

# A Data Damage Tracking Quarantine and Recovery (DTQR) Scheme for Mission-Critical Database Systems

Kun Bai, Peng Liu

College of IST  
The Pennsylvania State University  
University Park, PA 16802  
kbai@ist.psu.edu

College of IST  
The Pennsylvania State University  
University Park, PA 16802  
pliu@ist.psu.edu

## ABSTRACT

Database security research aims to protect a database from unintended activities, such as authenticated misuse, malicious attacks. In recent years, surviving DBMS from an attack is becoming even more crucial because networks have become more open and the increasingly critical role that database servers are playing nowadays. Unlike the traditional database failure/attack recovery mechanisms, in this paper, we propose a light-weight dynamic Data Damage Tracking, Quarantine, and Recovery (*DTQR*) solution. We built the DTQR scheme into the kernel of PostgreSQL. We comprehensively study this approach from a few aspects (e.g., system overhead, impact of the intrusion detection system), and the experimental results demonstrated that our DTQR can sustain an excellent data service while healing the database server when it is under a malicious attack.

## 1. INTRODUCTION

Database Damage Management (DDM), especially transparent Database Data Damage Tracking, Quarantine and Recovery (DTQR), is an important problem faced by a great number of mission/life/business-critical applications. These applications are the cornerstones of a variety of crucial information systems (e.g., banking, online stock trading, and air traffic control, etc) that must manage risk, business continuity, and data assurance in the presence of severe cyber-attacks. Today, many of the nation's critical infrastructures (e.g., financial services, telecommunication infrastructure, transportation control) rely on these crucial information systems to function. Although computer security research has achieved a significant progress in protecting applications and systems, mission/life/business-critical applications still expose a large amount of vulnerabilities to the public, which eventually leads to severe malicious attacks. Furthermore, due to data sharing, interdependencies, and interoperability between business processes and applications (e.g., the emerging Web Services), the hit could greatly magnify its

*damage* by causing catastrophic cascading effects, which may “force” an application to shut down itself for hours or even days before the application is recovered from the hit. In addition, because not all intrusions can be prevented, DTQR is an indispensable part of the corresponding security solution, and the quality of DTQR scheme may have significant impact on risk management, business continuity, and data assurance. Hence, these mission/life/business critical applications highly demand a high quality transparent damage quarantine and recovery scheme.

Existing database security research primarily focuses on vulnerability assessments against the database (e.g., static and dynamic identification of vulnerability holes) and multi-layers of information security (e.g., access control, authentication, integrity constraints, etc). However, since vulnerability cannot be completely removed from a system, successful attacks often occur and cause damage to the database system. Authorization/authentication based database protection mechanisms are an effective means to providing safe data access, but are very limited in dealing with authenticated misuse, malicious attacks or inadvertent mistakes made by authorized individuals. In addition, such mechanisms cannot handel the data corruption problem and do not deal with the problem of malicious transactions. Conventional failure recovery mechanisms do not defend the DBMS against some new threats that have come along with the rise of Internet, both from external source, e.g., SQL Slammer Worm [7], SQL Injection [18], as well as from malicious insiders. As we will explain shortly in section 3, once a DBMS is attacked, the damage (data corruption) done by these malicious transactions has severe impact on the DBMS 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 (e.g., the damage spreading example shown in Figure 1).

There are several research projects conducted to tackle the emerging data corruption threats. However, they are still quite limited in meeting the following highly desired requirements: (R1) *near zero run time overhead*, (R2) *zero system down time*, (R3) *non-block* for read-only transactions, (R4) *minimal delay time* for read-write transactions. As a result, these proposed approaches introduce three apparent issues: 1) substantial run time overhead, 2) long system outage, 3) substantial legitimate work loss. To overcome the above limitations, we propose *TRACE*, a zero system down time database data damage tracking, quarantine, and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM. EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia. Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...\$5.00

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 tagging scheme to track causality without the need to log read operations. In this way, TRACE has near zero run time overhead. We build 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 transparent database tracking, quarantine, and recovery solution that can simultaneously satisfy all the four highly desired requirements shown above.

The rest of the paper is organized as follows. We describe in section 2 the related work. Section 3 describes some threats TRACE intends to handle and the problem statement. Section 4 overviews the key ideas of TRACE and introduces how we develop TRACE in PostgreSQL database system. Section 5 demonstrates the experimental results of our TRACE system in comparison with current recovery mechanisms. Finally, section 6 summarizes what we have done and mentions our future work.

## 2. RELATED WORK

Fault tolerant approaches are recently proposed to survive 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 in [2] to develop a mechanism by which databases under attack could still be safely used. For traditional database systems, *Data oriented attack recovery* mechanisms [19] 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.

There are fundamental differences between failure recovery and attack recovery. The problem of damage quarantine and recovery problem cannot be solved by failure recovery techniques which are mature in handling random failures. Failure recovery in general assumes the semantics of *fail-stop*. 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. Failure recovery intuitively assumes that all transactional operations have equal rights to be recovered. 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.

Previous work[1, 2, 3, 13, 14, 19] 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 on-line recovery. The dynamic algorithm proposed in [1] leaks damage to innocent data while repairing the damage on-the-fly. In addition, these existing DTQR mechanisms are limited in satisfying the four requirements mentioned in section 1. Here we briefly summarize some main limitations of three representative database DTQR 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.

## 3. PRELIMINARIES AND PROBLEM STATEMENT

### 3.1 The Threat Model

In this paper, we deal with the data corruption problem caused by transaction level attacks in database systems. Transaction level attacks are not new. They have been studied in a good number of researches [2, 8, 22]. Transaction level attacks can be done through a variety of ways:

- First, the attacks can be done through identity theft related fraudulent transactions. The fraudulent transaction is executed from within a valid user session, stronger user authentication on its own does not protect against these forms of attack.
- Second, the attacks can be done through erroneous

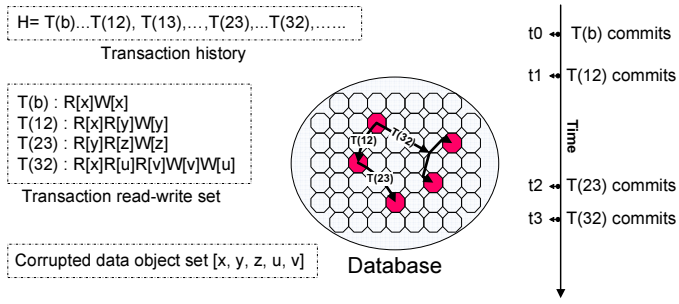


Figure 1: An Example of Damage Spreading

transactions issued by legitimate insiders due to mistakes. This kind of human errors can not be avoided completely because of the human nature.

- Third, the attacks can be done *through web applications*. Among the OWASP top ten most critical web application security vulnerabilities [18], five out of the top 6 vulnerabilities can directly enable the attacker to launch a malicious transaction, which can potentially corrupt the critical data stored in database. We list three top ranked vulnerabilities as follows.

1. *Unvalidated Input* - Information from web requests is not validated before being used by a web application. Attackers can use these flaws to attack backend components through a web application. Note that a major backend component is the database server, and a major way to attack a database server is to launch a malicious transaction.
2. *Cross Site Scripting (XSS) Flaws* - The attacker can first do a cross-site-scripting attack; then gain the user name and password of an account through the cookies he steals. Next, the attacker logs in an e-commerce site using the stolen user name and password; third, the attacker can issue a malicious transaction.
3. *Injection Flaws* - Through a SQL injection attack, the attacker can easily launch a malicious transaction.

Note, we use “attack” to denote both the malicious attacks and human errors.

## 3.2 Basic Concepts

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.

### 3.2.1 Database Model

A *database* system is a set of data objects, denoted as  $DB = \{o^1, o^2, \dots, o^n\}$ . A transaction  $T_i$  is a partial order with ordering relation  $\triangleleft_i$  [5], where

1.  $T_i \subseteq \{(r_i[o^x], w_i[o^x]) \mid o^x \text{ is a data object}\} \cup (a_i, c_i)$ ;
2. if  $r_i[o^x], w_i[o^x] \in T_i$ , then either  $r_i[o^x] \triangleleft_i w_i[o^x]$ , or  $w_i[o^x] \triangleleft_i r_i[o^x]$ ;
3.  $a_i \in T_i$  iff  $c_i \notin T_i$ .

and  $r, w, a, c$  relate to the operation of *read*, *write*, *abort*, and *commit*, respectively. The (usually concurrent) execution of a set of transactions is modeled by a structure called a history (system log). Formally, let  $T = \{T_1, T_2, \dots, T_n\}$  be a set of transactions. A complete history  $H$  over  $T$  is a partial order with ordering relation  $\triangleleft_H$ , where:

1.  $H = \cup_{i=1}^n T_i$ ;
2.  $\triangleleft_H \supseteq \cup_{i=1}^n \triangleleft_i$ .

### 3.2.2 Dependency Relations

To accomplish the two tasks, TRACE relies on correctly analyzing some specific dependency relationships. We first define the following two *relations*.

**DEFINITION 1. 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  $T_i \triangleleft T_j$ . Note, we assume strict 2PL scheme is applied as in most of the commercial DBMSs.

**DEFINITION 2. Data dependency relation:** Given any two transactions  $T_i \triangleleft T_j$ , if  $(W_{T_i} - \cup_{T_k \triangleleft T_i} W_{T_k}) \cap R_{T_j} \neq \emptyset$ , 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  $T_1, T_2, \dots, T_n, n \geq 2$ , that  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$ , then we denote it as  $T_1 \rightarrow^x T_n$ .

Now, we give the definition of two types of data corrupting transaction that are studied in this work.

**DEFINITION 3. Malicious transaction:** A transaction  $T$  whose write set  $W_T$  contains invalid data objects due to malicious intent or bad user inputs is a malicious transaction, denoted as  $T_b$ .

**DEFINITION 4. Affected transaction:** A transaction  $T_j$  that reads data objects updated by a malicious transaction  $T_b$  or by an affected transaction  $T_i$  that depends upon the updates of the malicious transaction  $T_b$ , formally  $R_{T_j} \cap W_{T_b} \neq \emptyset$  or  $R_{T_j} \cap W_{T_i} \neq \emptyset$ , is an affected transaction.

We denote a data object updated by a malicious transaction or an affected transaction as an *invalid* (corrupted) data object. Otherwise, it is a valid data object.

**EXAMPLE 1.** For example, given the transaction  $T_i : o^c = o^a + o^b, R_{T_i} = \{o^a, o^b\}$  and  $W_{T_i} = \{o^c\}$ ; the transaction  $T_j : o^f = o^d + o^c, R_{T_j} = \{o^c, o^d\}, W_{T_j} = \{o^f\}$ . 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  $T_j$ ,  $R_{T_j} \cap W_{T_i} = o^c$ .

If a transaction is malicious and makes the database state invalid, the entire effects of the transaction are invalid. Therefore, all updates to the data objects the transaction is writing are invalid. Based on above statements, we have, as shown in Figure 1,  $T_b \rightarrow T_{12} \rightarrow T_{23}, T_{12} \triangleleft T_{13}, T_b \rightarrow T_{32}$ . If  $T_b$  is a malicious transaction, transaction  $T_{12}$  and  $T_{32}$  are affected directly ( $T_{23}$  is affected indirectly via  $T_{12}$ ) and the data objects updated by  $T_{12}, T_{23},$  and  $T_{32}$  are invalid. Transaction  $T_{12}, T_{23},$  and  $T_{32}$  are legitimate (good) transactions.

### 3.3 Problem Statement

How to satisfy the four requirements listed in Section 1 without violating the following correctness criteria? To guarantee the correctness of TRACE, we define the correctness criteria.

DEFINITION 5. **Correctness Criteria:** When TRACE accomplishes its two tasks, the result is flawless if and only if the following conditions are satisfied:

1.  $\forall$  identified invalid data object  $o^x \in DB$  are restored to its latest pre-corruption state, which is a valid state;
2.  $\nexists$  invalid data object  $o^x$  is accessed by normal transactions (no damage leakage).

In particular, while ensuring the correctness, R1 requires that tracking data dependencies between transactions must not cause notable run time overhead. R2 indicates that the system (i.e., the database server) can never be stopped during damage tracking, quarantine, and repair. R3 indicates that read-only transactions (e.g., database queries) should not experience noticeable throughput degradation in the presence of data corruption attacks. R4 indicates that the damage propagation tracking operations, the damage quarantine operations, and the damage repair operations must be concurrently performed and synchronized in a fine-grained manner. In this way, the waiting time (or delay time) of read-write transactions may be minimized.

## 4. THE APPROACH

TRACE has two working modes, the *standby* mode and the *cleansing* mode. If no data corruption is reported by *Intrusion Detection System* (IDS), TRACE works in the standby mode and is invisible to the incoming transactions which are executed normally. If the IDS raises an alarm, TRACE will be activated and works in the cleansing mode to execute quarantine/assessment/cleansing procedures. Figure 2 illustrates the workflow of TRACE system. We use “cleansing” rather than “recovering” throughout this paper to emphasize the additional feature of our approach, which preserves the legitimate data in the process of restoring the database back to the consistent status in the recent past.

In the following sections, we overview our approach that enables TRACE system to meet the desired requirements. To build the TRACE that offers the feature of identifying/cleansing the corrupted data objects and meets the four requirements, we make several changes to the source code of the standard PostgreSQL 8.1 database [21].

### 4.1 Assumptions

When a DBMS is attacked, data corruption can be detected by a good number of existing intrusion detection (ID) techniques [6, 9, 11, 20, 23]. Although ID techniques and tools cannot do the damage tracking/quarantine/cleansing work, intrusion detection is a basic component of any intrusion recovery solution. In TRACE, we adopt the following work [6, 20] into our solution. All these techniques experience certain false positive rate and detection latency. Usually, the longer time the IDS spends, the more accuracy the IDS can achieve, however, the more damage the database has to suffer. Additionally, since false positives can mislead TRACE to revert updates done by innocent transactions, TRACE may need to get a data corruption

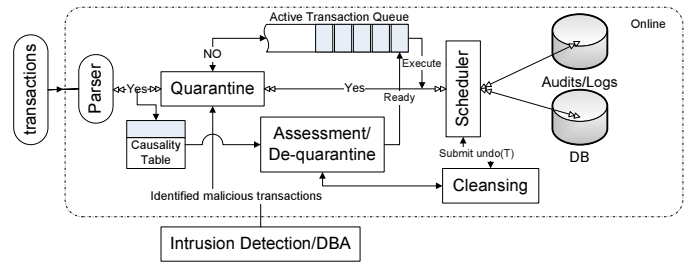


Figure 2: The TRACE System Workflow

TAG	ORIGINS	TS	PID
T(b)	----	00610	0x00001001
T(12)	T(b)	00710	0x00001101
...	...	...	...
T(23)	T(12),T(b)	00890	0x00002312
...	...	...	...
T(32)	T(b)	01102	0x00002401
T(32)	T(b)	01102	0x00002401

Figure 3: An Example of the Causality Table Construction

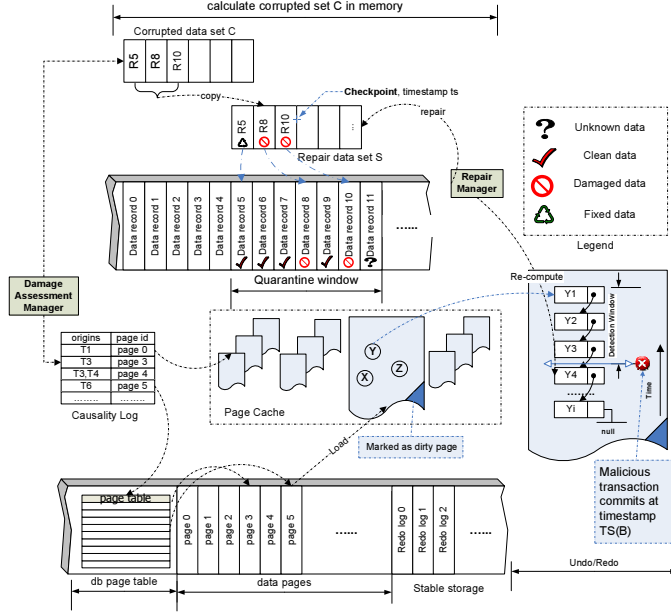
alarm verified to avoid unnecessarily cleansing any legitimate data records. Verifying a data corruption alarm is not always a difficult thing, e.g., by techniques such as storage jamming [16], checksum based corruption verification [4], or by a *Database Administrator* (DBA) who understands the applications being served by the database [15]. However, verification of a data corruption alarm can be time consuming. If the reporting delay is significant, more transactions in the system will be affected, which needs more efforts to make the cleansing done. We assume no blind write is performed in the database.

### 4.2 The Standby Mode

In the standby mode, TRACE uses a simple but efficient tagging scheme to transparently maintain the data dependency information at run time. The tagging idea is that we construct a *one-to-many* mapping from a tiny  $n$ -bits tag to a set of data objects. A *tag* is a unique  $n$  bits attached to a data object to indicate its *origin*. An origin is a data object’s ancestor indicating who creates/updates it. In this work, we set the tag at the record level and use a transaction id as the tag of a data record. We maintain in *Causality Table* (CT) (Figure 3) the inter-transaction dependencies to perform the damage tracing. Figure 3 shows the creation of Causality Table entries of the damage spreading example in Figure 1. In each CT entry, the field *TAG* has transaction id (*xid* is used in PostgreSQL) as the key of the entry. The field *ORIGINS* is a set of tags ( $xid_i$ ) indicating a data record’s origins. The field *TS* (timestamp) tells when the entry (data record) is created (updated). This field is filled when the corresponding transaction commits. The field *PID* (page id) tells where TRACE can look up for the corresponding data record (in memory or in stable storage).

TRACE creates an entry for each created/updated data record in CT. A data record may have multiple *origins* (a origin set) because it can be updated a number of times in its life time. The basic damage tracing idea is that if a trans-

action  $T_i$  reads a data record that is created/updated by transaction  $T_j$ , all data records updated by transaction  $T_i$  have tag  $T_j$  as one of their origins. If a tag has been identified as corrupted (or affected), all data records whose origin set has the tag are also believed as corrupted. Without blind writes, a data object's origin set will contain every tag that has last updated the data object. During the normal transaction processing, the origin set of each data object in CT does not have a complete origins. This will be fulfilled in the damage assessment procedure, namely *origin construction*.



**Figure 4: Identify/Repair Corrupted Data Records Overview**

In Figure 3, we assume that the malicious transaction  $T_b$  has no origin and calculates based on local inputs. The entry  $T_{12}$  has the tag  $T_b$  in its origin field. The complete *origins* set of transaction  $T_{23}$  is  $\{T_{12}, T_b\}$  according to the example in Figure 1. Since  $T_{12}$  and  $T_{23}$  have  $T_b$  in their ORIGINS, data records attached with tag  $T_{12}$  and  $T_{23}$  are also invalid. Similarly, entries tagged with  $T_{32}$  are invalid too. TRACE tagging scheme additionally uses eight byte timestamp to indicate when the transaction last updates the data record and one extra bit in the record header to denote a data record dirty/clean status. Thus, we need to modify the data structure of the data record defined in PostgreSQL. Each time a data record is updated by a transaction, the associated tag is accordingly updated.

Causality table process generally has three steps related to the *transaction begin*, *transaction in process*, *transaction commits*.

- *Transaction Begin*. A Causality table entry is generated for a data record when a transaction starts. Initially, the transaction identifier is assigned and entered into the *TAG* and *TS* fields.
- *Transaction In Process*. New versions of data records are generated during this period of time. When the transaction reads a database record  $o^x$ , it gets the transaction id that last updated the data record and

obtains the page id. We add both of them into the *ORIGINS* and *PID* in CT, respectively.

- *Transaction Commit*. At commit, we determine a timestamp for the transaction and store in the *TS* field in CT. We then revisit updated data records and perform a timestamping on each of the data records within the transaction.

### 4.3 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. If the components doing *damage quarantine*, *damage assessment*, *valid data de-quarantine* and *repairing on-the-fly* are not well coordinated, then substantial availability loss or deny-of-service can be experienced. In the following, we overview how each cleansing operation functions with the focus on how they coordinate with one another to minimize the availability loss.

#### 4.3.1 Damage Quarantine

Damage Quarantine is to prevent the normal transactions from accessing any invalid data objects, and then stop damage spreading and further reduce the repairing cost. When malicious transactions are detected by IDS, TRACE immediately sets up a time-based quarantine window  $qw$  (a time interval between  $ts_b$  and  $ts_d$ ). Here,  $ts_b$  indicates the time when the malicious transaction commits, and  $ts_d$  indicates the time when the data corruption is detected. TRACE blocks an incoming transaction  $T_i$  if any data object's timestamp  $ts_{o_i}$  in the read set  $R_{T_i}$  are contained in  $qw$  (shown in Algorithm 1). Access to the data objects updated/created earlier than the  $ts_b$  will still be allowed. In general, after a DBMS is attacked, majority data objects are still valid and available. As a result, requirement R2 can be satisfied.

#### Algorithm 1: Damage Quarantine Pseudo Code

```

Input:  $ts_b, ts_d, T_i$ 
1 begin
2    $qw_s \leftarrow ts_b; qw_e \leftarrow ts_d;$ 
3   while  $o_i \in R_{T_i}$  do
4     if  $ts_{o_i} \in [qw_s, qw_e]$  then
5        $\perp$  Deny the access of transaction  $T_i$ ;
6     else
7        $\perp$  Grant the access of transaction  $T_i$ ;
8 end

```

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.

### 4.3.2 Damage Assessment

Damage Assessment is to identify every corrupted data contained in the quarantine window  $qw$ . When malicious transactions are detected, TRACE starts scanning the causality table from the first entry whose tag (transaction id) 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 (Algorithm 2) includes two steps: 1) the IDS reports a malicious transaction. The malicious transaction identifier ( $T_b$ ) is the initial tag 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 tag ( $tid$ ) has an associated timestamp later than the malicious transaction timestamp  $ts_b$ . 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 tag  $T_b$ . In general, for each entry (a data record) in CT tagged with  $T_j$  (transaction id  $tid_j$ ), 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 tagged 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 (origin construction). TRACE stops the assessment process when any one of the following two conditions is true: a.) TRACE reaches the last entry of CT; b.) 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 request accessing to valid data are allowed to execute after the quarantine window is set up.

---

#### Algorithm 2: Damage Assessment Pseudo Code

---

```

Input:  $T_b, qw_e$ 
1 begin
   /*calculate the initial corrupted data object set
    $C(T_b)$  */
2 while  $o_i \in CT$  do
3   if  $(o_i.ts < ts_b) \ \&\& \ (o_i.mark == T_b)$  then
4      $o_i \rightarrow C(T_b)$ ;
5   else
6     break;
   /*calculate the corrupted data object set  $C(T_i)$  */
7 forall  $o_j \in CT$  do
8   if  $O_{T_j}^o \cap C_{T_i} \neq \emptyset$  then
9      $o_j \rightarrow C(T_j)$ ;
10    update  $O_{T_j}$  with  $O_{T_i}$ ;
11   if  $(\nexists o_{j+1} \in CT) \ || \ (ts_{o_{j+1}} \geq qw_e)$  then
12     break;
13 end

```

---

We now describe how we modify PostgreSQL to achieve damage assessment. We first review a feature of PostgreSQL, namely multi-versioning system. When a data record is up-

dated the old versions are not removed immediately. Instead, PostgreSQL implements a versioning system 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 4. TRACE uses these chained 'before' images to perform the calculation of the corrupted data record set and online repair (addressed in section 4.3.4). 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 generate (or delete) data record versions which match our needs. Then TRACE can go through the multi-version chain to find every historical version of a data record. In normal PostgreSQL operation, an *update/delete* of a data record does not immediately remove the old version of the data record. A data record may exist in multiple versions simultaneously. But eventually, when the disk space increasingly grows and outdated data records are no longer of interest of any transaction, the space they occupies must be reclaimed for reuse. PostgreSQL deals with it by *VACUUM* function. To support TRACE utilizing the multi-version data tuples, we set the PostgreSQL to non-VACUUM database system.

To assess the damage, TRACE locates the data record by the information stored in PID field in CT (for the details of page layout of PostgreSQL, we refer the reader to [21]). If a data page is not in memory, TRACE loads the data page containing the corrupted data record back into the memory (as shown in Figure 4). Then, TRACE traverses the associated data record version chain backwards in time to identify every invalid data record version and the valid data record version. 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  $ts_b$ . This version-data-status-identification scheme is a simple but effective solution to mark corrupted version data. We assume that all data versions of the corrupted data records in the quarantine window are suspicious. Thus, these versions cannot be trusted if there is no finer grained version-data-status-identification scheme provided. We will address a finer grained scheme in another paper. TRACE does not keep all invalid data record versions in the corrupted data set  $C(T_i)$  and the repair set  $S_r$  (on top of the  $S_r$ , we implemented a bloom filter to do the member checking, which will facilitate the de-quarantine procedure). For example, in Figure 4, 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  $S_r$ . The rest invalid data records will be discarded.

### 4.3.3 Release the Valid Data

This procedure is to release the valid data records con-

tained in the quarantine window  $qw$ . The abstract algorithm is listed in Algorithm 3. In parallel to damage assessment, de-quarantine procedure starts to function as soon as either over-contained valid data objects are identified or repaired invalid data objects are available.

TRACE needs to gradually filter out real invalid data for repairing and release 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 remains unknown. 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 4). 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 4), 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 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.

---

**Algorithm 3:** De-quarantine Valid Data Objects Pseudo Code

---

```

Input:  $T_i, ts_b$ 
1 begin
2   forall  $o_i \in T_i$  do
3     if ( $ts_{o_i} \geq ts_b$ )  $\&\&$  ( $o_i \notin S_r$ ) then
4        $\lfloor$  Deny the access of transaction  $T_i$ ; break;
5     else if ( $o_i \in S_r$ )  $\&\&$  ( $o_i.status == invalid$ )
6       then
7          $\lfloor$  Deny the access of transaction  $T_i$ ; break;
8     else if ( $o_i \in S_r$ )  $\&\&$  ( $o_i.status == valid$ ) then
9        $\lfloor$  continue;
10  Grant the access of transaction  $T_i$ ;
11 end

```

---

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.

#### 4.3.4 Repairing On-The-Fly

This module is to remove only the ill-effects without stopping the DBMS services. A repairing transaction  $undo(T_i)$  is implemented as removing all specific version data objects written by malicious/affected transaction  $T_i$  as the transaction  $T_i$  has never been executed. 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.

TRACE uses the multi-versioning system to look up 'before/after' images of damaged data objects without consulting the system log. Each data object  $o^x$  has multi-version records with the form  $\langle o^{x(v_1)}, o^{x(v_2)}, \dots, o^{x(v_n)} \rangle$ , where each  $v_i$ , ( $1 \leq i \leq n$ ) is a version number of the data record  $o^x$ . When TRACE identifies an invalid data record  $o^x$ , if the data record has been corrupted multiple times, TRACE will locate a correct version of the data record and performs the undo transaction only once to remove the invalid data record. A normal transaction that needs to read/write the data record  $o^{x(v_k)}$  must wait until the correct value of  $o^{x(v_k)}$  is restored by undo transaction. For a normal transaction that only needs to read the data record  $o^{x(v_k)}$ , multi-version data records break the dependency relations between the undo transactions and the normal transactions by providing an earlier valid version of the data object instead. Thus, it enables TRACE to execute the repairing and normal transactions concurrently and achieve minimal delay requirements.

---

**Algorithm 4:** On-the-fly Repair Pseudo Code

---

```

Input:  $T_i, ts_b$ 
1 begin
2   forall  $o_i \in S_r$  do
3     forall  $o_j \in T_j$  do
4       while  $o_j^{v_n}.ts \geq ts_b$  do
5          $\lfloor$   $o_j^{v_n}.dirty = 1$ ;
6          $\lfloor$   $n++$ ;
7          $\lfloor$   $undo(T_j) \leftarrow o_j^n$ ;
8        $\lfloor$  submit  $undo(T_j)$  to scheduler;
9 end

```

---

We now present how we implement the repairing method for the identified corrupted data records in PostgreSQL. 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 4). 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).

For an identified corrupted data record  $o^x$ , if TRACE notices that  $o^x$ 's timestamp is equal to the malicious transaction's timestamp  $ts_b$ , it knows that this corrupted data record is created by *Insert* operation by the malicious transaction. Then, TRACE removes the data record permanently from the database. If TRACE reads a "DEAD TUPLE"

mark attached on an invalid data record, TRACE knows the data record is removed by a *delete* operation by the malicious transaction. TRACE will unmark it and restores it with its right ‘before-image’. 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. TRACE guarantees serializability because of the following reasons: 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.

### 4.3.5 Garbage collection

Causality Table is a disk table that has the format  $\langle TAG, ORIGINS, TS, PID \rangle$ . We build a B-tree kind index ordered by *TAG* (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.

As we discussed in section 4, the Intrusion Detection System (IDS) has detection latency (or detection window), which has influence on when to collect the garbage. We therefore need to study the impact of the detection deficiencies. In this work, we assume that the detection delay is normally distributed with parameter  $T$  (detection latency) and the standard deviation  $\sigma$ . The expected value of the detection delay is:

$$E(x) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} e^{-(x-T)^2/2\sigma^2} dx = T \quad (1)$$

and the variation of the detection delay is:

$$Var(X) = E[(X - T)^2] = \sigma^2 \quad (2)$$

Thus, we have the cumulative distribution function

$$F(\alpha) = \Phi\left(\frac{\alpha - T}{\sigma}\right) \quad (3)$$

where  $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-y^2/2} dy$  is the cumulative distribution of a standard normal distribution. Hence, if there are no malicious transactions reported during a processing window  $t = k \times T$  and we only keep entries in the causality table within the time window  $t$ , the probability that the causality table does not contain an entry related to the malicious transactions is  $1 - F(t)$ . We define  $1 - F(t)$  as the missing probability. The missing probability does not necessarily indicate that we will miss processing anything but tells the probability that we need to scan the system logs to reconstruct the entries instead of obtaining them from the causality table directly. For example, given the detection delay is normally distributed with parameter  $T = 3s$  and  $\sigma^2 = 9$ . If we wait for  $t = 4 \times T = 12s$  and no malicious transaction has been reported, then the missing probability

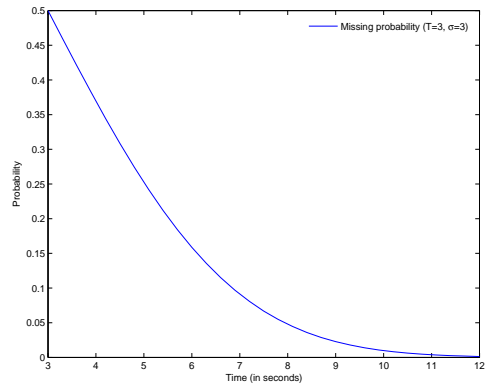


Figure 5: An Example of the relationship between the process window and the missing probability

is 0.0013 based on the above analysis (as shown in figure 5). According to the analysis, 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 marked entries that stay in the causality table longer than a selected  $kT$  because the probability that the marked entry is involved in a recent negative impact to the database is small enough. In practice, 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 [10]. We present the experimental results based on the following evaluation metrics. First, we demonstrate the system run time overhead imposed by TRACE, and the comparison of TRACE and peer work [15] on the run time overhead. Second, we demonstrate the comparison of TRACE and traditional ‘point-in-time’ (PIT) failure recovery method. Additionally, we show how the IDS detection latency and false positive rate impact TRACE.

We construct a database application based on the TPC-C benchmark. Transaction workloads are generated based on the application. For more detailed description of TPC-C benchmark, we refer the reader to [10]. 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 Run Time Overhead

We evaluate the system run time overhead of transactions with *update* statements because only when a data record is updated a tag is attached and generates a small piece of overhead. We use the application built up based on TPC-C benchmark. Up to 20,000 transactions execute on

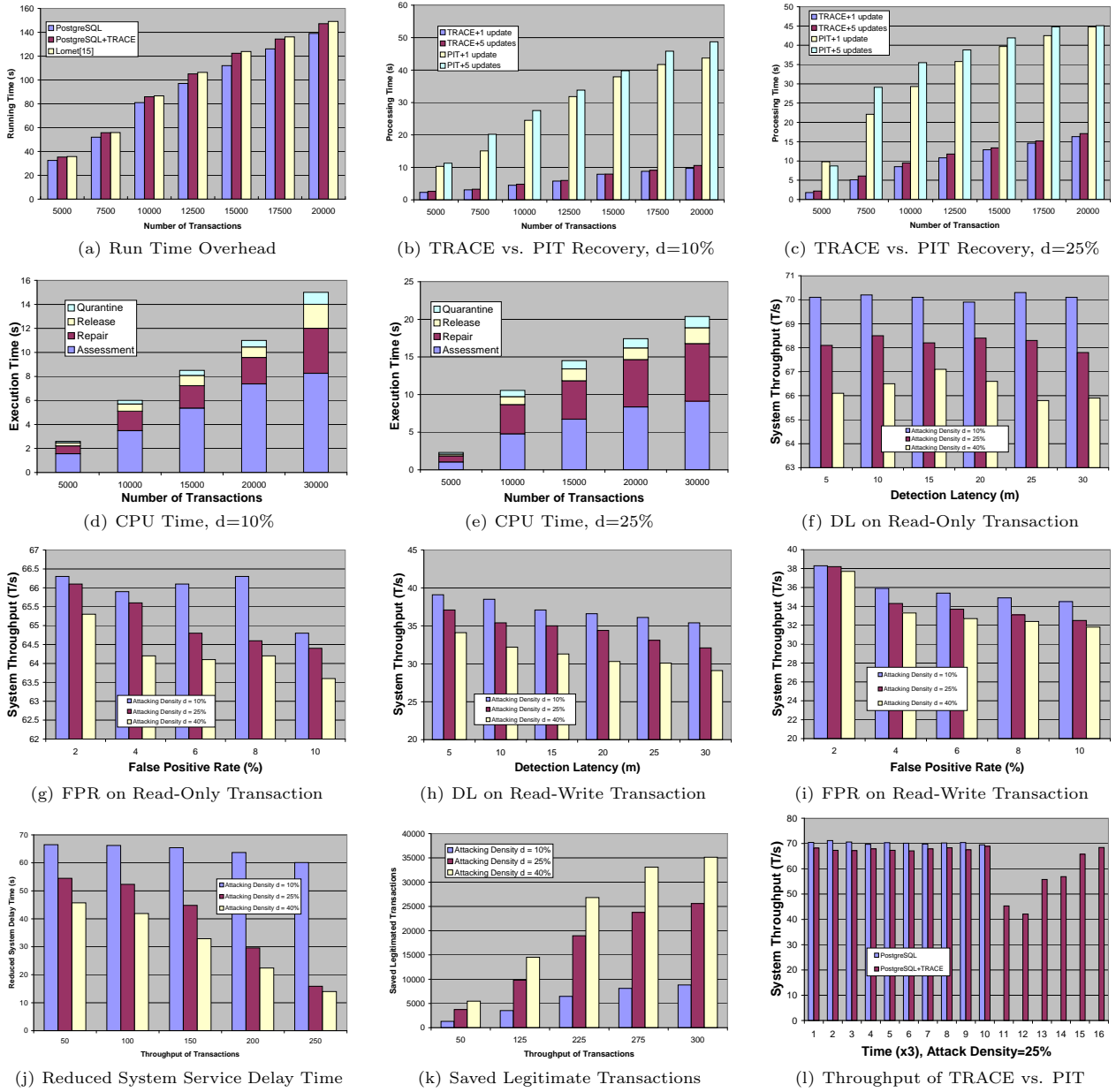


Figure 6: Evaluation of TRACE System

each application. For TPC-C application, we set up each transaction containing no more than 5 update statements. Figure 6(a) shows the comparison of system overhead of TRACE and the raw PostgreSQL system on TPC-C based application. Because TRACE provides additional functionalities, it has system overhead on the PostgreSQL by the size of transaction in terms of the number of 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 identify the invalid data records, TRACE maintains a causality table, which needs to allocate and access 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 7.1 ms. Additional 0.58 ms is added to each transaction (8% on average) to support causality tracking. We also implement the tagging method proposed in the work [15]. The results shown in the Fig-

ure 6(a) demonstrate that two methods are comparable and do not cause significant system overhead. In comparison with [1], which adds approximate 25%-35% run time overhead to the system, TRACE achieves a great improvement.

## 5.2 TRACE vs. ‘PIT’ on Cleansing Time

Next, we evaluate the performance of TRACE cleansing procedure and the ordinary PostgreSQL ‘point-in-time’ recovery procedure. We perform two experiments with 20,000 transactions having different ratios of corrupted data records within the database system, lightly damaged (15% of the total data records), heavily damaged (35% of the total data records).

We restrict the number of transactions to be equal for each experiment. For the first 1500 (3000) transactions of the light (heavy) damage experiment, they only contain *insert* statements. The rest of the transactions have *update* statements. Thus, each data record will be at least updated for multiple times. Each version of the data record

is kept in stable storage. To compare the TRACE recovery to the ordinary PostgreSQL ‘point-in-time’ recovery, we pre-process the database system when we apply PostgreSQL recovery because the typical procedure is to stop *postmaster* and execute *recovery.conf* file with appropriated settings. We choose *recovery\_target\_xid()* to indicate up to where the ‘point-in-time’ recovery procedure should perform. After this pre-processing, we re-start *postmaster* which will go into recovery mode and proceed to read through the archived WAL file it needs. Normally, PostgreSQL recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time or to some previous point in time. Upon completion of the recovery process, the *postmaster* will rename the recovery file *recovery.conf* to *recovery.done* to avoid unnecessary re-entry of recovery mode. Figure 6(b) and Figure 6(c) show the experimental results to indicate that TRACE uses much less time to restore the damaged database back to the consistent state in its recent past. Compared to the ‘point-in-time’, which installs a backup and unwinds all legitimated results since the time point, TRACE only removes the negative effects caused by malicious transactions and the affected transactions, and then de-commits much less transactions than the standard recovery procedure applied in PostgreSQL. Thus, TRACE recovery mechanism saves a great amount of system down time (up to 80% of the system down time is saved).

### 5.3 System CPU Time Distribution

To understand how well each key component of TRACE collaborate with others, we evaluate the TRACE components by studying the CPU time occupied by each component. We run the experiment 5 times with the attacking density  $d=10\%$  (25%), and 5000, 10000, 15000, 20000, and 30000 transactions, respectively. We demonstrate in Figure 6(d) the CPU time consumed by each of the key components when the database system is lightly damaged (15% of the total data records). We see that the damage assessment component approximately consumes 60% of the total execution time, and the damage repair component occupies 25% of the total execution time. The damage quarantine and the damage release components take a small piece of CPU time. When the database is lightly damaged, the workload of damage quarantine, repair, and release is relatively light. Most of the time is spent on the damage assessment because this component will identify the corrupted data (causality table analysis), access the storage (I/O operations), and flush the memory (loading data records). We demonstrate in Figure 6(e) the CPU time taken by each component when the database system is heavily damaged (35% of the total data records). We see that the damage assessment component still dominate the CPU time (approximate 50% of the total). The time spent by the damage repair component roughly increases by 10% because there are more damage data records to repair (submission of undo transaction). The results show that the damage assessment and damage repair components should be investigated more in order to achieve better performance in terms of consuming CPU time. We will study this in our future work.

### 5.4 Zero System Down Time

To evaluate the performance of TRACE on sustaining a good data service while recovering, we demonstrate in Figure 6(l) the system throughput of the PostgreSQL with/without

TRACE based on the TPC-C application. To filter out potential damage spreading transactions, we assume the transaction dependence is tight. For example, 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 6(l), we present an approximate 40 seconds system running-time window. Until the time point 11, the database system runs normally. During this partial time window, on average the throughput of PostgreSQL is slightly higher than the PostgreSQL with TRACE because TRACE will add system overhead into the system. At time point 11 (around 33 sec), a malicious transaction is identified. For traditional PostgreSQL system, the system shutdowns itself and stops providing service. For the PostgreSQL with TRACE, the system enables TRACE to carry out the damage quarantine/assessment/cleansing procedure. However, the database service is not harmed and the database system continues providing data access to new transactions while TRACE functions. During this partial time window (point 11 to point 16), the database armed with TRACE can still achieve near 57 T/s system throughput. 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. Hence, the requirement R2 is achieved.

### 5.5 Service Delay Time and Saved Legitimate 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 100 malicious transactions per second, the attacking density is 0.2. We run each setting 300 seconds on the TPC-C application to obtain stable results.

The experimental results are shown in Figure 6(j) and Figure 6(k). Figure 6(j) 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 6(k) 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.

## 5.6 IDS Impacts on TRACE System

When we evaluate the performance of TRACE on the requirement R3 and R4, we also consider the impact of the detection latency and the false positive rate of the IDS. To study the IDS impact on TRACE, we implement the system proposed in [11]. We demonstrate in Figure 6(f) and 6(h) the detection latency impact on TRACE, in Figure 6(g) and 6(i) the false positive rate impact on TRACE, respectively. Figure 6(f) shows the influence of the IDS detection latency on read-only transactions in terms of transaction throughput. It is clear that the read-only transaction throughput is not affected much as the detection latency increases. As long as the incoming transactions are read-only transactions, TRACE simply responses as the database normally operates if the data records requested by the transactions are not quarantined. If the data records requested by the transactions are quarantined, TRACE provides an old but valid version of the corresponding data record. Thus, the incoming read-only transactions are not blocked in the active transaction queue. As the damage rate increases along the X-axis of the Figure 6(f), we can see that read-only transactions also experience a little delay because of the following reasons: 1) TRACE consumes CPU time to do damage assessment and cleansing, 2) Detection latency causes more data records affected. Figure 6(g) shows the influence of false positive rate of IDS on processing read-only transactions. We set the false positive rate at 2%-10%. As the the false positive rate increases, the transaction throughput is approximately steady at the same level. Thus, the experimental results verify that the non-block for read-only transaction is satisfied.

Figure 6(h) shows the impact of the IDS detection latency on read-write transactions. The figure clearly shows that the read-write transactions are severely impacted by the IDS detection latency. As the detection latency increases along the X-axis, the system experiences a dramatic throughput downgrade. In addition, when the system is under the heavy data corruption circumstance (e.g., attack density  $d=25\%$ ), the system suffers even more throughput downgrade. This is because long detection latency imposes more corrupted data records on the database system due to the damage spreading. TRACE then needs to put more effort into the work of analyzing damage, repairing corrupted data records, and de-quarantining the fixed data records to the suspended transactions. Figure 6(i) presents the impact of the IDS false positive rate on read-write transactions. As expected, the false positive rate of IDS impairs the system throughput and causes delays to the newly incoming transactions as the detection latency does. As the IDS false positive rate increases, the delay to the incoming read-write transactions

ascends. This is because the false alarm from IDS enforces TRACE to act upon the alarm signal. TRACE then mistakenly quarantines legitimate work and rolls back the work it thinks as corrupted. Since TRACE has no pre-knowledge of whether the raised false alarm is true alarm or not, but only carries on what it has to do. Thus, TRACE causes the newly incoming read-write transactions to experience delays.

## 5.7 Discussion

The experimental results have shown that TRACE can achieve the goal of satisfying all four requirements. However, TRACE also has its limitations due to the impact of the false positive alerts of IDS. First, the false positive alerts can mistakenly force TRACE to act on innocent data objects and then cause undesired denial of service. Second, the false positive alerts can force TRACE to mistakenly repair uncorrupted data objects and then cause undesired performance degraded. As we mentioned in Section 4.1, the impact of false positive alerts can be effectively mitigated by data corruption verification techniques such as storage jamming and checkpoint based corruption verification. In fact, such negative impact is fundamentally caused by the need of *online* intrusion recovery instead of any specific recovery technique. *Offline* intrusion recovery, in which the database server is taken down until the repairs are done, will not suffer from such impact. However, the availability or business continuity loss caused by offline recovery can cost several magnitudes more than such impact.

## 6. CONCLUSION

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 the capability to achieve the aforementioned four requirements.

### 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.

## 7. REFERENCES

- [1] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transaction on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.
- [2] P. Ammann, S. Jajodia, C. McCollum, and B. Blaustein. Surviving information warfare attacks on databases. In *the IEEE Symposium on Security and Privacy*, pages 164–174, Oakland, CA, May 1997.
- [3] K. Bai and P. Liu. Towards database firewall: Mining the damage spreading patterns. In *22<sup>nd</sup> Annual Computer Security Applications Conference (ACSAC 2006)*, pages 449–462, 2006.

- [4] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Int'l Conf. on Extending Data Base Technology*, Mar 2000.
- [5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987. ISBN 0-201-10715-5.
- [6] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion detection in rbac-administered databases. In *ACSAC*, 2005.
- [7] CERT. Cert advisory ca-2003-04 ms-sql server worm. <http://www.cert.org/advisories/CA-2003-04.html>, January, 25 2003.
- [8] T. Chiueh and D. Paliana. Design, implementation, and evaluation of an intrusion resilient database system. In *Proc. International Conference on Data Engineering*, pages 1024–1035, April 2005.
- [9] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 59–68. ACM Press, New York, NY, USA, 2006.
- [10] <http://www.tpc.org/tpcc/>. *TPC-C Benchmark*.
- [11] S. Y. Lee, W. L. Low, and P. Y. Wong. Learning fingerprints for a database intrusion detection system. In *ESORICS*, 2002.
- [12] J.-L. Lin and M. H. Dunham. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.
- [13] P. Liu. Architectures for intrusion tolerant database systems. In *The 18th Annual Computer Security Applications Conference*, pages 311–320, 9-13 Dec. 2002.
- [14] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovery from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.
- [15] D. Lomet, Z. Vagena, and R. Barga. Recovery from "bad" user transactions. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 337–346, New York, NY, USA, 2006. ACM Press.
- [16] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.
- [17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. 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] OWASP. Owasp top ten most critical web application security vulnerabilities. <http://www.owasp.org/documentation/topten.html>, January, 27 2004.
- [19] B. Panda and J. Giordano. Reconstructing the database after electronic attacks. In *the 12th IFIP 11.3 Working Conference on Database Security*, Greece, Italy, July 1998.
- [20] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *ICDM*, pages 488–498, 2006.
- [21] PostgreSQL. <http://www.postgresql.org/>.
- [22] R. Sobhan and B. Panda. Reorganization of the database log for information warfare data recovery. In *Proceedings of the fifteenth annual working conference on Database and application security*, pages 121–134, Niagara, Ontario, Canada, July 15-18 2001.
- [23] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *Conference on Detection of Intrusions and Malware Vulnerability Assessment (DIMVA)*, pages 123–140, 2005.