Damage Quarantine and Recovery in Data Processing Systems

Peng Liu[1] Sushil Jajodia[2] Meng Yu[3]

[1]College of Information Sciences and Technology
Pennsylvania State University
University Park, PA 16802
pliu@ist.psu.edu
[2]Center for Secure Information Systems
George Mason University
Fairfax, VA 22030-4444
jajodia@gmu.edu
[3]Department of Computer Science
Monmouth University
West Long Branch, New Jersey 07764-1898
myu@monmouth.edu

Abstract

In this article, we address transparent Damage Quarantine and Recovery (DQR), a very important problem faced today by a large number of mission/life/business-critical applications and information systems that must manage risk, business continuity, and assurance in the presence of severe cyber attacks. Today, these critical applications still have a "good" chance to suffer from a big "hit" from attacks. Due to data sharing, interdependencies, and interoperability, the hit could greatly "amplify" its damage by causing catastrophic cascading effects, which may "force" an application to halt for hours or even days before the application is recovered. In this paper, we first do a thorough discussion on the limitations of traditional fault tolerance and failure recovery techniques in solving the DQR problem. Then we present a systematic review on how the DQR problem is being solved. Finally, we point out some remaining research issues in fully solving the DQR problem.

Key words: Damage Quarantine and Recovery, Transaction Processing, Data Integrity, Security

1 Introduction

In this article, we address transparent Damage Quarantine and Recovery (DQR), an important problem faced today by a large number of mission/life/business-critical applications. These applications are the cornerstones of a variety of crucial information systems 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 information systems to function.

There are at least two main reasons on why mission/life/business-critical applications have an urgent need for transparent damage quarantine and recovery. Firstly, despite that significant progress has been made in protecting applications and systems, mission/life/business-critical applications still have a "good" chance to suffer from a big "hit" from attacks. Due to data sharing, interdependencies, and interoperability between business processes and applications, the hit could greatly "amplify" 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. (Note that high speed Internet, e-commerce, and global economy have greatly increased the speed and scale of damage spreading.) The cascading damage and loss of business continuity (i.e., DoS) may yield too much risk. Because not all intrusions can be prevented, DQR is an indispensable part of the corresponding security solution, and a quality DQR scheme may generate significant impact on risk management, business continuity, and assurance.

Secondly, due to several fundamental differences between failure recovery and attack recovery, the DQR problem cannot be solved by failure recovery technologies which are very mature in handling random failures. (a) Failure recovery in general assumes the semantics of *fail-stop*, while attack recovery in general cannot assume the semantics of attack-stop, since to achieve the adversary's goal, most attacks (except for DoS) do not allow themselves to simply crash the system; they prefer hidden damage and alive zombies, spyware, bots, etc. Assuming fail-stop, *quarantine* is not really a problem for failure recovery; however, intrusion/damage quarantine is a challenging research topic in attack recovery and it can make a big difference. (b) Failure recovery assumes that all operations (e.g., transactions) have equal rights to be recovered, while attack recovery can never assume "equal rights" because neither malicious operations nor corrupted operations should be recovered.

Towards understanding and solving the DQR problem, the rest of the article is organized as follows. In Section 2, we present a comprehensive yet tangible description of the DQR problem. In Section 3, we do in-depth discussions on the limitations of traditional fault tolerance and failure recovery techniques in solving the DQR problem. In Section 4, we present a systematic review on how the DQR problem is being solved. In Section 5, we propose a set of remaining research issues in fully solving the DQR problem and conclude the paper.

2 Overview of the DQR Problem

We are concerned with the DQR needs of mission/life/business-critical information systems. Since those information systems have been designed, implemented, deployed, and upgraded over several decades, they run both *conventional* applications, which typically use proprietary user interfaces and application-level client-server protocols [Bir05], and *modern* applications, which are typically web-bounded running standard Web Services protocols.

Nevertheless, both conventional and modern mission/life/business-critical applications share some common characteristics: they are typically part of a large-scale, semantically rich, networked, interoperable information system; they are typically stateful and data-intensive; they are typically 24*7 applications requiring superb business continuity (i.e., availability); and they typically require guaranteed recoverability (and data integrity).

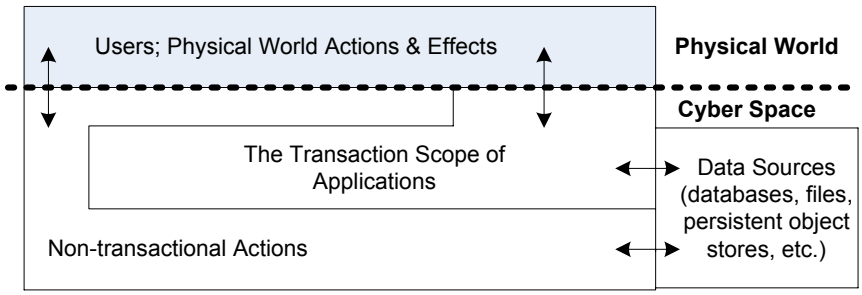

Figure 1: Transaction Level Scope of Applications

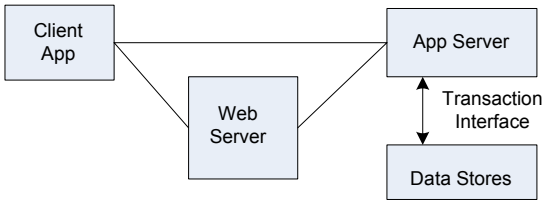

Figure 2: The Transaction Model in Concern

Although the DQR problem may be addressed at several abstraction levels (e.g., disk level, OS level, DBMS level, transaction level, application level), solving the DQR problem at the transaction level is particularly appealing due to the following reasons. The *transaction* abstraction has revolutionized the way *reliability*, including *recoverability*, is engineered for applications. Through a simple API interface provided by an easy-to-use transaction (processing) package which is today an integral part of mainstream application development environments such as J2EE and .NET, programmers can make applications *transactional* in a rather automatic, effort-free fashion. And the benefits of making applications transactional are significant: "failure atomicity simplifies the maintenance of invariants on data" [SDD85]; a guaranteed level of data consistency can be achieved without worrying about say race conditions; durability makes it much easier for programmers to get the luxury of recoverability.

As a result, the transaction mechanism is embraced by not only database systems [BHG87], but also a large variety of computer systems and applications [Gra93], including operating systems (e.g., VINO provides kernel transaction support [SESS96]), file systems (e.g., Camelot provides transactional access to user-developed data structures stored in files [SDD85]; and [LR04] argues that transactional file systems can be fast), distributed systems (e.g., QuickSilver uses transactions as a unified failure recovery mechanism [HMSC88]), persistent object stores (e.g., Augus supports transactions on abstract objects [LCJS87]), CORBA, and Web Services.

To leverage the strength, recovery facilities, and popularity of the transaction mechanism, and more important to make the proposed DQR solutions transparent to existing applications, it is a good idea to develop DQR theories and mechanisms at the transaction level. Since real world mission/life/business-critical applications typically deploy the transaction mechanism, transaction-level DQR solutions will have wide applicability.

2.1 Scope of Transaction Level DQR

In the rest of this paper, we will focus on transaction level DQR problems, models, and solutions. In particular, the transaction-level *scope* of an application and its *environment* are shown in Figure 1. In an information system, the transaction processing components of an application do not form an "isolated" system. Instead, these components will interact with their environment, which includes the Physical World, the various non-transactional actions, and the various types of data sources. Through these *interactions*, inputs are taken, physical world effects can be caused, and non-transactional attacking actions can "poison" the application's transaction scope. Although we are aware that the cyberspace damage and cascading effects can certainly cause damage in the physical world, this paper will focus on the *cyberspace* DQR solutions which will help minimize the damage caused in the physical world.

Based on how the transaction abstraction is implemented, different real world applications may deploy different transaction models. In this paper, we will focus on the transaction model shown in Figure 2. This model is widely used by conventional client-server applications and the well-known three-tier web applications. Applications running Transactional Web Services [Bir05] and cross-site "business transactions" (a.k.a. workflows) require more advanced transaction models, which are out of the scope of this paper. As we will mention shortly in Section 5, these advanced transaction models would introduce additional challenges in solving the DQR problem.

2.2 The Threat Model and Intrusion Detection Assumption

Working at the transaction level does *not* mean that malicious transactions are the only threat we can handle. Instead, as shown in Figure 1, we allow threats to come from either inside or outside of the transaction-level scope of applications. Nevertheless, to exploit the application's transaction mechanism to achieve a malicious goal, both inside and outside threats need to either directly *corrupt* certain data objects or get certain malicious transactions launched. Outside non-transactional attack actions (e.g., Witty worm) may bypass the transaction interface and corrupt some data objects via low-level (e.g., file or disk) operations. In addition, non-transactional buffer overflow attacks may break in certain running program of the application; then the attacker can manipulate the program to launch certain malicious transactions.

Inside the transaction scope, *insider attack* [Sch05] is probably the most serious threat. Since insiders (i.e., disgruntled employees of a bank) are typically not savvy in hacking, issuing malicious transactions (using a different user account) is typically the way they attack. Based on the study by [CK96], most (application level) attacks are from insiders. Besides insider attack, (a) *identity theft* may literally "transform" an outsider into an insider. (b) *SQL injection* attacks, though currently most used to steal sensitive information, has full capability to maliciously update data objects. (c) Five out of the top six web application vulnerabilities identified by OWASP [OWA04] may enable the attacker to launched a malicious transaction. They are *unvalidated input*, *broken access control*, *broken authentication and session management*, *cross site scripting* (which helps the attacker to steal user name and passwords), and *injection flaws*. (d) Finally, *erroneous* transactions caused by user/operator mistakes instead of attacks are yet another major threat to data integrity.

The intrusion detection assumption: We assume that a set of *external* intrusion detection sensors will do their job and tell us which operations (or transactions) were malicious or which

data objects were originally corrupted by the attack. These sensors may be a network-level (e.g., [Pax99]), host-level (e.g., [FHSL96]), database-level (e.g., [CGL00]) or transaction-level (e.g., [SFL97, BKTV05]) intrusion detection sensor. These sensors may enforce misuse detection (e.g., [Ilg93]), anomaly detection (e.g., [JV91, LX01]), or specification-based (e.g., [KRL97, SGF+02]) detection mechanisms. We assume these sensors are usually associated with false positives, false negatives, and detection latency. Finally, sensors that detect data corruption (e.g., [MG96, BGJ00, MVS00]) may also be used.

Remark Although some intrusion detection sensors could raise a good number of false positives or false negatives, the alarms raised by many intrusion/error detection sensors can actually be *verified* before any DQR operation is performed. (In this way, the negative impact of false positives and false negatives on the correctness/quality of DQR may be avoided.) For example, (a) most user/operator mistakes can be easily verified by the operation audit trails. (b) Many data corruption detectors have 100% accuracy. (c) When a strong correlation is found between one alert X and some other alerts, alert X may be verified as a true intrusion.

2.3 The DQR Problem/Solution Space

In our view, the DQR problem is a 6-dimensional problem:

- (1) The *damage propagation* dimension explains why cascading effects can be caused and why quarantine is needed. Although some specific types of damage (e.g., when an individual credit card account is corrupted) could be self-contained, a variety types of damage are actually very infectious due to data sharing, interdependencies, and interoperability between business processes and applications. For example, in a travel assistant Web Service, if a set of air tickets are reserved due to malicious transactions, some other travelers may have to change their travel plans in terms of which airlines to go, which nights to stay in hotel, etc.. Furthermore, the changed travel plans can cause cascading effects to yet another group of travelers; and the propagation may go on and on.

- (2) The *recovery* dimension covers three semantics for recovery: the *coldstart* semantics mean that the system is "halted" while damage is being assessed and repaired. (Damage assessment is to identify the set of corrupted data objects. Damage repairing is to restore the value of each corrupted data object to the latest before-infection version.) To address the DoS threat, recovery mechanisms with *warmstart* or *hotstart* semantics are needed. Warmstart semantics allow continuous, but degraded, running of the application while damage is being recovered. Hotstart semantics make recovery transparent to the users.

- (3) The *quarantine* dimension covers a spectrum of quarantine strategies: (a) coldstart recovery without quarantine, (b) warmstart recovery with conservative, reactive quarantine, (c) warmstart recovery with proactive or predictive quarantine, (b) hotstart recovery with optimistic quarantine, to name a few.

- (4) The *application* dimension covers the various transaction models deployed by conventional and modern applications. The uniqueness of each model may introduce new challenges for solving the DQR problem.

- (5) The *correctness* dimension tells whether a DQR scheme is *correct* in terms of consistency, recoverability, and quarantinability.

- (6) The *quality* dimension allows people to measure and compare the *quality levels* achieved by a set of correct yet different DQR schemes.

2.4 What Transaction Level DQR Solutions Cannot Do

First, although transaction-level DQR solutions will help minimize the damage caused by cyberspace attacks in the physical world, they cannot repair physical damage, which is a different field of study. Second, transaction-level DQR solutions are not designed to patch software which is another critical intrusion recovery problem. Nevertheless, transaction-level DQR solutions and software patching are complementary to each other. Transaction-level DQR solutions can help quarantine and repair the damage done by unpatched software broken-in by the adversary.

3 Traditional Failure Recovery Techniques and Their Limitations

DQR theories and mechanisms draw on work from several areas of systems research such as survivable computing, fault-tolerant computing, and transaction processing. Among all the relevant areas, the closest one should be Failure Recovery, which is part of Fault Tolerance [LA90]. In the literature, failure recovery has not only been extensively studied in data processing systems [BHG87, MHL$^+$92, Gra93], but also been thoroughly studied in other types of computing systems. In [BBG$^+$89] and [MBPR96], operating systems failure recovery is investigated. In [HMSC88], recovery management in distributed system is investigated. In [EAmWJ02], rollback recovery techniques for long-run applications are thoroughly discussed. In [LD97, LD01, Jef85, LL91], checkpoint-based rollback recovery is discussed. In [SS98], reliability modeling and evaluation criteria are thoroughly discussed. More recently, (a) David Patterson et al. have proposed the concept of ROC (Recovery -Oriented Computing) [PBB$^+$02] in which recovery is used as a general technique for dealing with failure in complex systems. For example, in [CF01] a model of "recursive recovery" is proposed in which a complex software system is decomposed into a multi-layer modular self-recovering implementation. (b) The Nooks approach [SBL03] makes device driver failures transparent to operating systems.

Unfortunately, due to the fundamental differences mentioned in Section 1 between failure recovery and attack recovery, existing failure recovery techniques cannot effectively deal with malicious attacks. For example, (a) rolling back the application's state to a previous corruption-free *checkpoint* will lose *all* the good work done after the checkpoint. (b) Maintaining frequent checkpoints [AJM95, MPL92, Pu86] may not work since no checkpoint taken between the time of attack and the time of recovery can be used. (c) Standy replica systems will not only replicate good work, but also replicate infection!

With DQR in *data processing systems* as the theme of this paper, this section will focus on failure recovery technologies for data processing systems and their limitations in solving the DQR problem. In the following, we classify failure recovery technologies for data processing systems into three categories: transactional undo/redo, replication-based recovery, and storage media backup-restore, and discuss them in three subsections, respectively.

3.1 Transactional Undo/Redo

The crux of transactional undo/redo techniques is correcting the application states that are corrupted due to failures. For data-processing systems or data-oriented applications in which doing read and write operations on various data objects (managed by a set of databases) represents the main activities, failure recovery is rooted in the *transaction concept* [GR93] which has been around for a long time. This concept encapsulates the *ACID* (Atomicity, Consistency, Isolation, and Durability) properties [BHG87, GR93]. Data-oriented applications are not limited to the database area [DLA02, DBSW89, GS89, HMSC88, LS83, NKK86]. The basic recovery procedure is almost the same for all applications: when a failure happens, a set of *undo* operations will be performed to rollback the application's *state* to the most recent *checkpoint*, which is maintained through logging, then a set of *redo* operations will be performed to restore the state to exactly the failing point. Nevertheless, the concrete recovery algorithms depend heavily upon how changes are logged. WAL (Write Ahead Logging) is today the standard approach widely accepted by the database industry. Some of the commercial systems and prototypes based on WAL are ARIES [MHL+92], IBM's AS/400 [CC89], IBM's DB2 [Cru84], Microsoft's SQL Server [sql], and Oracle's Oracle Database [ora].

Besides the basic idea of WAL, a set of important enhancements such as (a) using log sequence number (LSN) to correlate the state of a page with respect to logged updates of that page and (b) fuzzy checkpoints are proposed by ARIES [MHL+92], the de facto (industry) standard for transaction recovery models.

Finally, in addition to such standard recovery techniques as WAL, the database industry has developed various proprietary recovery tools. For example, DB2 Log Analysis Tool [db2a] allows you to monitor data changes; DB2 Recovery Expert [db2b] analyzes and provides diagnostics of altered database assets, and can roll data changes backward or forward; Oracle Recovery Manager [drd] manages the database backup and restore process; and Oracle Data Guard creates, maintains, manages and monitors one or more standby databases.

Limitations in Solving the DQR Problem: Although existing transaction recovery methods are matured in handling failures, they are not designed to deal with malicious attacks. In particular, first, the durability property ensures that traditional recovery mechanisms never undo committed transactions. However, the fact that a transaction commits does not guarantee that its effects are desirable. Specifically, a committed transaction may reflect inappropriate and/or malicious activity.

Second, although attack recovery is related to the notion of *cascading abort* [BHG87], cascading aborts only capture the *read-from* relation between active transactions, and in standard recovery approaches cascading aborts are avoided by requiring transactions to read only committed data [KLS90].

Third, there are two common approaches to handling the problem of undoing committed transactions: rollback and compensation. (3a) The rollback approach is simply to roll back all activity – desirable as well as undesirable – to a checkpoint believed to be free of damage. The rollback approach is effective, but expensive, in that all of the desirable work between the time of the checkpoint and the time of recovery is lost. Although there are algorithms for efficiently establishing snapshots on-the-fly [AJM95, MPL92, Pu86], maintaining frequent checkpoints may not work since no checkpoint taken between the time of attack and the time of recovery can be used. (3b) The compensation approach [GM83, GMS87] seeks to undo either

committed transactions or committed steps in long-duration or nested transactions [KLS90] without necessarily restoring the data state to appear as if the malicious transactions or steps had never been executed. There are two kinds of compensation: action-oriented and effect-oriented [KLS90, Lom92, WHBM90, WS92]. Action-oriented compensation for a transaction or step T_i compensates only the actions of T_i. Effect-oriented compensation for a transaction or step T_i compensates not only the actions of T_i, but also the actions that are affected by T_i. Although various types of compensation are possible, all of them require semantic knowledge of the application, and none of them is adopted by mainstream commercial systems.

Fourth, classic *redo* operations cannot repair damage because they do not reexecute affected transactions.

3.2 Replication-based Recovery

The crux of the replication based recovery is using redundancy to mask/tolerate failures. Replication-based recovery does not undo erroneous operations. In data-oriented applications, the replication idea is embodied through the widely adopted practice of data replication [GHOS96, BHG87] and *standby* databases [drd]. In such replicated systems, each request (or transaction) will be processed by all the *replicas* in which each data object is replicated. When a failure happens to the primary database, the responses (or outputs) generated by a standby (or replicated) database can be returned to the client as if the failure had never happened. (In distributed computing, the replication idea is embodied through such techniques as RAPS (reliable array-structured partitioned service), the state-machine approach [Sch90], and virtual synchrony [BC91].)

Limitations in Solving the DQR Problem: Both data replication and standy databases will not only replicate good work, but also replicate infection!

3.3 Storage Media Backup-Restore

The idea of storage media backup-restore is proven very practical and valuable. It is fully embraced by the IT industry: Computer Associates large enterprise backup solutions [ca], Symantec LiveState recovery products [sym], the Sonasoft Solution [son], just to name a few. This idea is complementary to the recovery idea and the replication idea, but in many cases it cannot achieve fine-grained data consistency, while the two other ideas can.

Limitations in Solving the DQR Problem: Among the data objects included in a *backup*, storage media backup-restore techniques cannot distinguish clean data objects from dirty, corrupted ones.

4 Solving the DQR Problem

In this section, we present a systematic review on how the DQR problem is being solved in the literature. Although self repairable file systems are proposed [ZC03, GPF+05], most DQR mechanisms proposed in the literature are transaction-level solutions. So here we concentrate on transaction-level DQR solutions.

4.1 The Model

In our model, a *transaction* is a set of *read* and *write* operations that either *commits* or *aborts*. For clarity, we assume there are no *blind* writes, although the theory can certainly be extended to handle blind writes. At the transaction level, an application (e.g., the application types shown in Figure 2) is a transaction execution *history*. Since recovery of uncommitted transactions is addressed by standard mechanisms [BHG87], we can safely ignore aborted transactions and only consider the committed *projection* $C(H)$ of every history H. We define $<_H$ to be the usual partial order on $C(H)$, namely, $T_i <_H T_j$ if $<_H$ orders operations of T_i before conflicting operations of T_j (Note that in H the operations of different transactions are often interleaved). Two operations *conflict* if they are on the same data object and one is write.

In principle, the *correctness* of a DQR scheme (or solution) can be "checked" either by the operations performed by the scheme or by the resulted effects. Here, we use the resulted history of a DQR scheme to study its correctness. In our model, the DQR histories resulted from a DQR scheme may contain the following information:

- A DQR history may contain two types of *malicious* transactions, four types of *legitimate* transactions, and one type of *cleaning* transactions: Type 1 malicious transactions are issued by attackers or malicious code; more broadly, transactions executed by mistake can be viewed as a Type 2 malicious transaction A legitimate transaction may be either a *regular* transaction or a *reexecuted* transaction; and both regular and reexecuted transactions may be *affected* or *damaged* if they read any corrupted data object. Finally, cleaning transactions only contain backward or forward overwrite operations, depending upon how the recovery is performed.
- A classic history consists of only operations, while a DQR history is an interleaved sequence of operations and data store states. The *data store* contains all the data objects that a transaction may access. The *state* of the data store at time t is determined by the latest committed values of every data object in the store.
- A data store state (e.g., a database state) contains three types of *corrupted* data objects and two types of *clean* data objects. Type 1 corrupted data objects are originally generated by the writes of malicious transactions. Type 2 are originally generated by affected transactions. Type 3 are originally generated by non-transactional attacking actions outside of the application's transaction scope. Note that a corrupted data object may be read or updated several times before it is *repaired* (a.k.a. *cleaned*). Type 1 clean data objects are never corrupted. Type 2 clean data objects are once corrupted, but they are repaired.

4.1.1 Damage Propagation

Based on the threat model, we know where malicious transactions come from. To see how affected transactions are generated and how the damage spreads, we should do dependency (or causality) analysis.

Definition 4.1 (dependency graph) As stated in [AJL02], transaction T_j is *dependent upon* T_i in a history if there exists a data object o such that T_j reads o after T_i has updated o; T_i does not abort before T_j reads o; and every transaction (if any) that updates o between the time T_i updates o and T_j reads o is aborted before T_j reads x. In a history, T_1 *affects* T_2 if the ordered pair $(T_1,\ T_2)$ is in the transitive closure of the *dependent upon* relation. Finally, we define the *dependency graph* for a (any) set of transactions S in a history as $DG(S) = (V, E)$

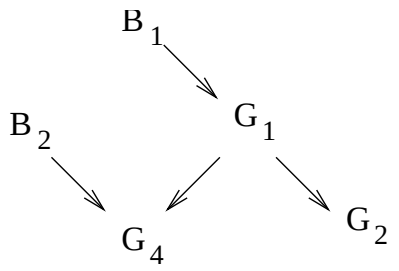

Figure 3: Dependency Graph for History H_1

in which V is the union of S and the set of transactions that are affected by S. There is an edge, $T_i \rightarrow T_j$, in E if $T_i \in V$, $T_j \in (V - S)$, and T_j is dependent upon T_i. □

Example Consider the following history over $(B_1, B_2, G_1, G_2, G_3, G_4)$:

$$H_1 : r_{B_1}[x] w_{B_1}[x] c_{B_1} r_{G_1}[x] w_{G_1}[x] r_{G_3}[z] w_{G_3}[z] c_{G_3} r_{G_1}[y] w_{G_1}[y] c_{G_1}$$
$$r_{G_2}[y] w_{G_2}[y] r_{B_2}[z] w_{B_2}[z] c_{B_2} r_{G_2}[v] w_{G_2}[v] c_{G_2} r_{G_4}[z] w_{G_4}[z] r_{G_4}[y] w_{G_4}[y] c_{G_4}$$

In H_1, B_1 and B_2 are malicious transactions while the other three are legitimate transactions; $r_T[x]$ ($w_T[x]$) is a read (write) operation by transaction T on data object x; c_T is the commit operation of T. Let $\mathbf{B} = \{B_1, B_2\}$, $DG(\mathbf{B})$ is shown in Figure 3.

Lemma 4.1 In a DQR history, a legitimate transaction is affected if and only if it is in dependency graph DG(all malicious transactions plus all the legitimate transactions that read the original version of any Type 3 corrupted data object). Being conservative, we assume all updates done by affected transactions may *spread* the damage. □

4.1.2 Do We Have to Sacrifice Durability?

A main concern people may have on DQR solutions is whether they will compromise *Durability*, a fundamental property of transaction processing and transactional failure recovery mechanisms. In other words, do we have to sacrifice Durability in doing DQR? Fortunately, the answer is NO. To keep durability, DQR schemes never really need to undo a malicious or affected transaction; instead, they can execute cleaning transactions to *semantically* revoke the effect of a committed transaction. By semantically revoking the effect of a committed transaction, we can achieve the following: (a) The effect of a committed transaction will always be kept durable; we never revoke or reverse any physical effect of a committed transaction on the persistent storage. (b) A cleaning transaction will change the data store state in exactly the *same* way as a regular transaction performing a set of updates. Because executing regular transactions will never compromise Durability, executing cleaning transactions (to do damage recovery) will never compromise Durability.

4.1.3 The Spectrum of DQR Schemes

The concept of DQR histories allows us to see the differences between the ones on the "spectrum" of DQR schemes. (a) On one end of the spectrum, a *static* DQR scheme will stop processing new transactions until every corrupted data object is *repaired* or cleaned. (A corrupted data object is *repaired* if its value is restored to the latest clean version before corruption.) So

since the *time of detection*, which is also the time when the recovery starts, the corresponding DQR history will proceed with only cleaning transactions until the repair is completed. In addition, affected transactions should be reexecuted; otherwise, DoS is caused. (b) On the other end, an optimistic, *dynamic* DQR scheme may do dependency analysis (a.k.a. damage assessment), execute cleaning transactions, execute to-be-reexecuted transactions, and execute new transactions *concurrently*. (c) Semi-dynamic DQR schemes may certainly stay on the spectrum between the two ends. For example, in [YLZ04, LVB06], there is a dedicated *scan* phase during which dependency analysis is performed, but *no* new transactions can be executed.

Section organization In the rest of this section, without losing generality, we will focus on the two "ends" of the spectrum of DQR schemes, that is, we will review static DQR solutions and dynamic DQR solutions in Section 4.2 and Section 4.3, respectively.

4.2 Static DQR Solutions

Static DQR solutions "halt" the database (service) before the repair is completed. Since no new transactions can be executed during static DQR, the damage will not spread unless there are incorrect repair operations. Hence, damage quarantine is not an issue in static DQR. As a result, static DQR has two aspects: *damage assessment*, which identifies every corrupted data object, and *damage repair*, which restores the value of each corrupted data object to its pre-corruption version.

In terms of how damage assessment and repair can be done, existing static DQR methods are either *data-oriented* [PG98] or *transaction-oriented* [AJL02]. Transaction-oriented methods assess and repair the damage by identifying and backing out affected transactions. In particular, they work as follows.

- *Damage Assessment* Build the dependency graph defined in Definition 4.1 for the set of malicious transactions detected. Based on Lemma 4.1, the dependency graph consists of all and only the affected transactions that have "contributed" to damage propagation. Assuming that read operations are logged together with write operations, it is not difficult to build the dependency graph. It is shown in [AJL02] that the log can be scanned forward only once (i.e., one-pass) from the entry where the first malicious transaction starts to locate every affected transaction.

- *Repair* When the damage assessment part is done, scan backward from the end of the log to semantically revoke (or undo) the effects of all the malicious transactions and the transactions included in the dependency graph. Note that here the undoes should be performed in the reverse commit order.

In contrast, data-oriented methods use the read and write operations of transactions to trace the damage spreading from one data object to another, and compose a specific piece of code to repair each damaged data object. In particular, data-oriented methods work as follows.

- *Damage Assessment* Construct a specific damage propagation graph in which each node is a (corrupted) data object while each directed edge from node x to y is a transaction T such that T reads x and writes y. The damage propagation graph can be built by one-pass scanning of the log.

- *Repair* Once the damage propagation graph is constructed, for each data object x contained in the graph, search through the log to find the latest pre-corruption version of x. Then repair x by overwriting the value of x with the searched version.

Comparison Data-oriented methods are more flexible and better at handling blind writes, however, composing cleaning code for each data object can be time consuming and prone to errors. Transaction-oriented methods use a cleaning transaction, which can be easily composed, to repair multiple data objects at the same time, thus they are more robust and efficient.

4.2.1 Maintaining Read Information

Both data-oriented methods and transaction-oriented methods rely on the read-from relationships between transactions. (Transaction T_1 *reads from* T_2 if there is a data object x such that T_1 reads x after T_2 updates x, and no other transaction updates x between these two operations.) However, the read-from information is not maintained by commercial DBMSes, since such information is not necessary for failure recovery. As a result, the transaction log maintained by a commercial DBMS actually does not contain sufficient information for the aforementioned DQR mechanisms to succeed. Therefore, maintaining the read-from information is an important task in engineering practical DQR systems.

In the literature, several representative techniques are proposed to maintain the read-from information. In [LJL+04], read operations are extracted from SQL statement texts. In particular, [LJL+04] assumes that each transaction belongs to a transaction *type*, and the *profile* (or source code) for each transaction type is known. For each transaction type ty_i, [LJL+04] extracts a *read set template* from ty_i's profile. The template specifies the kind of objects that transactions of type ty_i could read. Later on when a transaction T_i is executed, the template for $type(T_i)$ will be *materialized* to produce the read set of T_i using the input arguments of T_i (Note that these input arguments are embedded in T_i's SQL statements). This method is transparent to the DBMS kernel, however, in some scenarios it can only obtain approximate read sets.

In [PcC05], the DBMS is extended to provide support for read triggers. In contrast, commercial DBMSes only support insert/update triggers. This method can obtain the exact read sets and it has reasonable run-time overhead, but it requires a major extension to the kernel.

In [LVB06], a more aggressive approach is taken to maintain the read-from information. In this approach, Recovery Manager, the "core" of commercial transaction management systems, is modified to log reads. In particular, when the system commits a transaction, all the read information about the transaction will be consolidated into a single log record; then this special reads-keeping log record will be forced onto the disk together with other writes-keeping log records. This approach has minimal run-time overhead, but it requires the largest amount of changes to the DBMS kernel.

4.2.2 Static Repair via History Rewriting

From the correctness point of view, both data-oriented methods and transaction-oriented methods would result in a history that is *conflict equivalent* to the *serial* history composed of only the legitimate, unaffected transactions. ($C(H_1)$ is conflict equivalent to $C(H_2)$ if they contain the same set of operations and they order every pair of conflicting operations in the same way.)

Nevertheless, the history rewriting framework proposed in [LAJ00] shows that if we relax the correctness requirement from conflict equivalence to *view equivalence*, we may even save the work of affected transactions.

In particular, by exploiting two new semantic relationships between transactions, denoted *can-follow* and *can-precede*, respectively, the history rewriting framework can rewrite every "infected" history, which always starts with a malicious transaction, to a ready-to-repair history in which every legitimate, unaffected transaction precedes all the malicious transactions. Such a rewritten history typically looks like the following. Here, G_i is a legitimate, unaffected transaction and AG_i is an affected transaction. In addition, F_i is called a *fix*. A fix for a transaction like B_1 is a set of variables read by the transaction given values as in the original position of the transaction before the history is rewritten.

$$G_{i1}...AG_{j1}...G_{in}...AG_{jm}\ B_1^{F_1}\ AG_{k1}^{F_{k1}}...B_l^{F_l}...AG_{kp}^{F_{kp}}$$

The study in [LAJ00] shows that (a) each rewritten history and the original history will result in the same final database state, and (b) the work of all the legitimate transactions preceding $B_1^{F_1}$ in the rewritten history can be saved by executing a specific compensating transaction for each of the transactions in the *suffix* of the rewritten history. The suffix starts with $B_1^{F_1}$. Note that the last transaction in the rewritten history should be the first one to compensate, and $B_1^{F_1}$ should be the last one. Since every legitimate, unaffected transaction will precede $B_1^{F_1}$, the work of all unaffected transactions will be kept. Moreover, since affected transactions may precede $B_1^{F_1}$, the work of many affected transactions may be saved as well.

4.3 Dynamic DQR Solutions

In static DQR, new transactions are blocked during the repair process. This prevents static DQR mechanisms from being deployed by 24*7 database applications. As 24*7 database applications are becoming more and more common, dynamic DQR solutions that can do non-stop, zero down-time attack recovery are in demand.

4.3.1 Dynamic DQR Solutions with Reactive Quarantine

To have zero down-time, neither damage assessment nor repair can block the execution of new transactions. This means that dependency analysis, execution of new transactions, execution of cleaning transactions, and reexecution of affected transactions need to be done in parallel. To meet this challenge, people may wonder if the traditional transaction management architecture needs to be rebuilt. Fortunately, Figure 4 shows that the traditional transaction management architecture [BHG87] is adequate to accommodate on-the-fly repair. The *Repair Manager* is applied to the growing logs of on-the-fly histories to mark any bad as well as affected transactions. For every bad or affected transaction, the Repair Manager builds a cleaning transaction and submits it to the *Scheduler*. The cleaning transaction is only composed of write operations. The *Scheduler* schedules the operations submitted either by user transactions or by cleaning transactions to generate a correct on-the-fly history. Affected transactions that are semantically revoked (or undone) can be resubmitted to the Scheduler either by users or by the Repair Manager. Finally, the *Recovery Manager* executes the operations submitted by the Scheduler and logs them.

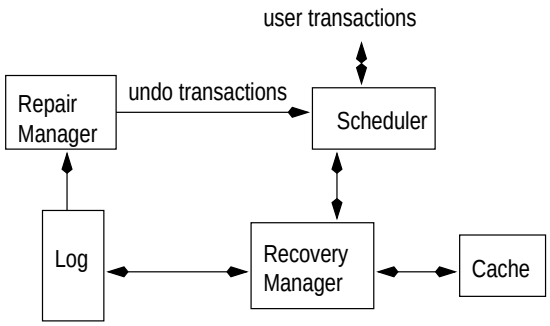

Figure 4: Architecture of an On-the-fly Repair System

On-the-fly attack recovery faces several unique challenges. First, since new transactions may first read corrupted data objects then update clean data objects, the damage may continuously spread, and the attack recovery process may never *terminate*. Accordingly, we face two critical questions. (a) Will the attack recovery process terminate? (b) If the attack recovery process terminates, can we detect the termination? Second, we need to do repair *forwardly* since the assessment process may never stop. The assessment process may never stop since the damage may continuously spread. Third, cleaned data objects could be re-damaged during attack recovery.

To tackle challenge 2, we must ensure that a later on cleaning transaction will not accidentally damage an object cleaned by a previous cleaning transaction. For this purpose, the system should "remember" the data objects that are already repaired and not yet re-damaged. To tackle challenge 3, we must not mistake a cleaned object as damaged, and we must not mistake a re-damaged object as already cleaned. To tackle challenge 1, the study in [AJL02] shows that when the damage spreading speed is quicker than the repair speed, the repair may never terminate. Otherwise, the repair process will terminate, and under the following three conditions we can ensure that the repair terminates: (1) every malicious transaction is cleaned; (2) every identified damaged object is cleaned; (3) further damage assessment scans will not identify any new damage (if no new attack comes).

From a state-transition angle, the job of attack recovery is to get a *state* of the database, which is determined by the values of the data objects, where (a) no effects of the malicious transactions are there and (b) the work of good transactions should be retained as much as possible. In particular, transactions transform the database from one state to another. Good transactions transform a good database state to another good state, but malicious transactions can transform a good state to a damaged one. Moreover, both malicious and affected (good) transactions can make an already damaged state even worse. We say a database state S_1 is *better* than another one S_2 if S_1 has fewer corrupted objects. The goal of on-the-fly attack recovery is to get the state better and better, although during the repair process new attacks and damage spreading could (temporarily) make the state even worse. (A state-oriented object-by-object attack recovery scheme is proposed in [PG98].)

Finally, it should be noticed that from the transaction scheduling viewpoint, on-the-fly repair introduces new scheduling constraints. For example, (a) when a read operation $r_T[x]$ is scheduled, x must be clean. (b) Conflicting cleaning transactions should be scheduled in the same order in which they are submitted by the Repair Manager. The order is critical to the

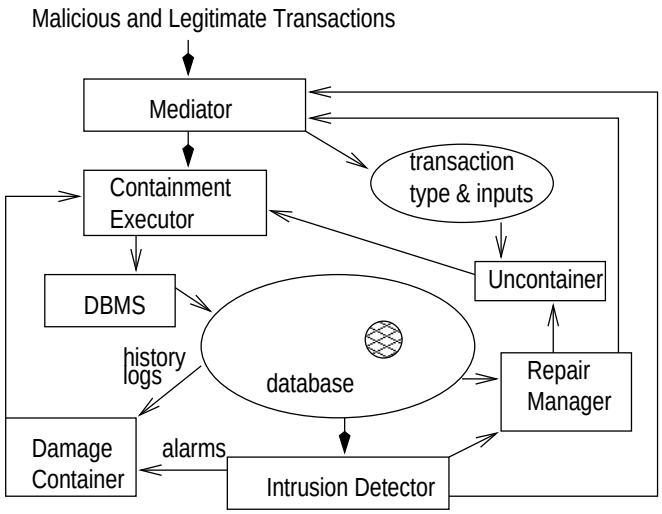

Figure 5: Proactive Damage Quarantine

correctness of repair. (c) When a cleaning operation $w_U[x]$ is scheduled, x must be dirty.

4.3.2 Dynamic DQR Solutions with Proactive Quarantine

From the viewpoint of on-the-fly non-stop recovery, fault/damage quarantine can be viewed as part of recovery. The goal of damage quarantine is to prevent the damage from spreading out during recovery. One problem of the solution shown in Figure 4 is that its damage quarantine may not be effective, since it *contains* the damage by disallowing transactions to read the set of data objects that are identified (by the Damage Assessor) as corrupted. This reactive *one-phase* damage containment approach has a serious drawback, that is, it cannot prevent the damage caused on the objects that are corrupted but not yet located from spreading. Assessing the damage caused by a malicious transaction B can take a substantial amount of time, especially when there are a lot of transactions executed during the detection latency of B. During the *assessment latency*, the damage caused during the detection latency can spread to many other objects before being contained.

The approach shown in Figure 5 integrates a novel multi-phase damage containment technique to tackle this problem. In particular, the damage containment process has one containing phase, which instantly contains the damage that *might* have been caused (or spread) by the intrusion as soon as the intrusion is detected, and one or more later on uncontaining phases to uncontain the objects that are mistakenly contained during the containing phase, and the objects that are cleaned. In this approach, the *Damage Container* will enforce the containing phase (as soon as a malicious transaction is reported) by sending some containing instructions to the *Containment Executor*. The *Uncontainer*, with the help from the Damage Assessor, will enforce the uncontaining phases by sending some uncontaining instructions to the Containment Executor. The Containment Executor controls the access of the user transactions to the database according to these instructions.

When a malicious transaction B is detected, the containing phase must ensure that the damage caused directly or indirectly by B will be contained. In addition, the containing phase

must be quick enough because otherwise either a lot of damage can leak out during the phase, or substantial availability can be lost. Time stamps can be exploited to achieve this goal. The containing phase can be done by just adding an access control rule to the Containment Executor, which denies access to the set of objects updated during the period of time from the time B commits to the time the containing phase starts. This period of time is called the *containing-time-window*. When the containing phase starts, every active transaction should be aborted because they could spread damage. New transactions can be executed only after the containing phase ends.

It is clear that the containing phase *overcontains* the damage in most cases. Many objects updated within the containing time window can be undamaged. And we must uncontain them as soon as possible to reduce the corresponding availability loss. Accurate uncontainment can be done based on the reports from the Damage Assessor, which could be too slow due to the assessment latency. [LJ01] shows that transaction *types* can be exploited to do much *quicker* uncontainment. In particular, assuming that (a) each transaction T_i belongs to a transaction type $type(T_i)$ and (b) the *profile* for $type(T_i)$ is known, the *read set template* and *write set template* can be extracted from $type(T_i)$'s profile. The templates specify the kind of objects that transactions of $type(T_i)$ can read or write. As a result, the *approximate* read-from dependency among a history of transactions can be quickly captured by identifying the read-from dependency among the types of these transactions. Moreover, the type-based approach can be made more accurate by *materializing* the templates of transactions using their inputs before analyzing the read-from dependency among the types.

Other damage quarantine methods (a) In [AJMB97], a color scheme for marking and containing damage is used to develop a mechanism by which databases under attack could still be safely used. This scheme assumes that each data record has an (accurate) initial damage mark or color (note that such marks may be generated by the damage assessment process), then specific color-based access controls are enforced to make sure that the damage will not spread from corrupted data objects to clean ones.

(b) *Attack Isolation* The idea is to isolate likely suspicious transactions before a definite determination of intrusion is reported. In particular, when a suspicious session B is discovered, isolating B and the associated transactions transparently into a separate environment that still appears to B to be the actual system allows B's activities to be kept under surveillance without risking further harm to the system. An isolation strategy that has been used in such instances is known as *fishbowling*. Fishbowling involves setting up a separate lookalike host or file system and transparently redirecting the suspicious entity's requests to it. This approach allows the incident to be further studied to determine the real source, nature, and goal of the activity, but it has some limitations, particularly when considered at the application level. First, the substitute host or file system is essentially sacrificed during the suspected attack to monitor B, consuming significant resources that may be scarce. Second, since B is cut off from the real system, if B proves innocent, denial of service could still be a problem. While some types of service B receives from the substitute, fishbowl system may be adequate, in other cases the lack of interaction with the real system's resources may prevent B from continuing to produce valid results. On the other hand, if the semantics of the application are such that B can continue producing valid work, this work will be lost when the incident concludes even if B is deemed innocent and reconnected to the real system. The fishbowling mechanism makes no provision for re-merging updates from the substitute, fishbowl system back into the real system.

In [LJM00, LWL06], these limitations are overcome by taking advantage of action semantics and the dependency relationships between transactions. In this method, as in the case of fishbowling, when B comes under suspicion, B is allowed to continue working while the security officer attempts to determine whether there is anything to worry about. At the same time, the system is isolated from any further damage B might have in mind. However, this method provides the isolation without consuming duplicate resources to construct an entirely separate environment, allows options for partial interaction across the boundary, and provides data-consistency-preserving algorithms for smoothly merging B's work back into the real system should B prove innocent. Among the partial interaction options, the *one-way isolation* concept is particularly interesting. One-way isolation allows being-isolated transactions to read the newest updates done by (trusted) transactions running on the main database, but forbids trusted transactions from reading any updates done by being-isolated transactions.

4.4 Quality Evaluation

Correctness does not always imply high quality. Two correct DQR schemes may yield very different quality levels in the DQR services they provide. In failure recovery, the MTTF-MTTR model (Mean Time To Failure - Mean Time To Recovery model) provides a neat yet precise way to gain concrete understanding of the quality of a recovery service which is measured by MTTF/(MTTF+MTTR), and this quality model has played a crucial role in advancing the theories and technologies of failure recovery. Unfortunately, due to the reasons mentioned in Section 1, the MTTF-MTTR model is no longer sufficient for defining the quality of DQR services.

In principle, the *quality* of DQR services can be evaluated by a vector composed of three criteria regarding *data integrity* and two criteria regarding *availability*:

- *C1: Dirtiness* depends on the percentage of corrupted data objects in each data store state.
- *C2: Data Freshness* When a clean yet older version of a corrupted data object o is made accessible during recovery, freshness depends on whether a fresher version of o is used by new transactions. Note that one clean version can be much fresher than another clean version of the same data object.
- *C3: Data Consistency* Violation of serializability can compromise data consistency no matter the history is multi-versioned or not.
- *C4: Rewarding Availability* The more clean or cleaned data objects are made accessible to new transactions, the more *rewarding* availability (or business continuity) is achieved. The more rewarding availability, the less denial-of-service will be caused.
- *C5: Hurting Availability* The more corrupted data objects are made accessible to new transactions, the more *hurting* availability is yielded. Because hurting availability will hurt data integrity and spread the damage, hurting availability is worse than letting the corrupted objects be quarantined.

An important finding gained in reliability evaluation research (e.g., [SS98, Tri02]) is that state transition models may play a big role in quality evaluation. A state transition model specific for DQR systems can be the model shown in Figure 6, where in terms of any portion of the application (e.g., a set of data objects), the system has 6 basic states: they are self explanatory except that the 'M' state means that the portion is Marked as damaged. Ignoring the 'Q' state,

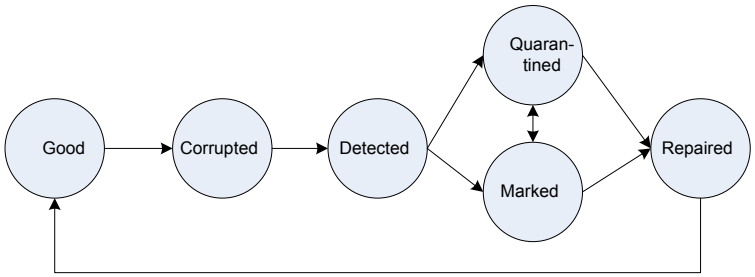

Figure 6: DQR System State Transition

we could measure Dirtiness by (MTTC+MTTM+MTTR)/(MTTC+MTTD+MTTM+MTTR); and Rewarding Availability by (MTTC+MTTR)/(MTTC+MTTD+MTTM+MTTR). In [WL06], this idea is well justified in the context of intrusion tolerant database systems through Continuous Time Markov Chain based state transition model analysis and prototype experiments based validation.

5 Remaining Research Issues and Concluding Remarks

Although DQR is not a new concept, existing attack (or intrusion) recovery research activities (see Section 4) are still quite limited in satisfying the DQR needs of real world applications, for the following reasons: (1) A theoretic understanding of the correctness and quality of DQR schemes is still missing in the literature. Since classic failure recovery theories cannot handle quarantine or on-the-fly recovery, new DQR theories are necessary to understand the strength and weakness of existing DQR schemes, inspire the development of novel DQR schemes, and make DQR a rigor field of study, for example. (2) There is still a big gap in engineering practical DQR capabilities for real world applications. For one example, Web Services (WS) and service-oriented architectures have significantly changed the way applications are developed, but no WS aware techniques have yet been developed to do transparent DQR for WS-based applications. For another example, existing transaction-level DQR mechanisms either require major changes in system design or suffer from significant DoS or performance overhead.

Therefore, to fully solve the DQR problem, a holistic approach should be taken to make an integrated set of innovative contributions on four fundamental aspects of DQR: theories, mechanisms, applications, and systems.

- New DQR theories should be developed to (a) address quarantine and *transparency*, (b) define *quality* of DQR services, and (c) integrate *recoverability* and *quarantinability*.
- New DQR schemes should be developed to advance the state-of-the-art DQR techniques from the paradigm of read-write-dependency analysis to the new paradigm of mark-based causality tracing, which will significantly improve transparency and efficiency.
- Non-blocking repair schemes should be developed to advance the state-of-the-art DQR techniques, from the paradigm of “clean-then-reexecute” recovery to the new paradigm of “cleaning-free” recovery, which avoids the overhead introduced by cleaning transactions.

- New DQR schemes should be developed to advance the state-of-the-art from the paradigm of "lock-competing reexecution" to the new paradigm of "non-blocking repair".
- New DQR schemes should be developed to advance the state-of-the-art from the paradigm of "pre-programmed DQR" to "adaptive or self-reconfigurable DQR".
- DQR theories and mechanisms should handle both conventional applications (which require ACID properties) and modern applications which adopt a weaker consistency model to make distributed "business transaction" processing (on top of Web Services) practical, scalable, and efficient.
- From the perspective of system building, complete open-source DQR tools and systems should be prototyped and evaluated using the appropriate benchmarks.

Acknowledgement Peng Liu was supported in part by NSF CCR-TC-0233324 and NSF/DHS 0335241.

References

[AJL02] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious trans. *IEEE Trans. on Knowledge and Data Engineering*, 15(5):1167–1185, 2002.

[AJM95] P. Ammann, S. Jajodia, and P. Mavuluri. On the fly reading of entire databases. *IEEE Trans. on Knowledge and Data Engineering*, 7(5):834–838, October 1995.

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

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[BC91] K. Berman and R. Cooper. The ISIS Project: Real Experience with a Fault Tolerant Programming System. *Operating Systems Review*, pages 103–107, 1991.

[BGJ00] D. Barbara, R. Goel, and S. Jajodia. "Using Checksums to Detect Data Corruption". In *Int'l Conf. on Extending Data Base Technology*, Mar 2000.

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

[Bir05] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer, 2005.

[BKTV05] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion Detection in RBAC-administered Databases. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[ca] Ca data availability solutions. http://www3.ca.com/solutions/SubSolution.aspx?ID=312.

[CC89] B. E. Clark and M. J. Corrtgan. Application System/400 performance characteristics. *IBM Syst. J.*, 28(3), 1989.

[CF01] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Eighth IEEE HOTOS*, 2001.

[CGL00] C. Y. Chung, M. Gertz, and K. Levitt. Demids: A misuse detection system for database systems. In *14th IFIP WG11.3 Working Conference on Database and Application Security*, 2000.

[CK96] Carter and Katz. Computer Crime: An Emerging Challenge for Law Enforcement. *FBI Law Enforcement Bulletin*, 1(8), December 1996.

[Cru84] R. Crus. Data recovery in IBM Database 2. *IBM Syst. J.*, 23(2), 1984.

[db2a] Db2 log analysis tool for z/os. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2lat.html.

[db2b] Db2 recovery expert for multiplatforms. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2re/.

[DBSW89] G. N. Dixon, G. D. BARRINGTON, S. SHRIVASTAVA, and S. M. Wheater. The treatment of persistent objects in Arjuna. *Comput. J.*, 32(4), 1989.

[DLA02] P. Dasgupta, R. Leblanc, and W. Appelbe. The Clouds distributed operating system. In *Proceedings 8th International Conference on Distributed Computing Systems*, San Jose, Calif., 2002.

[drd] Oracle data protection and disaster recovery solutions. http://www.oracle.com/technology/deploy/availability/htdocs/OracleDRSolutions.html.

[EAmWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[FHSL96] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 1996.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and S. Shasha. The dangers of replication and a solution. In *ACM SIGMOD*, 1996.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(2):186–213, June 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

[GPF+05] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara. The Taser Intrusion Recovery System. In *ACM SOSP*, 2005.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

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

[GS89] A. Gheith and K. Schwan. CHAOS: Support for real-time atomic transactions. In *Proc. 19th International Symposium on Fault-Tolerant Computing*, Chicago, 1989.

[HMSC88] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in Quick-Silver. *ACM Transactions on Computer Systems*, 6(1), 1988.

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

[Jef85] David R. Jefferson. Virtual time. *ACM Transaction on Programming Languages and Systems*, 7(3):404–425, July 1985.

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

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating trans. In *the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[KRL97] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a Specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.

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

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus . In *ACM SOSP*, pages 111–122, 1987.

[LD97] Jun-Lin Lin and Margaret H. Dunham. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[LD01] Jun-Lin Lin and Margaret H. Dunham. A low-cost checkpointing technique for distributed databases. *Distributed and Parallel Databases*, 10(3):241–268, 2001.

[LJ01] P. Liu and S. Jajodia. Multi-phase damage confinement in database systems for intrusion tolerance. In *14th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 2001.

[LJL+04] P. Liu, J. Jing, P. Luenam, Y. Wang, L. Li, and S. Ingsriswang. "The Design and Implementation of a Self-Healing Database System". *J. of Intelligent Information Systems (JIIS)*, 23(3):247–269, 2004.

[LJM00] P. Liu, S. Jajodia, and C.D. McCollum. Intrusion confinement by isolation in information systems. *J. of Computer Security*, 8(4):243–279, 2000.

[LL91] Yi-bing Lin and Edward D. Lazowska. A study of time warp rollback machanisms. *ACM Transactions on Modeling and Computer Simulations*, 1(1):51–72, January 1991.

[Lom92] D.B. Lomet. MLR: A recovery method for multi-level systems. In *ACM-SIGMOD International Conference on Management of Data*, pages 185–194, San Diego, CA, June 1992.

[LR04] Barbara Liskov and Rodrigo Rodrigues. Transactional File Systems Can Be Fast. In *11th ACM SIGOPS European Workshop*, 2004.

[LS83] B. Liskov and R. SCHEIFLER. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Program. Lang. Syst.*, 5(3), 1983.

[LVB06] D. Lomet, Z. Vagena, and R. Barga. Recovery from Bad User Transactions. In *ACM SIGMOD*, 2006.

[LWL06] Peng Liu, Hai Wang, and Lunquan Li. Real-time Data Attack Isolation for Commercial Database Applications. *Elsevier Journal of Network and Computer Applications*, 29(4):294–320, 2006.

[LX01] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[MBPR96] G. Muller, M. Banatre, N. Peyrouze, and R. Rochat. Lessons from FTM: An Experiment in the Design & Implementation of a Low-Cost Fault-Tolerant System. *IEEE Transactions on Reliability*, 45(2):332–340, 1996.

[MG96] 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.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking. *ACM Trans. on Database Systems*, 17(1):94–162, 1992.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only trans. In *ACM SIGMOD International Conference on Management of Data*, pages 124–133, San Diego, CA, June 1992.

[MVS00] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *4th Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[NKK86] E. Nett, J. Kaiser, and R. Kroger. Providing recoverability in a transaction oriented distributed operating system. In *Proc. 6th International Symposium on Fault-Tolerant Computing*, Cambridge, May 1986.

[ora] Oracle database. http://www.oracle.com/database/index.html.

[OWA04] OWASP. Owasp top ten most critical web application security vulnerabilities. http://www.owasp.org/documentation/topten.html, January, 27 2004.

[Pax99] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, pages 2435–2463, 1999.

[PBB+02] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kycyman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley Computer Science, 2002. CSD-02-1175.

[PcC05] Dhruv Pilania and Tzi cker Chiueh. Design, Implementation, and Evaluation of an Intrusion Resilient Database System. In *Proc. International Conference on Data Engineering*, 2005.

[PG98] 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.

[Pu86] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, October 1986.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *ACM SOSP*, 2003.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch05] Bruce Schneier. Attack trends 2004 and 2005. *ACM Queue*, 3(5), June 2005.

[SDD85] A. Z. Spector, D. Daniels, and D. Duchamp. Distributed Transactions for Reliable Systems. In *ACM SOSP*, 1985.

[SESS96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI*, 1996.

[SFL97] S. Stolfo, D. Fan, and W. Lee. Credit card fraud detection using meta-learning: Issues and initial results. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[SGF+02] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Implementation of Argus Specification-based anomaly detection: a new approach for detecting network intrusions. In *ACM CCS*, 2002.

[son] Sonasoft disaster recovery solutions. http://www.sonasoft.com/solutions/disaster.asp.

[sql] Sql server. http://www.microsoft.com/sql/default.mspx.

[SS98] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[sym] Symantec livestate recovery products provide fast, reliable and cost-effective system and data recovery. http://www.symantec.com/press/2004/n041005.html.

[Tri02] K. S. Trivedi. *"Probability and statistics with reliability, queuing and computer science applications"*. John Wiley and Sons, 2002.

[WHBM90] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *the Ninth ACM SIGACT-SIGMOD-SIGART Symposium of Principles of Database Systems*, pages 109–123, Nashville, Tenn, April 1990.

[WL06] H. Wang and P. Liu. Modeling and Evaluating the Survivability of an Intrusion Tolerant Database System. In *Proc. ESORICS (European Symposium on Research in Computer Security)*, 2006.

[WS92] G. Weikum and H.-J. Schek. Concepts and applications of multilevel trans. and open nested trans. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, Inc., 1992.

[YLZ04] M. Yu, P. Liu, and W. Zang. "Self Healing Workflow Systems under Attacks". In *24th IEEE Int'l Conf. on Distributed Computing Systems*, 2004.

[ZC03] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, 2003.