:222–232, February 1987.

[GL91] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

[Gra93] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., 2 edition, 1993.

[GW76] P. P. Griffiths and B. W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.

[HL93] P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, 1993.

[IKP95] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[Ilg93] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1993.

[JL93] R. Jagannathan and T. Lunt. System design document: Next generation intrusion detection expert system (nides). Technical report, SRI International, Menlo Park, California, 1993.

[JMA99] S. Jajodia, C. D. McCollum, and P. Ammann. Trusted recovery. *Communications of the ACM*, 42(7):71–75, July 1999.

[JSSB97] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 474–485, May 1997.

[JV91] H. S. Javitz and A. Valdes. The sri ides statistical anomaly detector. In *Proceedings IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 1991.

[JV94] H. S. Javitz and A. Valdes. The nides statistical component description and justification. Technical Report A010, SRI International, March 1994.

[LAJ00] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovery from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[LB98] T. Lane and C.E. Brodley. Temporal sequence learning and data reduction for anomaly detection. In *Proc. 5th ACM Conference on Computer and Communications Security*, San Francisco, CA, Nov 1998.

[LJM] P. Liu, S. Jajodia, and C.D. McCollum. Intrusion confinement by isolation in information systems. *Journal of Computer Security*. To appear.

[LM98] Teresa Lunt and Catherine McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corporation, McLean, VA, 1998.

[LSM99] Wenke Lee, Sal Stolfo, and Kui Mok. A data mining framework for building intrusion detection models. In *Proc. 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[LTG+92] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H. S. Javitz, A. Valdes, P. G. Neumann, and T. D. Garvey. A real time intrusion detection expert system (ides). Technical report, SRI International, Menlo Park, California, 1992.

[Lun93] T.F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[MG96a] J. McDermott and D. Goldschlag. Storage jamming. In D.L. Spooner, S.A. Demurjian, and J.E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 365–381. Chapman & Hall, London, 1996.

[MG96b] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.

[MHL94] B. Mukherjee, L. T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.

[PG98] B. Panda and J. Giordano. Reconstructing the database after electronic attacks. In *Proceedings of the 12th IFIP 11.3 Working Conference on Database Security*, Greece, Italy, July 1998.

[PK92] P.A. Porras and R.A. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the 8th Annual Computer Security Applications Conference*, San Antonio, Texas, December 1992.

[RBKW94] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, 1994.

[SC98] R. Sandhu and F. Chen. The multilevel relational (mlr) data model. *ACM Transactions on Information and Systems Security*, 1(1), 1998.

[SG97] S.-P. Shieh and V.D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):661–667, 1997.

[SM97] D. Samfat and R. Molva. Idamn: An intrusion detection architecture for mibile networks. *IEEE Journal of Selected Areas in Communications*, 15(7):1373–1380, 1997.

[WSQ94] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, 1994.

A TPC-C Databases and Transactions

The benchmark portrays a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. As the Company's business expands, new warehouse and associated sales districts are created. Each regional warehouse covers 10 districts. Each district serves 3000 customers. All warehouse maintain stocks for the 100,000 items sold by the Company. Customers call the Company to place a new order or request the status of an existing order. Orders are composed of an average of 10 order items.

The database has nine tables whose structures are omitted here(See [Gra93] for the structures of these tables). The entity-relationship diagram of the database is shown in Figure 4 where all numbers shown illustrate the minimal database population requirements. The numbers in the entity blocks represent the cardinality of the tables. The numbers next to the relationship arrows represent the cardinality of the relationships.

B Read and Write Set Templates of TPC-C Transactions

In TPC-C, the term **database transaction** as used in the specification refers to a unit of work on the database with full ACID properties. A **business transaction** is composed of one or more database transactions. In TPC-C a total of five types of business transactions are used to model the processing of an order (See [Gra93] for the source codes of these transactions). The read and write Set templates of these transaction types are as follows.

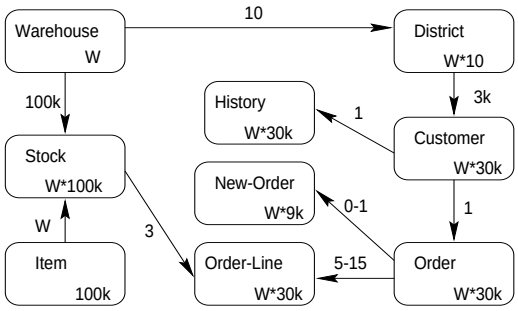


Figure 4. Entity-Relationship Diagram of the TPC-C Database

- The **New-Order** transaction consists of entering a complete order through a single database transaction. The template for this type of transaction is ('+' denotes string concatenation):

  Input= warehouse number(w_id), district number(d_id), customer number(c_id); a set of items(ol_i_id), supplying warehouses(ol_supply_w_id), and quantities(ol_quantity)
  Read_Set= { Warehouse.w_id.W_TAX; District.(w_id+d_id).(D_TAX, D_NEXT_O_ID); Customer.(w_id+d_id+c_id).(C_DISCOUNT, C_LAST, C_CREDIT); Item.ol_i_id.(I_PRICE, I_NAME, I_DATA); Stock.(ol_supply_w_id+ol_i_id).(S_QUANTITY, S_DIST_xx, S_DATA, S_YTD, S_ORDER_CNT, S_REMOTE_CNT) }
  Write_Set= { x=District.(w_id+d_id).D_NEXT_O_ID; New-Order.(w_id+d_id+x); Order.(w_id+d_id+x); R_1= { ol_i_id }; Order-Line.(w_id+d_id+x+R_1) }

- The **Payment** transaction updates the customer's balance, and the payment is reflected in the district's and warehouse's sales statistics, all within a single database transaction. The templates for this type of transaction are:

  Input= warehouse number(w_id), district number(d_id), customer number(c_w_id, c_d_id, c_id) or customer last name(c_last), and payment amount(h_amount)
  Read_Set= { Warehouse.w_id.(W_NAME, W_STREET_1, W_STREET_2, W_STATE, W_YTD); District.(w_id+d_id).(D_NAME, D_STREET_1, D_STREET_2, D_CITY, D_STATE, D_ZIP, D_YTD); **[Case 1,** the input is customer number: Customer.(c_w_id+c_d_id+c_id).(C_FIRST,

C_LAST,C_STREET_1, C_STREET_2,
C_CITY, C_STATE, C_ZIP, C_PHONE,
C_SINCE, C_CREDIT, C_CREDIT_LIM,
C_DISCOUNT, C_BALANCE,
C_YTD_PAYMENT,
C_PAYMENT_CNT, C_DATA);
Case 2, the input is customer last name:
Customer.(c_w_id+c_d_id+c_last).(C_FIRST,
C_LAST,C_STREET_1, C_STREET_2,
C_CITY, C_STATE, C_ZIP, C_PHONE,
C_SINCE, C_CREDIT, C_CREDIT_LIM,
C_DISCOUNT, C_BALANCE,
C_YTD_PAYMENT,
C_PAYMENT_CNT, C_DATA) **]** }
Write_Set= { Warehouse.w_id.W_YTD;
District.(w_id+d_id).D_YTD;
[Case 1, the input is customer number:
{ Customer.(c_w_id+c_d_id+c_id).(C_BALANCE,
C_YTD_PAYMENT, C_PAYMENT_CNT);
History.(c_id+c_d_id+c_w_id+d_id+w_id).* }
Case 2, the input is customer last name:
{ Customer.(c_w_id+c_d_id+c_last).(C_BALANCE,
C_YTD_PAYMENT, C_PAYMENT_CNT);
History.(c_d_id+c_w_id+d_id+w_id).* } **]** }

- The **Order-Status** transaction queries the status of a customer's most recent order within a single database transaction. The templates for this type of transaction are:

Input= customer number(w_id+d_id+c_id) or
customer last name(w_id+d_id+c_last)
Read_Set= { **[Case 1,** the input is customer number:
Customer.(w_id+d_id+c_id).(C_BALANCE,
C_FIRST, C_LAST, C_MIDDLE);
Case 2, the input is customer last name:
Customer.(w_id+d_id+c_last).(C_BALANCE,
C_FIRST, C_LAST, C_MIDDLE)**]** ;
x=Order.(w_id+d_id+c_id).O_ID;
Order.(w_id+d_id+c_id).(O_ENTRY_D,
O_CARRIER_ID);
Order-line.(w_id+d_id+x).(OL_I_ID,
OL_SUPPLY_W_ID, OL_QUANTITY,
OL_AMOUNT, OL_DELIVERY_D) }
Write_Set= { }

- The **Delivery** transaction processes ten new (not yet delivered) orders within one or more database transactions. The templates for this type of transaction are:

Input= warehouse number(w_id), district number(d_id),
and carrier number(o_carrier_id)
Read_Set= { R_1 = New-Order.(w_id+d_id).NO_O_ID;
R_2 = Order.(w_id+d_id+R_1).O_C_ID;
Order.(w_id+d_id+R_1).(O_CARRIER_ID,
OL_DELIVERTY_D, OL_AMOUNT);
Customer.(w_id+d_id+R_2).(C_BALANCE,
C_DELIVERY_CNT) }
Write_Set= {R_1 = New-Order.(w_id+d_id).NO_O_ID;
R_2 = Order.(w_id+d_id+x).O_C_ID;
Order.(w_id+d_id+R_1).O_CARRIER_ID;
Customer.(w_id+d_id+R_2).(C_BALANCE,
C_DELIVERY_CNT);
New-Order.(w_id+d_id+R_1);
Order-Line.(w_id+d_id+R_1).OL_DELIVERY_D }

- The **Stock-Level** transaction retrieves the stock level of the last 20 orders of a district. The templates for this type of transaction are:

Input= warehouse number(w_id), district number(d_id),
Read_Set = { x = District.(w_id+d_id).D_NEXT_O_ID;
$R_1 = \{x-1, ..., x-19, x-20\}$;
R_2 = Order-Line.(w_id+d_id+R_1+
OL_NUMBER).OL_I_ID;
Stock.(w_id+R_2).S_QUANTITY }
Write_Set= { }