TRACE: Zero-Down-Time Database Damage Tracking, Quarantine, and Cleansing with Negligible Run-Time Overhead

Kun Bai[1], Meng Yu[2], and Peng Liu[1]

[1] College of IST, The Pennsylvania State University,
University Park, PA 16802 USA
{kbai,pliu}@ist.psu.edu
[2] Department of Computer Science, Western Illinois University,
Macomb, IL 61455 USA
m-yu2@wiu.edu

Abstract. As Web services gain popularity in today's E-Business world, surviving DBMSs from an attack is becoming crucial because of the increasingly critical role that database servers are playing. Although a number of research projects have been done to tackle the emerging data corruption threats, existing mechanisms are still limited in meeting four highly desired requirements: *near-zero-run-time overhead*, *zero-system-down time*, *zero-blocking-time* for read-only transactions, *minimal-delay-time* for read-write transactions. In this paper, we propose *TRACE*, a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. TRACE consists of a family of new database damage tracking, quarantine, and cleansing techniques. We built TRACE into the kernel of PostgreSQL. Our experimental results demonstrated that TRACE is the first solution that can simultaneously satisfy the first two requirements aforementioned and the first solution that can satisfy all the four requirements.

1 Introduction

As Web services gain popularity and are embraced in industry to support today's E-Business world, surviving the back-end DBMSs from E-Crime is becoming even more crucial because of the increasingly critical role that they are playing. The cost of attacks on the DBMSs are often at the magnitude of several millions of dollars. Unfortunately, traditional DBMS protection mechanisms do not defend the DBMSs against some new threats that have come along with the rise of Internet, both from external source, e.g., SQL Slammer Worm [6], SQL Injection [19], as well as from malicious insiders. Existing DBMS security techniques, e.g., authentication based access control, integrity constraints, and roll-back recovery mechanisms, are designed to guarantee the correctness, integrity, and availability of the stored data, but are very limited in dealing with data corruption and do not deal with the problem of malicious transactions. As we will explain shortly in section 2, once a DBMS is attacked, the damage (data

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 161–176, 2008.

corruption) done by these malicious transactions has severe impact on it because not only is the data they write invalid (corrupted), but the data written by all transactions that read these data may likewise be invalid. In this way, legitimate transactions can accidentally spread the damage to other innocent data. Techniques, such as implemented in Oracle *flashback* [18] and [11], can handle corrupted data, but are costly to use and have serious impact on the compromised DBMSs. These techniques can seriously impair the database availability because not only the malicious transaction, but all the transactions committed after the malicious transaction are rolled back.

Although a good number of research projects have been done to tackle the emerging data corruption threats, existing mechanisms are still quite limited in meeting four highly desired requirements: (R1) *near-zero-run-time overhead*, (R2) *zero-system-down time*, (R3) *zero-blocking-time* for read-only transactions, (R4) *minimal-delay-time* for read-write transactions. As a result, these proposed approaches introduce two apparent issues: 1) substantial run time overhead, 2) long system outage. To see why existing data corruption tracking/recovering mechanisms are limited in satisfying the four requirements, here we briefly summarize some main limitations of three representative database damage tracking/recovering solutions [1][8][15]. In ITDB [1], a dynamic damage (data corruption) tracking approach is proposed to perform on-the-fly repair. However, it needs to log read operations to keep track of inter-transaction dependencies, which causes significant run time overhead. This method may initially mark some benign transactions as malicious thus preventing normal transactions access the data modified by them, and it can spread damage to other innocent data during the on-the-fly repair process. As a result, requirement R1 cannot be satisfied. In [8], an inter-transaction dependency graph is maintained at run time both to determine the exact extent of damage and to ease the repair process and increase the amount of legitimate work preserved during an attack. However, it does not support on-the-fly repair which results in substantial system outage. As a result, requirement R2 cannot be satisfied. In [15], another inter-transaction dependency tracking technique is proposed to identify and isolate ill-effects of the malicious transactions. In order to maintain the data dependency, this technique also needs to record a read log, which is not supported in existing DBMS and will pose a serious performance overhead . Additionally, it only provides off-line post-corruption database repair. Table 1 lists the major mechanisms and their limitations, and the four requirements will shortly be further explained in Section 2.

To overcome the above limitations, we propose *TRACE*, a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. The service outage is minimized by (a) cleaning up the compromised data on-the-fly, (b) using multiple versions to avoid blocking read-only transactions, and (c) doing damage assessment and damage cleansing concurrently to minimize delay time for read-write transactions. Moreover, TRACE uses a novel marking scheme to track causality without the need to log read operations. In this way, TRACE has near-zero run-time overhead. We build

Table 1. Existing Approaches and Their Limitations

Proposals	R_1	R_2	R_3	R_4
[1]		✓	✓	
[8]	✓			
[15]	✓			
TRACE	✓	✓	✓	✓

TRACE prototype into the DBMS kernel of PostgreSQL, which is currently the most advanced open-source DBMS with transaction support, not layered on top as ITDB [1]. In summary, TRACE is the *first* integrated database tracking, quarantine, and recovery solution that can satisfy all the four highly desired requirements shown in Table 1. In particular, TRACE is the first solution that can simultaneously satisfy requirements R1 and R2.

2 Preliminaries

2.1 The Threat Model

In this paper, we deal with data corruption problem in DBMS. Data corruption can be caused by a variety of ways. First, the attacks can be done *through web applications*. Among the *OWASP* [19] top ten most critical web application security vulnerabilities, three out of the top 6 vulnerabilities can directly enable the attacker to launch a malicious transaction, some of which are listed as follows: 1)*Injection Flaws*; 2)*Unvalidated Input*; 3)*Cross Site Scripting (XSS) Flaws*. Second, the attacks can be done *through insider attacks*. TRACE focuses on handling data corruption caused by Injection Flaws and insider attacks through transaction level attack. Transaction level attacks have been well studied in a good number of researches [2,8,22]. In this paper, we use "attack" to denote both the malicious attacks and human errors. We use a SQL injection attack to simulate the malicious transaction and evaluate our TRACE system in the experiments.

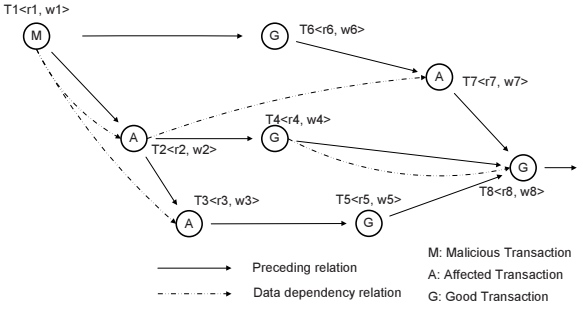


Fig. 1. An Example of Damage Spreading

2.2 Definitions

In this section, we formally describe the problems TRACE intends to solve. When a database is under an attack, TRACE needs to do: 1) identify corrupted data objects according to the damage causality information maintained at run time, and 2) carry out cleansing process to "clean up" the database on-the-fly. Here, the cleansing process includes damage tracking, quarantine, and repair. To accomplish the above tasks, TRACE relies on correctly analyzing some specific dependency relationships. We first define the following two *relations*.

Basic Preceding Relation: Given two transaction T_i and T_j, if transaction T_i is executed before T_j, then T_i precedes T_j, which we denote as $T_i \lhd T_j$. Note, we assume *strict 2PL* scheme is applied as in most of the commercial DBMSs.

Data Dependency Relation: Given any two transactions $T_i \lhd T_j$, if $(W_{T_i} - \bigcup_{T_i \lhd T_k \lhd T_j} W_{T_k}) \cap R_{T_j} \neq \emptyset$, then T_j is *dependent* on T_i, which is denoted as $T_i \rightarrow T_j$. We use R_T and W_T to denote the read set and the write set of transaction T. If there exist transactions $T_1, T_2, ..., T_n, n \geq 2$, that $T_1 \rightarrow T_2 \rightarrow$, ..., T_n, then we denote it as $T_1 \rightarrow^{\ltimes} T_n$.

For example, given the transaction $T_i : o^c = o^a + o^b$, $R_{T_i} = \{o^a, o^b\}$ and $W_{T_i} = \{\underline{o^c}\}$; the transaction T_j: $o^f = o^d + o^c$, $R_{T_j} = \{\underline{o^c}, o^d\}$, $W_{T_j} = \{o^f\}$. Obviously, we have $T_i \rightarrow T_j$ because transaction T_j reads the data object o^c written by transaction T_i. If data objects (e.g., o^c) contained in the write set W_{T_i} are corrupted, write set W_{T_j} is affected because of the dependency relation of T_i and T_j, $R_{T_j} \cap W_{T_i} = o^c$. Based on above statements, we have, as shown in Figure 1, $T_1 \rightarrow T_2$, $T_2 \lhd T_4$, $T_4 \rightarrow T_8$. If T_1 is a malicious transaction, transaction T_2, T_3 are affected directly (and T_7 is affected indirectly via T_2) and the data objects updated by T_2, T_3, T_7 are invalid. Transaction T_4, T_5, T_6, and T_8 are legitimate (good) transactions.

3 Overview of Approach

TRACE has two working modes, the *standby* mode and the *cleansing* mode. In this section, we overview our approach that enables TRACE system to meet the requirements listed in Table 1. We use "cleansing" rather than "recovering" throughout this paper to emphasize the additional feature of our approach to preserve legitimate data in the process of restoring the database back to the consistent status in the recent past. The goal of TRACE is to maintain maximum service online while quarantining, analyzing, and recovering the corrupted data objects stored in the DBMS.

3.1 The Standby Mode

In this working mode, TRACE uses a simple but efficient marking scheme to maintain the data dependency information at the run time. An ideal marking scheme should be transparent to the normal transaction process when there is

no malicious transactions detected. The marking idea is that TRACE constructs a *one-to-many* mapping from a set of tiny marks to the set of data objects. A *mark* is a unique n bits attached to a data object to indicate its *origin*. An origin is a data object's ancestor who creates/updates it. In this work, we set the mark at the record level and use a transaction ID as the mark of a data record. We maintain in *Causality Table* (CT) the inter-mark dependencies to perform the damage tracing (addressed in section 3.2). We create an entry for each created/updated data record in CT. A data record may have multiple *origins* because it can be updated a number of times in its life time. If a transaction T_i reads a database record that is created by transaction T_j, all database records updated by transaction T_i have T_j as one of their origins. If any origin of a data record has been identified as corrupted (or affected), all records that have the same origin are also believed as corrupted. Without blind writes, the origins will contain every mark (transaction ID) that has last updated the data object. However, during the normal transaction processing, the origins does not have a complete set of marks. This will be fulfilled in the damage assessment procedure.

3.2 The Cleansing Mode

In the cleansing mode, TRACE uses the causality information obtained in standby mode to identify and selectively repair only the data corrupted by the malicious/affected transactions on-the-fly. In the following, we overview how each cleansing operation functions with the focus on how they coordinate with one another.

⁕ **Damage Quarantine** is to prevent the normal transactions from accessing any invalid data objects, and then stop damage propagating and further reduce the repairing cost. When malicious transactions are detected by IDS, TRACE immediately sets up a time-based quarantine window (a time interval between the malicious transaction timestamp ts_b and the time TRACE receives the detection report). TRACE uses the timestamp ts_b to block the access to the data objects contained in the window. All access to the data records that are last updated later than the timestamp ts_b will be blocked. Access to the data records updated/created earlier than the timestamp ts_b will still be allowed. As a result, requirement R2 can be satisfied.

⁕ **Damage Assessment** is to identify every corrupted data within the quarantine window. When malicious transactions are detected, TRACE starts scanning the causality table from the first entry whose mark (transaction ID, tid) is the detected malicious transaction T_b, and then calculates the corrupted data set $C(T_i)$ up to the transaction T_i.

The abstract damage assessment algorithm includes two steps: 1) the Intrusion Detection System (IDS) reports a malicious transaction. The malicious transaction identifier (T_b) is the initial mark of damaged data records. During scanning the causality table, TRACE knows it has found all data records corrupted by the malicious transaction T_b when it encounters an entry whose mark (tid) has an associated timestamp later than the malicious transaction timestamp ts_b.

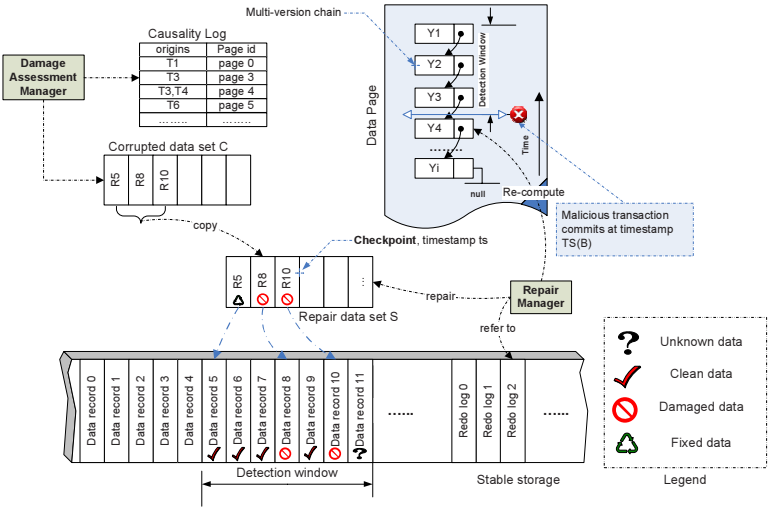


Fig. 2. Identify/Repair Corrupted Data Records Overview

At this point, TRACE obtains the initial corrupted data set $C(T_b)$. 2) Then, TRACE processes the causality table starting from the entries whose origins set O contains the mark T_b. In general, for each entry in CT associated with transaction T_j (mark tid_j), if the entry's origins set $O_{T_j} \cap C_{T_i} \neq \emptyset$ (C_{T_i} stands for the known corrupted transactions tids in C up to T_i), TRACE puts the data records with T_j (tid_j) into corrupted set $C(T_j)$, and then adds T_i's origins set O_{T_i} into T_j's origins set O_{T_j} in CT. TRACE stops the assessment process when any one of the following two conditions is true: 1) TRACE reaches the last entry of CT; 2) TRACE reaches an entry whose timestamp is equal to the time point when TRACE starts quarantine procedure. For condition 2, any entry in CT beyond this time point is valid because only transactions accessing to valid data are executed after the quarantine window is set up.

⁎ **Valid Data De-Quarantine** is to release the valid data objects in the quarantine window. In parallel to damage assessment, de-quarantine procedure executes to release data objects that either are over-contained valid data objects or corrupted data objects that have been repaired.

TRACE needs to gradually filters out real invalid data for repairing and release the over-quarantined valid data according to the following rules. When a data record is requested by a newly submitted transaction, 1) if the data record's timestamp is later than the malicious transaction timestamp ts_b and this data record is not included in repair set S_r, the access to it is denied. In this situation, whether the data record is invalid or valid is not known. The submitted transaction is put into active transaction queue to wait until the data record's status is clear (e.g. the data record 11 in Figure 2). 2) If the requested data record is in the repair set S_r and the status is invalid (e.g., the data record 8,

10 (R8, R10) in Figure 2), the access is not allowed. 3) If the data record is in S_r and the status is valid (e.g., the data record 5 (R5)), the data record has been fixed and is free to access. To guarantee the correctness of condition 1), we introduce a '*checkpoint*' to the repair set S_r. Each time TRACE copies the newly identified corrupted data records in the corrupted data set C to the repair set S_r, TRACE sets a 'checkpoint' in S_r. Among the data records in S_r, there is a data record whose timestamp ts_i is the greater than others, but smaller than the 'checkpoint'. If an incoming transaction requests a data record whose timestamp is smaller than ts_i and the data object is not included in the repair set S_r at this 'checkpoint', the data record is clean and is allowed to access. This is because TRACE ensures all corrupted data records before this 'checkpoint' have been identified and copied into S_r. After a corrupted data object is fixed, the data object's status is reset to clean.

⁎ **Repairing On-The-Fly** is to remove the ill-effects without stoping the DB services. A repairing transaction (undo) performs a task on corrupted data objects as if the malicious/affected transactions that result in invalid database states have never been executed. For instance, an undo (T_i) transaction is implemented as removing all specific version data objects written by transaction T_i as the transaction T_i never executes. To avoid the serialization violation, we must be aware that there exist some scheduled preceding relations between the undo transactions and the normal transactions. This is handled by submitting the undo transactions to the scheduler.

4 Design and Implementation of TRACE Atop PostgreSQL

To build the TRACE that offers the feature of identifying/repairing the corrupted data objects and meets the four requirements, we make several changes to the source code of a standard PostgreSQL 8.1 database [21].

4.1 Implementing the Marking Scheme

We maintain in *Causality Table* (CT) the inter-mark dependencies to perform the damage tracking. TRACE attaches the mark (several bytes, see Figure 3) to each data record. To construct the Causality Table (CT) entries for each corresponding data record, we need to modify the data structure of the data record defined in PostgreSQL and modify the *Executor Module*. The Executor executes a plan tree. When the tuples are returned, TRACE knows exactly what data records are accessed, updates the mark attached with each updated data record, and inserts the entry into the CT. To avoid insufficient precision and give each transaction a unique time, we extend the timestamp value with an additional four byte sequence number (SN). TRACE marking scheme additionally uses another eight byte transaction ID to indicate the transaction that last updates the data record and one bit in the record header to denote a data record dirty/clean status. Figure 3 gives the record layout with additional marking bytes.

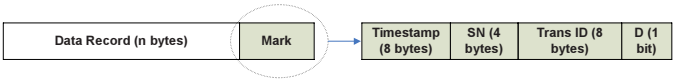
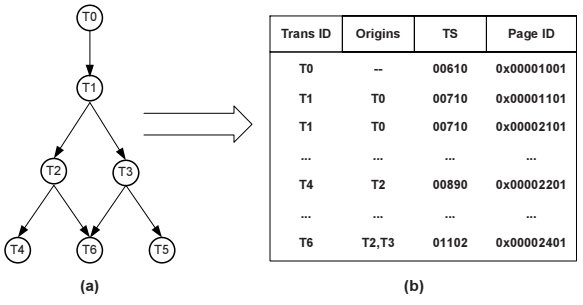


Fig. 3. Data Record Structure with Marking Bits



Trans ID	Origins	TS	Page ID
T0	--	00610	0x00001001
T1	T0	00710	0x00001101
T1	T0	00710	0x00002101
...	...	...	...
T4	T2	00890	0x00002201
...	...	...	...
T6	T2,T3	01102	0x00002401

Fig. 4. An Example of the Causality Table Construction

TRACE creates an entry in the CT for each updated data record. Figure 4(b) reflects the creation of marks as shown in Figure 4(a) using a simple example. Field *TransID* (the mark) has transaction ID (*xid* in PostgreSQL) as the key of the entry. Field *Origins* is a set of mark (xid_i) indicating a data record's origins. Field *TS* records the timestamp when the entry is created. We initially set the *timestamp* field with the transaction identifier, and replace it when the transaction commits. Field *PageID* indicates the page where TRACE can look up for the corresponding data record. In Figure 4(b), we assume transaction T_0 has no origin and calculates based on local inputs. Mark T_6 has origin T_2 and T_3. The complete *origins* list of a data record updated by transaction T_6 is $\{T_3, T_2, T_1, T_0\}$ and T_4 is $\{T_2, T_1, T_0\}$ according to Figure 4(a). If mark T_1 is identified as malicious, data records with attached mark T_1 are invalid. Since T_2 has T_1 in its origins, data records attached with mark T_2 are also invalid. Thus, mark T_6 is invalid too. Compared with the dynamic dependency analysis technique, which needs to dynamically check whether the write set of a transaction overlaps the read set of a transaction (such operations may consume a substantial amount of time), the marking scheme in TRACE does not need to perform any check operations, because all the information has been recorded in the CT.

4.2 Maintaining Before Images

In PostgreSQL, when a data record is updated the old versions usually are not removed immediately after the *update/delete* operations. A versioning system is implemented by keeping different versions of data records in the same tablespace (e.g. in the same *data page*). Each version of a data record has several hidden attributes, such as *Xmin* (the transaction id *xid* of the transaction that generates

the record version) and *Xmax* (the *xid* of the transaction that either generates a new version of this record or deletes the record). For example, when a transaction T_i performs an update on a data record o^x, it takes the valid version v of o^x, and makes a copy v^c of v. Then, it sets v's *Xmax* and v^c's *Xmin* to the transaction id xid_i. A data record is visible by a transaction if *Xmin* is valid and *Xmax* is not. Different versions of the same data record are chained by a hidden pointer as shown in Figure 2. TRACE utilizes these chained 'before' images to do the corrupted data record repairing (addressed in section 4.3). To make the hidden versions visible, we modify the index scan and sequential scan functions in *executor* module of PostgreSQL to identify the *xid* of transactions that generated (or deleted) data record versions which match our needs. Then we can go through the multi-version chain to find every historical version of a data record.

4.3 Damage Assessment Module

To assess the damage, TRACE locates the data record by the page id stored in CT. If the data page is not in memory, TRACE loads the data page containing the corrupted data record back into the memory. If more than one data record is in the same data page, the offset of the data record is used to locate the right data record. Then, TRACE traverses the associated data record version chain backwards in time (using the hidden previous version pointer of each on-page data record, and this could span over multiple data pages) to identify every invalid data record version and the valid data record version. Within the detection window, a data record can be updated multiple times. All updates corrupted both directly and transitively by the malicious transaction T_b must have timestamps associated with them later than the timestamp ts_b. As TRACE traverses the multi-version chain, it marks every invalid version by setting the *dirty* bit until it finds a data record version whose timestamp is less than the malicious timestamp ts_b. TRACE does not keep all invalid data record versions in the repair set S_r (implemented as a hash table with the transaction id as the hashed key). For example, in Figure 2, the data record Y_3 is invalid because of malicious transactions and then the version Y_2 and Y_1 are invalid. Thus, to identify corrupt versions of data records, TRACE needs merely keep the invalid version Y_1 in S_r.

4.4 Quarantine/De-quarantine Module

To implement the damage quarantine in PostgreSQL, we modify the executor module source code. The plan tree of PostgreSQL is created to have an optimal execution plan, which consists of a list of nodes as a pipeline. Normally, each time a node is called, it returns a data record. Starting from the root node, upper level nodes call the lower level nodes. Nodes at the bottom level perform either sequential scan or index scan. We make changes to the function of the bottom level nodes as well as the return results from the root node. By default, the executor module of PostgreSQL executes a sequential scan to open a relation, iterates over all data records. We change the executor module to check the

timestamp attached to each data record while scanning the data records in the quarantine phase. If a data record satisfies the query condition and its timestamp is later than the timestamp ts_b, the executor knows the incoming transaction requests a corrupted data record. Therefore, it either discards the return result from the root node or asks the damage assessment and de-quarantine modules for further investigation, and then puts the transaction to active transaction queue to wait.

In de-quarantine phase, we modify the executor function to check whether each scanned data record from the sequential scan is already in the repair set S_r or not. A similar change has been introduced for B-tree index scan nodes. During the normal database time, this procedure is transparent and bypassed without affecting performance. We maintain the repair set S_r as a mirror of the corrupted data set C is to enable damage assessment, de-quarantine and repairing modules run concurrently without the access conflict.

4.5 On-The-Fly Repairing

For each data record o^x in the repair set S_r, TRACE traverses backwards the hidden multi-version chain to the version whose timestamp ts_{o^x} is immediately earlier than the malicious transaction's timestamp ts_b (e.g., Y_4 in Figure 2). This version of data record is the correct '*before-image*' of the data record o^x. Only this version can be used to construct the undo transaction and eliminate the negative effects. To undo a damaged data record, repairing module simply restores the 'before-image' of this data record to its next version (e.g., restore version Y_4 to version Y_1 because Y_4 is Y_1's correct 'before-image', and then get rid of the version Y_3, Y_2, set the dirty mark of Y_1 to 0). This mechanism provides the TRACE system the ability to selectively restore a table or a set of data records to a specified time point in the past very quickly, easily and without taking any part of the database offline. One correctness concern with the on-the-fly repair scheme is whether it will compromise serializability. Due to the following reasons, TRACE will guarantee serializability: (a) all repairs are done within the quarantined area, so the repairs will not interfere the execution of new transactions; (b) our de-quarantine operations ensure serializability by doing atomic per-transitive-closure de-quarantine.

Causality Table is a disk table that has the format <*TransID*, *origins*, *timestamp*, *Page ID*>. We build a B-tree kind index ordered by *TransID* (transaction id *xid*) on top of the causality table and maintain in the main memory, which permits fast access to the related information to assess the damage. However, if we do not remove historical entries from the causality table, it intends to become very large and the index maintained in main memory accordingly become hideous. To keep the causality table relatively small, high performance, and without losing the track of cascading effect, we garbage collect the causality table entries which are no longer of an interest of the damage assessment. To garbage collect the causality table entries without missing the track of the cascading effects by malicious transactions, it will be safe to garbage collect those mark entries that stay in the causality table longer than a selected time

window because the probability that the mark entry is involved in a recent negative impact to the database is small enough. For the sake of the simplicity, in our experiments we assume that the detection latency is normally distributed with parameter $T = 3s$ (detection latency) and standard deviation $\sigma = 9s$, and we set the processing window $t = 100 \times T = 300$ seconds. Thus, we have a very small probability that garbage collection will harm the causality tracing once an attack is reported.

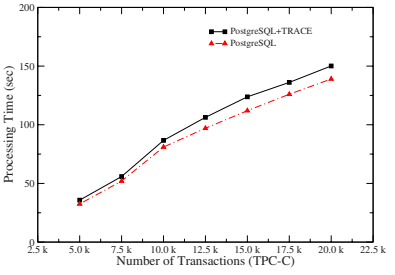
5 Experimental Results

We implement TRACE as a subsystem in PostgreSQL database system, and evaluate the performance of TRACE based on TPC-C benchmark and a clinic OLTP application. We present the experimental results based on the following evaluation metrics. First, motivated by requirement R1, we demonstrate the system overhead (i.e., the run time overhead) introduced by TRACE. Our experiments show the overhead is negligible. Second, motivated by requirements R2, R3 and R4, we demonstrate the comparison of TRACE and *'point-in-time'* (PIT) recovery method in terms of system performance, data availability.

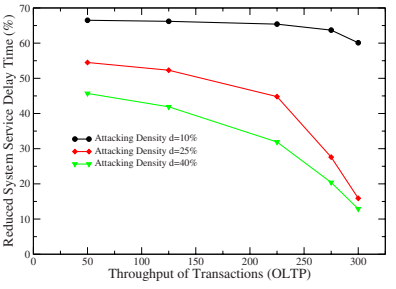
We construct two database applications based on the TPC-C benchmark and the clinic OLTP. The OLTP application defines 119 tables with over 1140 attributes belonging to 9 different sub-routines. In this work, we use 2 of the 9 sub-routines which contain 10 tables and over 100 attributes. For more detailed description of TPC-C benchmark and the OLTP application we refer the reader to [10,24]. Transaction workloads are based on above two applications. A transaction includes both read and write operations. The experiments conducted in this paper run on Debian GNU/Linux with Intel Core Due Processors 2400GHz, 1GB of RAM. We choose PostgreSQL 8.1 as the host database system and compile it with GCC 4.1.2. The TRACE subsystem is implemented using C.

5.1 System Overhead

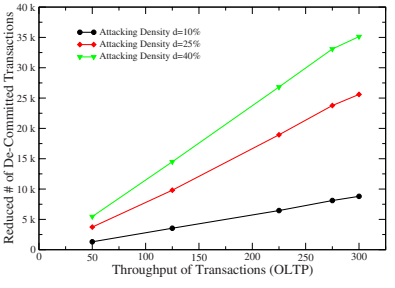
We evaluate the system overhead of transactions with *insert/update* statements. We create two applications with TPC-C benchmark and the clinic OLTP templates. Up to 20,000 transactions execute on each application. Among the 20K transactions, *20%* are *insert* statements, and the rest are update statements. For TPC-C application, we set up each transaction containing no larger than 5 *insert/update* statements. For OLTP application, we set up each transaction with at most 10 *insert/update* statements. Figure 5(a) shows the comparison of system overhead of TRACE and the raw PostgreSQL system on TPC-C. Because TRACE provides additional functionalities, it has system overhead on the PostgreSQL by the size of transaction in terms of the number of *insert/update* statements. The overhead introduced by TRACE comes from the following possible reasons: 1) for every *insert/update* operation, TRACE needs to create a CT entry and updates the timestamp field in CT. 2) To trace the invalid data records, TRACE maintains a causality table, which needs to allocate and access



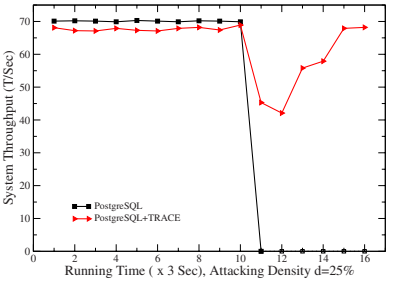
(a) System Overhead of TRACE on TPCC



(b) Reduced System Delay with d



(c) Saved Legitimate Transactions with d



(d) System Throughput of TRACE vs. PostgreSQL d=25%

Fig. 5. Evaluation of TRACE System

more disk storage when storing the causality information. For the TPC-C case of 20K transactions, we run the experiment 50 times and the average time of executing a transaction is in 7.1 ms. Additional 0.58 ms is added to each transaction (8% on average) to support causality tracking.

5.2 Reduced Service Delay Time and De-committed Transactions

We define the *service delay time* as the delay time experienced by a transaction T_i, denoted as $(t_n - t_m)^{T_i}$, where t_m is the time point the transaction T_i requests a data record, and t_n is the time point the transaction gets served. The average system outage time for n transactions is denoted as $\frac{\sum_{i=1}^{n}(t_n-t_m)^{T_i}}{n}$. For example, if the database system with PIT recovery needs $10s$ to restore and back to service, and during the time of recovery 100 transactions are submitted to the server, the average service delay for a transaction is $10s$. For the database with TRACE, the average service delay is $\frac{\sum_{i=1}^{100}(t_n-t_m)^{T_i}}{100}$. Then, the reduced service delay time is $10 - \frac{\sum_{i=1}^{100}(t_n-t_m)^{T_i}}{100}$ for this case. We define the attacking density $d = \frac{b}{t}$, where b and t are the throughput of malicious transactions within t and the total throughput of transactions, respectively. For example, if the total throughput of the system is 500 transaction per second, where there are 50 malicious/affected transactions per

second, the attacking density is 10%. We run each setting 300 seconds on the OLTP application to obtain stable results.

The experimental results are shown in Figure 5(b) and Figure 5(c). Figure 5(b) shows the reduced system service delay time w.r.t. different attacking density and throughput. We observe that, with TRACE component, the reduced system delay time is significant. The percentage of reduced service delay time decreases as the system throughput increases, and it decreases sharply (down to 15%) as the attacking density d increases. The reason is when the attacking density and throughput is light, TRACE spends less time to analyze the causality table and has much less corrupted data records to repair. As the d and throughput increase, TRACE causes the database system running busy in identifying the corrupted data records. However, even the percentage decreases, TRACE still saves a great amount of system outage time and makes the system stay online. Figure 5(c) shows the reduced de-committed transactions w.r.t. different attacking density and throughput. We observe that TRACE can save a large amount of innocent transactions, and then avoids re-submitting these transactions. This reduces the processing cost because the re-submitting process is very labor intensive and re-executing some of these transactions may generate different results than their original execution. During the 5 minutes experiment run, 150,000 transactions are executed on average. When the attack density is set at 10%, for example, there are 15,000 transactions are affected, the work of 10,000 legitimate transactions is saved, and around 80,000 records are cleaned with about 20% throughput degradation.

Figure 5(d) demonstrates the throughput performance of the PostgreSQL with/without TRACE based on the clinic OLTP application. To filter out potential damage spreading transactions, we assume the transaction dependence is tight. For instance, if a transaction T_x does not access compromised data but rely on the result of a transaction T_y, transaction T_x will still be filtered out (held in the active transaction queue) if transaction T_y is filtered out due to accessing compromised data because the result directly from transaction T_y is dirty. In Figure 5(d), we present an approximate 40 *sec* time window. On average the throughput of PostgreSQL is slightly higher than the PostgreSQL with TRACE because TRACE will add overhead into the system. At time point 11 (around 33 sec), a malicious transaction is identified. For traditional PostgreSQL system, the system stops providing service. For PostgreSQL with TRACE, the system will continue providing data access while starting the damage quarantine/assessment/repair procedure. In the worst time point, the throughput degradation ratio of TRACE is less than 40%, and the degradation ratio is quickly improved to 20% within 3 seconds. Overall, the goal of continuing providing service when the system is under an attack is met with satisfactory system throughput performance.

6 Related Work

Failure handling, e.g., [7,9,23], aims at guaranteeing the atomicity of database systems when some transactions failed. For instance, when a disk fails (media

failure), most traditional recovery mechanisms (media recovery) focus on recovering the legitimate state of the database to its most recent state upon system failure by applying backup load and redo recovery [5]. Unlike a media failure, a malicious attack cannot always be detected immediately. The damage caused by the malicious/erroneous transactions is pernicious because not only is the data they touch corrupted, but the data written by all transactions that read this data is invalid. Removing inconsistency induced by malicious transactions is usually based on the recovery mechanisms [17], in which a backup is restored, and a recovery log is used to roll back the current state. The usual database recovery techniques to deal with such corrupted data are costly to perform, introducing a long system outage while the backup is used to restore the database. Thus it can seriously impair database availability because not only the effects of the malicious transaction but all work done by the transactions committed later than the malicious transaction are unwound, e.g., their effects are removed from the resulting database state. These transactions then need to be re-submitted in some way (i.e. redo mechanisms) so as to reduce the impact onto the database. Checkpoint techniques [12] are widely used to preserve the integrity of data stored in databases by rolling back the whole database system to a specific time point. However, all work, done by both malicious and innocent transactions, will be lost.

Fault tolerant approaches [2] are introduced to survive and recover a database from attacks and system flaws. A color scheme for marking damage and a notion of integrity suitable for partially damaged databases are proposed to develop a mechanism by which databases under attack could still be safely used. For traditional database systems, *Data oriented attack recovery* mechanisms [20] recover compromised data by directly locating the most recent untouched version of each corrupted data, and *transaction oriented attack recovery* [14] mechanisms do attack recovery by identifying the transactions that are affected by the attack through read-write dependencies and rolls back those affected transactions. Some work on OS-level database survivability has recently received much attention. For instance, in [4], checksums are smartly used to detect data corruption. *Storage jamming* [16] is used to seed a database with dummy values, access to which indicates the presence of an intruder.

Attack recovery has different goals from media failure recovery, which focuses on malicious and affected transactions only. Previous work [1,2,3,13,14,20] of attack recovery heavily depends on exploiting the system log to find out the pattern of damage spreading and schedule repair transactions. The analysis of system log is very time consuming and hard to satisfy the performance requirement of online recovery. In addition, the dynamic algorithm proposed in [1] leaks damage to innocent data while repairing the damage on-the-fly. In [15], a similar idea of recovering from bad transactions is proposed to automatically identify and isolate the ill-effects of a flawed transaction, and then preserving much more of the current database state while reducing the service outage time. This technique requires both a write log and a read log. Although a write log is quite common in modern DBMS, maintaining a read log poses a serious performance overhead

and therefore is not supported in existing DBMS. In addition, it requires the system be off-line to mark the identified invalid data and repair them. In [8], an advanced dependency tracking technique is proposed. This approach tracks and maintains inter-transaction dependency at run time to determine the exact extent of damage caused by a malicious attack. The drawbacks of this approach are 1) it does not support on-line damage repair and thus results in long system outage, 2) it does not support *Redo* transaction and thus results in permanently data lost.

7 Conclusion and Future Work

We have dealt with the problem of malicious transactions that result in corrupted data. TRACE identifies the invalid data records and all subsequent data submitted by legitimated transactions affected by the malicious transactions directly or indirectly. Our marking scheme used in damage assessment enables us only de-commit the effects from affected transactions. Working with multi-version data records makes it unnecessary to restore a backup which is always online. Overall, our system removes far fewer transactions than the conventional recovery mechanisms and in turn provides a much shorter system service delay. In our immediate future work, we would develop a new cleansing mechanism that combines the attack recovery with conventional failure recovery of database systems.

Acknowledgement. This work was supported by NSF CNS-0716479, AFOSR MURI: Autonomic Recovery of Enterprise-wide Systems After Attack or Failure with Forward Correction, and AFRL award FA8750-08-C-0137.

References

1. Ammann, P., Jajodia, S., Liu, P.: Recovery from malicious transactions. IEEE Transaction on Knowledge and Data Engineering 14(5), 1167–1185 (2002)
2. Ammann, P., Jajodia, S., McCollum, C., Blaustein, B.: Surviving information warfare attacks on databases. In: The IEEE Symposium on Security and Privacy, Oakland, CA, May 1997, pp. 164–174 (1997)
3. Bai, K., Liu, P.: Towards database firewall: Mining the damage spreading patterns. In: 22^{nd} Annual Computer Security Applications Conference (ACSAC 2006), pp. 449–462 (2006)
4. Barbara, D., Goel, R., Jajodia, S.: Using checksums to detect data corruption. In: Int'l Conf. on Extending Data Base Technology (March 2000)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Publishing Company, Reading (1987)
6. CERT. Cert advisory ca-2003-04 ms-sql server worm (January 25, 2003), `http://www.cert.org/advisories/CA-2003-04.html`
7. Chen, Q., Dayal, U.: Failure handling for transaction hierarchies. In: Gray, A., Larson, P.-Å. (eds.) Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham, U.K, April 7-11, 1997, pp. 245–254. IEEE Computer Society, Los Alamitos (1997)

8. Chiueh, T., Pilania, D.: Design, implementation, and evaluation of an intrusion resilient database system. In: Proc. International Conference on Data Engineering, April 2005, pp. 1024–1035 (2005)
9. Eder, J., Liebhart, W.: Workflow recovery. In: Conference on Cooperative Information Systems, pp. 124–134 (1996)
10. TPC-C Benchmark, http://www.tpc.org/tpcc/
11. Lake, C.: Journal based recovery tool for ingres
12. Lin, J.-L., Dunham, M.H.: A survey of distributed database checkpointing. Distributed and Parallel Databases 5(3), 289–319 (1997)
13. Liu, P.: Architectures for intrusion tolerant database systems. In: The 18th Annual Computer Security Applications Conference, December 9-13, 2002, pp. 311–320 (2002)
14. Liu, P., Ammann, P., Jajodia, S.: Rewriting histories: Recovery from malicious transactions. Distributed and Parallel Databases 8(1), 7–40 (2000)
15. Lomet, D., Vagena, Z., Barga, R.: Recovery from "bad" user transactions. In: SIGMOD 2006: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 337–346. ACM Press, New York (2006)
16. McDermott, J., Goldschlag, D.: Towards a model of storage jamming. In: The IEEE Computer Security Foundations Workshop, Kenmare, Ireland, June 1996, pp. 176–185 (1996)
17. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17(1), 94–162 (1992)
18. ORACLE. Oracle database advanced application developer's guide (2007)
19. OWASP. Owasp top ten most critical web application security vulnerabilities (January 27, 2004), http://www.owasp.org/documentation/topten.html
20. Panda, B., Giordano, J.: Reconstructing the database after electronic attacks. In: The 12th IFIP 11.3 Working Conference on Database Security, Greece, Italy (July 1998)
21. Postgresql, http://www.postgresql.org/
22. Sobhan, R., Panda, B.: Reorganization of the database log for information warfare data recovery. In: Proceedings of the fifteenth annual working conference on Database and application security, Niagara, Ontario, Canada, July 15-18, 2001, pp. 121–134 (2001)
23. Tang, J., Hwang, S.-Y.: A scheme to specify and implement ad-hoc recovery in workflow systems. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 484–498. Springer, Heidelberg (1998)
24. Yao, Q., An, A., Huang, X.: Finding and analyzing database user sessions. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 851–862. Springer, Heidelberg (2005)