The Pennsylvania State University

The Graduate School

Department of Information Sciences and Technology

Damage Management in Database Management Systems

A Dissertation in

Information Sciences and Technology

by

Kun Bai

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2010

The dissertation of Kun Bai was reviewed and approved[1] by the following:

Peng Liu
Associate Professor of Information Sciences and Technology
Dissertation Adviser
Chair of Committee

Chao-Hsien Chu
Professor of Information Sciences and Technology

Thomas La Porta
Distinguished Professor of Computer Science and Engineering

Sencun Zhu
Assistant Professor of Computer Science and Engineering

Frederico Fonseca
Associate Professor of Information Sciences and Technology
Associate Dean, College of Information Sciences and Technology

[1]Signatures on file in the Graduate School.

Abstract

In the past two decades there have been many advances in the field of computer security. However, since vulnerabilities cannot be completely removed from a system, successful attacks often occur and cause damage to the system. Despite numerous technological advances in both security software and hardware, there are many challenging problems that still limit effectiveness and practicality of existing security measures.

As Web applications gain popularity in today's world, surviving Database Management System (DBMS) from an attack is becoming even more crucial than before because of the increasingly critical role that DBMS is playing in business/life/mission-critical applications. Although significant progress has been achieved to protect the DBMS, such as the existing database security techniques (e.g., access control, integrity constraint and failure recovery, etc.,), the buniness/life/mission-critical applications still can be hit due to some new threats towards the back-end DBMS. For example, in addition to the vulnerabilities exploited by attacks (e.g., the SQL injections attack), databases can be damaged in several ways such as the fraudulent transactions (e.g., identity theft) launched by malicious outsiders, erroneous transactions issued by the insiders by mistakes. When the database is under such a circumstance (attack), rolling back and re-executing the damaged transactions are the most used mechanisms during the system recovery. This kind of mechanism either stops (or greatly restricts) the database service during repair, which causes unacceptable data availability loss or denial-of-service for mission critical applications, or may cause serious damage spreading during

on-the-fly recovery where many clean data items are accidentally corrupted by legitimate new transactions. In this study, we address database damage management (DBDM), 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.

Although a number of research projects have been done to tackle the emerging data corruption threats, existing mechanisms are still limited in meeting four highly desired requirements: *near-zero-run-time overhead*, *zero-system-down time*, *zero-blocking-time* for read-only transactions, *minimal-delay-time* for read-write transactions. Firstly, to achieve the four highly desired requirements, we propose *TRACE*, a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. TRACE consists of a family of new database damage tracking, quarantine, and cleansing techniques. We built TRACE into the kernel of PostgreSQL. Secondly, motivated by the limitation of TRACE mechanism, we propose a novel proactive damage management approach denoted *database firewalling*. This approach deals with transaction level attacks. Pattern mining and Bayesian network techniques are adopted in the firewalling framework to mine frequent *damage spreading patterns* and to predict the data integrity in the face of attack when certain type of attack occurs repeatedly. This pattern mining and Bayesian inference approach provides a probability based strategy to estimate the data integrity on the fly. With this probabilistic feature, the database firewalling approach is able to enforce a policy of transaction filtering to dynamically filter out the potential *damage spreading transactions*.

Table of Contents

List of Tables x

List of Figures xi

Acknowledgments xiii

Chapter 1. Introduction 1

1.1 Problems in This Study 4

1.1.1 Light Weighted Damage Quarantine and Recovery System . . 4

1.1.2 Preventive Damage Management Approach 5

1.2 Contributions of This Work 7

1.3 Outline 8

Chapter 2. Related Works 10

2.1 Traditional Database Security Research 11

2.2 Failure Handling Research 16

2.3 Intrusion Detection System (IDS) Research 20

2.4 Intrusion Tolerant Database Research 27

2.4.1 Self-Healing Software Systems 27

2.4.2 Intrusion Tolerant Database Systems 32

Chapter 3. The TRACE Damage Management System 42

3.1 Introduction 42

3.2 Background 44
3.2.1 Existing Solutions and Their Limitations 45
3.3 Preliminaries and Problem Statement 47
3.3.1 The Threat Model 47
3.3.2 An Motivated Example using SQL Injection Attack 48
3.3.3 Basic Concepts of the Database System 49
3.3.4 Problem Statement 50
3.3.4.1 Dependency Relations 50
3.3.4.2 Problem Statement 52
3.4 Overview of Approach 53
3.4.1 Assumptions 54
3.4.2 The Standby Mode 55
3.4.3 The Cleansing Mode 58
3.4.3.1 Damage Quarantine 58
3.4.3.2 Damage Assessment 60
3.4.3.3 Valid Data De-Quarantine 61
3.4.3.4 Repairing On-The-Fly 64
3.5 Design and Implementation of TRACE atop PostgreSQL 65
3.5.1 Implementing the Marking Scheme 66
3.5.2 Maintaining Before Images 68
3.5.3 Damage Assessment Module 68
3.5.4 Quarantine/De-Quarantine Module 69
3.5.5 On-The-Fly Repairing 71

3.5.6 Garbage collection 72
3.6 TRACE-FG: A Fine-Grained Damage Management Scheme 75
3.6.1 A Fine-grained Version-data-status-identification Method . . 75
3.6.2 Fine-grained De-quarantine and Repair 78
3.7 Experimental Results 79
3.7.1 Evaluation of TRACE System 79
3.7.1.1 System Overhead 80
3.7.1.2 Zero System Down Time 81
3.7.1.3 TRACE vs. 'point-in-time' recovery 83
3.7.1.4 System CPU Time Distribution 86
3.7.1.5 Reduced Service Delay Time and Saved Legitimate Transactions 87
3.7.1.6 IDS Impacts on TRACE System 90
3.7.2 Evaluation of TRACE-FG System 93
3.7.2.1 Saved Data Versions 95
3.7.2.2 Reduced Delay Time/Saved De-committed Transactions 98
3.7.2.3 TRACE-FG vs. TRACE and 'point-in-time' on System Throughput 100
3.8 Discussion 101
3.9 Appendix 102
3.9.1 TPC-C Transaction Read/Write Set Template 102
3.9.2 Clinic OLTP Application Transaction Read/Write Template . 105

Chapter 4. Preventive Damage Management through Filtering of Damage Spreading Transactions 109
4.1 Introduction 109
4.2 Background and Preliminaries 111
4.2.1 The Threat Model 111
4.2.2 The System Model 112
4.3 Overview of Approach 113
4.3.1 Rational 114
4.3.2 Database Firewall Architecture 115
4.4 Mining Damage Spreading Patterns 118
4.4.1 Two Types of Damage Spreading Patterns 118
4.4.2 Mining The Damage Spreading Patterns 119
4.4.2.1 Basic Definitions 120
4.4.2.2 Finding One-Hop Spreading Patterns 122
4.4.2.3 Finding Multi-Hop Spreading Patterns 123
4.5 Integrity Level Estimator 124
4.5.1 Bayesian Network Based Analysis on Data Integrity 126
4.5.2 Integrity Estimation Using Bayesian Network 128
4.5.3 Algorithm of Integrity Estimation 131
4.5.4 Updating the Filtering Rules On-The-Fly 131
4.6 Experimental Results 133
4.6.1 Experiment Settings 134
4.6.2 Mining Algorithms Testing 135

4.6.3 System Integrity Analysis 137

4.6.4 System Availability Analysis 139

4.6.5 System Throughput Analysis 141

Chapter 5. Conclusion 145

5.1 Summary 145

5.2 Future Work 148

5.2.1 Research Plan 148

5.2.1.1 Task 1. Non-Blocking Repair Using a Shadow Database. 149

5.2.1.2 Task 2. Machine Learning based Preventive Damage Management. 150

Bibliography 151

List of Tables

3.1 Existing Approaches and Their Limitations. 44

3.2 TPC-C Parameters and Their Values 80

4.1 An Example of Attack History H_i^A 112

4.2 An Example of An Attack History H_i^A Grouped By Patient ID and Sorted By Transaction Time 113

4.3 One-Hop Spreading Pattern Mapping 121

4.4 An Example of Attack Histories $H_i^{A'}$ After Mapping 121

4.5 An Example of Using Attack Histories to Mine One-Hop Spreading Patterns 121

4.6 TPC-C Parameters and Their Values 135

4.7 Parameters of the synthetic datasets 135

4.8 Parameter values of the datasets 136

4.9 Measurement of the Mining Algorithm of One-hop Patterns 136

4.10 Speed of System Integrity Estimation 138

List of Figures

3.1 An Example of Damage Spreading 46
3.2 The TRACE System Workflow 55
3.3 An Example of the Causality Table Construction 56
3.4 Identify/Repair Corrupted Data Records Overview 59
3.5 Example of Repairing Corrupted Data Records 62
3.6 Data Record Structure with Marking Bits 67
3.7 TRACE Assessment/Repair with Causality Table in PostgreSQL 70
3.8 An Example of the relationship between the process window and the missing probability 74
3.9 Cases of Assessment Procedure Limitations 76
3.10 Evaluation of TRACE System 82
3.11 Evaluation of TRACE System 85
3.12 Evaluation of TRACE System 88
3.13 Evaluation of TRACE System 89
3.14 Evaluation of TRACE System 92
3.15 Evaluation of TRACE System 94
3.16 Evaluation of TRACE-FG System 96
3.17 Evaluation of TRACE-FG System 97
3.18 System Throughput of TRACE-FG vs. TRACE and 'PIT' with d=%25 101

4.1 Database Firewall Architecture 115

4.2 The Database Firewall Workflow 117

4.3 An Example of Two Types of Damage Spreading 119

4.4 An Example of Bayesian Network with the Damage Spreading Probability Table 129

4.5 System Integrity Estimation Process. Real System Integrity SI_r=0.8 . . 139

4.6 System Availability without Repairing Procedure 140

4.7 System Availability with Repairing Procedure 142

4.8 System Throughput 143

Acknowledgments

This dissertation represents the culmination of years of research that succeeded only with the support and encouragement of a widespread collection of talented scientists. I am deeply indebted to my excellent adviser, Dr. Peng Liu. I think back five or six years and remember the student that I was before entering his tutelage, and I realize just how much I have grown because of his influence. Peng helped me understand the process of research and why that process is important. Beyond simply helping me complete a paper or finalize a set of experiments, Peng taught me how to be a scientist. By his thoughtfully and patiently training, I gained insights about how to teach and advise others. I cannot image how I could achieve my current status without his constantly support and encouragement. I thank my dissertation committee – Professor Chao-Hsien Chu, Professor Thomas La Porta, and Professor Sencun Zhu – for taking the time to study my research and to provide useful insights that improved the dissertation. Throughout my graduate study, I have had a number of fruitful collaborations with other researchers. I thank C. Lee Giles, Prasenjit Mitra for their cooperative research. Finally, I would like to thank my family, my beloved wife (Ying) and daughter (Serena) for encouraging me throughout my five years of Ph.D. study.

Chapter 1

Introduction

Database Damage Management (DDM), especially the self-healing capability, is an important problem faced today by a great number of mission/life/business critical applications. As the Internet applications gain popularity and are embraced in industry to support today's E-Business world, more and more threats towards the back-end database systems are identified. Surviving the back-end Database Management System (DBMS) from E-Crime is becoming even more crucial than before because of the increasingly critical role that the DBMS is playing and more critical and valuable information stored in databases which is now processed through the Web and is world wide accessible. A database system with the self-healing capability aims to assure these applications, such as banking, online stock trading, and air traffic control, etc., with high data integrity and service availability because these applications are the cornerstones of a variety of crucial information systems and infrastructures that must manage risk, business continuity, and data assurance in the presence of severe cyber-attacks. Although significant progress has been made in protecting such applications and systems, these mission/life/business-critical applications still have a "good" chance to suffer from a big "hit" from 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. As we have seen that malicious attacks are difficult to prevent, self-healing capability is an indispensable part of the corresponding high assurance solution to satisfy the increasing demands on risk management, business continuity, and data assurance.

Conventional database management system (DBMS) failure recovery mechanisms are very mature in handling random failures, but have fundamental differences from attack self-healing. Existing failure recovery and the DBMS security techniques (e.g., authentication based access control, integrity constraints) are designed to guarantee the correctness, integrity, and availability of the stored data, but are very limited in dealing with data corruption. Moreover, 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 [15], SQL injection [55], as well as from malicious insiders. As we will explain shortly in section 3.3, once a database server is attacked, the damage (data corruption) done by these attacks can have severe impact because not only is the data they write invalid (corrupted), but the data written by all other legitimate transactions that read these corrupted data may likewise become invalid. In this way, legitimate transactions can accidentally spread the damage to other innocent data.

In database security research, recent research concerns are the confidentiality, integrity, and availability of data stored in a database. Existing proposed research works primarily address how to protect the security of a database, especially its confidentiality, but seldom focus on how to survive successful database attacks, which can seriously

impair the integrity and availability of a database. These existing DBMS security techniques are designed to guarantee the correctness, integrity, and availability of the stored data, but are very limited in dealing with data damage (corruption) and do not deal with the problem of malicious transactions. For example, access control can be subverted by the inside or outside attacker who has assumed an insider's identity. Integrity constraints are weak at prohibiting plausible but incorrect data.

In addition, these insufficiently protected Web applications are not very difficult to break through [35] due to a well known reason that system vulnerabilities cannot be completely eliminated. For instance, online applications are often a combination of the following components: application servers, databases, and application specific code (e.g., server-side transaction procedures). Usually, on one hand, the back-bone software infrastructure (e.g., the web servers and database) is developed by experienced developers who have comprehensive knowledge of the security. On the other hand, the application oriented code is often developed under tight schedules by programmers who lack security training. Thus, such vulnerabilities can be exploited by skillful attackers with some efforts. One of the consequent results of such malicious exploits is the data corruption and integrity loss, a primary threat to the current data intensive applications. Experience with data-intensive applications has shown that a variety of attacks can successfully fool traditional protection mechanisms. In addition to the vulnerabilities exploited by attacks (e.g., the SQL injections attack), data corruption (data damage) can be caused in several ways such as the fraudulent transactions (e.g., identity theft) launched by malicious outsiders, erroneous transactions issued by the insiders by mistakes.

Applications mentioned above are mission/business-critical and are the cornerstones of a variety of crucial information systems, which play important role in many nation's critical infrastructures, such as financial services,telecommunication infrastructure, and transportation control. Hence, one of the main challenges of current database security research is how to manage the data corruption (damage).

1.1 Problems in This Study

Damage management is a very important problem faced today by many mission, life, or business-critical applications and information systems that must manage risk, business continuity, and assurance in the presence of severe cyber attacks. Damage management is a broad research topic that spans over several perspectives. In this study, we investigate the critical damage management techniques in database security research from the following dimensions: *damage spreading*, *damage quarantine and recovery*, and *damage management correctness and quality.*

1.1.1 Light Weighted Damage Quarantine and Recovery System

Experience with data-intensive applications such as credit card billing, online banking, inventory tracking, and online stock trading, has shown that a variety of attacks successfully fool traditional database protection mechanisms. Due to data sharing and inter-operability between business process and applications, cyber attacks could even enhance their damage because of catastrophic cascading effects.

Techniques, such as *flashback*[53] implemented in Oracle database and [36], can handle damaged data, but are costly to use and have serious impact on the compromised

DBMSs. These techniques can seriously impair the database availability because not only the malicious transaction, but all the transactions committed after the malicious transaction are rolled back. Although a good number of research projects have been done to tackle the emerging data corruption threats, existing mechanisms are still quite limited in meeting four highly desired requirements: (R1) *near-zero-run-time overhead*, (R2) *zero-system-down time*, (R3) *zero-blocking-time* for read-only transactions, (R4) *minimal-delay-time* for read-write transactions. As a result, these proposed approaches introduce two apparent issues: 1) substantial run time overhead, 2) long system outage.

In this study, we propose *TRACE*, a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. The service outage is minimized by (a) cleaning up the compromised data on-the-fly, (b) using multiple versions to avoid blocking read-only transactions, and (c) doing damage assessment and damage cleansing concurrently to minimize delay time for read-write transactions.

1.1.2 Preventive Damage Management Approach

As we will explain shortly in chapter 3.3 in details, once a DBMS is attacked, the damage (data corruption) done by these malicious transactions has severe impact on the DBMS because not only are 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. When a database system is attacked, the immediate negative effects (basically, the data damage) caused particularly by this attack are usually relatively limited. However, the

negative effect can enlarge its impact largely because the data damage can be spread to other innocent data stored in the database system due to the nature of transactional read/write dependency of the database systems. we name it *damage spreading*. The database system armed with existing security technologies cannot continue providing satisfactory services because the integrity of some data objects is compromised. Simply identifying and repairing these compromised data objects by operating undo and redo transactions still cannot ensure the database integrity due to the damage spreading.

We propose to answer the following question: When a database system is identified under an attack by either an intrusion detection system (IDS) or a database administrator (DBA), how can the server prevent spread of damage while continuously providing data services? In this work, we take the first step to solve this problem. In particular, we propose a novel proactive damage management approach denoted *database firewalling*. This approach deals with transaction level attacks. We assume that these data corruption related attacks often leave a fingerprint in the system after attacks are launched, namely *damage spreading pattern*. The idea of this approach is to extract robust damage spreading pattern out of previous attack histories. When specific damage spreading patterns repeatedly appear under the same *type* of attacks, we can extract these patterns using some data mining techniques. Then, we can use these mined patterns to quickly predict the data objects that could have been corrupted during the intrusion detection latency window even before the time-consuming damage assessment procedure starts. Thereafter, we can set up firewalling rules to police data access requests from newly arrived transactions and block only the data objects that match the pre-mined patterns. In this way, spread of damage can be dramatically reduced (if the patterns are good),

while the great majority of data objects in the database will be continuously accessible during the entire online recovery process.

1.2 Contributions of This Work

In this work, we make the following contributions:

- we develop a light weight erroneous transaction tracing system, *TRACE* which captures the bad transactions caused by damage propagation, with a minimum amount of system outage.
- we develop a suite of tools to isolated the identified data records and repair them on-the-fly, and thus achieve the maximum system throughput during the time period of the system inconsistence.
- we propose *TRACE-FG*, a fine-grained zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. TRACE-FG is able to decompose the "in-danger" unit and save the legitimated work.
- we apply our approach to open source PostgreSQL database, and find better performance than the current recovery approach applied in PostgreSQL database.
- we have demonstrated practical performance achievement and the feasibility of deployment of TRACE in the real database system.
- we propose a novel proactive damage management approach denoted *database firewalling*.

- we present an association rule based mining algorithm to discover the frequent damage spreading pattern. In addition, we provide an bayesian network based algorithm to dynamically estimate the data integrity using the discovered frequent spreading patterns.

1.3 Outline

The proposal is organized as follows. In Chapter 2, we present the main researches related to our studies. In Chapter 3, we propose TRACE, a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. In Chapter 4, we propose a novel mechanism, called *database firewalling* in this study to protect good data from being corrupted. In Chapter 5, we summarize the proposal, and overview the future works.

Chapter 3 is organized as follows. Section 3.3 describes some threats TRACE intends to handle and the problem statement. Section 3.4 overviews the key ideas of TRACE to identify and repair damaged data records on-the-fly. Section 3.5 introduces how we develop TRACE in PostgreSQL database system. Section 3.6 proposes a fine-grained damage management system TRACE-FG. Section 3.7 demonstrates the experimental results of our TRACE system in comparison with current recovery mechanisms.

Chapter 4 is organized as follows. Some definitions used in this paper, the architecture of the database firewall, and an example of how damage spreads are presented in section 4.2. The framework of frequent *spreading pattern* mining is presented in section 4.4.2. The idea of predicting the integrity of data objects using Bayesian network are presented in section 4.5. Empirical studies are conducted in section 4.6.

Chapter 5 is organized as follows. Section 5.1 summarizes the dissertation. We mainly focus on investigating the critical security techniques in database damage management. Section 5.2.1 describes the future work. In the future work, we see that many opportunities for continued research in data damage management remain. We focus on developing from two perspectives that could improve our existing proposed approach.

Chapter 2

Related Works

In this chapter, we summarize the related works in the literature in database security researches, and give an overview on the different perspectives these researches have addressed.

we review the work in the database security and related areas, first considering the most common addressed problem in recent database security research. Section 2.1 considers this problem and presents research works in the complementary fields of the traditional database security. We focus on research in the areas of authorization, multi-level secure database, and inference control. Section 2.2 considers different failure handling model designs and how those designs impact. We focus on failure handling systems that cover from media failure recovery and workflow failure recovery, as these reflect our own area of work. We investigate the most basic problem in intrusion detection: how to detect attacks in Computer Networks and Database systems research and examine the trade-offs among different model construction algorithms in Section 2.3. In contrast to the active research in the construction and design of models, few studies of detection effectiveness and resilience to intelligent attackers have been conducted. Section 2.4 discusses the small body of work in designing intrusion tolerant database system models and compares the work with our methods of evaluation. As described in Chapter

1, our research contributes new solutions to secure database design, and evaluation. In each of these areas, we contrast previous research with our own work.

2.1 Traditional Database Security Research

In general, database security concerns the confidentiality, integrity, and availability of data stored in a database. A broad span of research from authorization [24, 28, 63], to inference control [1], to multilevel secure databases [65, 76], and to multilevel secure transaction processing [5], addresses primarily how to protect the security of a database, especially its confidentiality. In this chapter, we overview the existing proposed methodologies in database security research and the related topics from the perspectives of failure handling, intrusion detection, and intrusion tolerant, respectively.

In [24], the authors address a common problem in a multiuser database system, in which the database must selectively permit users to share data, while retaining the ability to restrict data access. There must be a mechanism to provide protection and security, permitting information to be accessed only by properly authorized users. Further, when tables or restricted views of tables are created and destroyed dynamically, the granting, authentication, and revocation of authorization to use them must also be dynamic. Each of these issues and their solutions in the context of the relational database management system System R are discussed. When a database user creates a table, he is fully and solely authorized to perform upon it actions such as read, insert, update, and delete. He may explicitly grant to any other user any or all of his privileges on the table. In addition he may specify that that user is authorized to further grant these privileges to still other users. The result is a directed graph of granted privileges originating from the

table creator. At some later time a user A may revoke some or all of the privileges which he previously granted to another user B. This action usually revokes the entire subgraph of the grants originating from A's grant to B. It may be, however, that B will still possess the revoked privileges by means of a grant from another user C, and therefore some or all of B's grants should not be revoked. This problem is discussed in detail in [24], and an algorithm for detecting exactly which of B's grants should be revoked is also presented.

In [28], the authors present a new access control mechanism. Although several access control policies can be devised for controlling access to information, all existing authorization models, and the corresponding enforcement mechanisms, are based on a specific policy (usually the closed policy). As a consequence, although different policy choices are possible in theory, in practice only a specific policy can be actually applied within a given system. However, protection requirements within a system can vary dramatically, and no single policy may simultaneously satisfy them all. In [28], the authors present a flexible authorization manager (FAM) that can enforce multiple access control policies within a single, unified system. FAM is based on a language through which users can specify authorizations and access control policies to be applied in controlling execution of specific actions on given objects. The authors formally define the language and properties required to hold on the security specifications and prove that this language can express all security specifications. Furthermore, they show that all programs expressed in this language (called FAM/CAM-programs) are also guaranteed to be consistent (i.e., no conflicting access decisions occur) and CAM-programs are complete (i.e., every access is either authorized or denied). They also illustrate how several well-known protection policies proposed in the literature can be expressed in the FAM/CAM language and how

users can customize the access control by specifying their own policies. The result is an access control mechanism which is flexible, since different access control policies can all coexist in the same data system, and extensible, since it can be augmented with any new policy a specific application or user may require.

The conventional models of authorization have been designed for database systems supporting the hierarchical, network, and relational models of data. However, these models are not adequate for next-generation database systems that support richer data models that include object-oriented concepts and semantic data modeling concepts. In [63], the authors present a fuller model of authorization that fills a few major gaps that the conventional models of authorization cannot fill for next-generation database systems. We also further formalize the notion of implicit authorization and refine the application of the notion of implicit authorization to object-oriented and semantic modeling concepts. They also describe a user interface for using the model of authorization and consider key issues in implementing the authorization model.

Security-control methods suggested in the literature are classified into four general approaches: conceptual, query restriction, data perturbation, and output perturbation. Criteria for evaluating the performance of the various security-control methods are identified. Security-control methods that are based on each of the four approaches are discussed, together with their performance with respect to the identified evaluation criteria. In [1], the authors consider the problem of providing security to statistical databases against disclosure of confidential information. A detailed comparative analysis of the most promising methods for protecting dynamic-online statistical databases is also presented. To date no single security-control method prevents both exact and

partial disclosures. There are, however, a few perturbation-based methods that prevent exact disclosure and enable the database administrator to exercise statistical disclosure control. Some of these methods, however introduce bias into query responses or suffer from the 0/1 query-set-size problem (i.e., partial disclosure is possible in case of null query set or a query set of size 1). The authors recommend directing future research efforts toward developing new methods that prevent exact disclosure and provide statistical-disclosure control, while at the same time do not suffer from the bias problem and the 0/1 query-set-size problem. Furthermore, efforts directed toward developing a bias-correction mechanism and solving the general problem of small query-set-size would help salvage a few of the current perturbation-based methods.

Many multilevel relational models have been proposed in recent database research. Different models offer different advantages. In [65], the authors adapt and refine several of the best ideas from previous models and add new ones to build the new Multilevel Relational (MLR) data model. MLR provides multilevel relations with element-level labeling as a natural extension of the traditional relational data model. MLR introduces several new concepts (notably, data-borrow integrity and the UPLEVEL statement) and significantly redefines existing concepts (poly-instantiation and referential integrity as well as data manipulation operations). A central contribution of this paper is proofs of soundness, completeness, and security of MLR. A new data-based semantics is given for the MLR data model by combining ideas from SeaView, belief-based semantics, and LDV. This new semantics has the advantages of both eliminating ambiguity and retaining upward information flow. MLR is secure, unambiguous, and powerful. It has five integrity properties and five operations for manipulating multilevel relations. Soundness,

completeness, and security show that any of the five database manipulation operations will keep database states legal (i.e., satisfy all integrity properties), that every legal database state can be constructed, and that MLR is non-interfering. Multilevel security poses many challenging problems for transaction processing. The challenges are due to the conflicting requirements imposed by confidentiality, integrity, and availability, the three components of security. We identify these requirements on transaction processing in Multilevel Secure (MLS) database management systems (DBMSs) and survey the efforts of a number of researchers to meet these requirements. In [5], the authors emphasize primarily on centralized systems based on kernelized architecture. And, they briefly overview the research in the distributed MLS DBMSs as well.

The addition of stringent security specifications to the list of requirements for an application poses many new problems in DBMS design and implementation, as well as database design, use, and maintenance. Tight security requirements, such as those that result in silent masking of withholding of true information from a user or the introduction of false information into query answers, also raise fundamental questions about the meaning of the database and the semantics of accompanying query languages. In [76], the authors propose a belief-based semantics for secure databases, which provides a semantics for databases that can lie about the state of the world, or about their knowledge about the state of the world, in order to preserve security. This kind of semantics can be used as a helpful retrofit for the proposals for a multilevel secure database model (a particularly stringent form of security), and may be useful for less restrictive security policies as well. They also propose a family of query languages for multilevel secure relational database applications, and base the semantics of those languages on our semantics

for secure databases. Our query languages are free of the semantic problems associated with use of ordinary SQL in a multilevel secure context, and should be easy for users to understand and employ.

2.2 Failure Handling Research

Databases can handle erroneous data if a disk fails, either catastrophically or by losing some bits. This is called media failure, and how to deal with it is denoted as media recovery. Commonly, database systems take regular backups, a special form of replica optimized for high speed writing and reading. Media recovery involves backup load (restore) and redo recovery (roll forward) using a media recovery log that records transaction updates since the backup, hence restoring the database to its most recent state. This process can be arduous, and result in a long outage.

The failure handling of workflow has been discussed in recent work [16, 18, 72]. In [16], the authors develop unified techniques for complex business processes modeled as cooperative transaction hierarchies. Multiple cooperative transaction hierarchies often have operational dependencies, thus a failure occurring in one transaction hierarchy may need to be transferred to another. The authors also introduce the notion of transaction execution history tree which allows one to develop a unified hierarchical failure recovery mechanism applicable to both nested and flat transaction structures. They also develop a cross-hierarchy undo mechanism for determining failure scopes and supporting backward and forward failure recovery over multiple transaction hierarchies. These mechanisms form a structured and unified approach for handling failures in flat transactional workflows, along a transaction hierarchy, and across transaction hierarchies. In [18], the

authors state that workflow management systems (WFMSs) more and more become the basic technology for organizations to perform their daily business processes (workflows). A consistent and reliable execution of such processes is crucial for all organizations. They claim that this can only be achieved by integrating transactional features - especially "workflow transactions" - into WFMSs. Based on this idea, they discuss in detail advanced workflow recovery concepts which are necessary for the reliable and consistent execution of business processes in the presence of failures and exceptions. Additionally, they distinguish between different workflow types and present adequate recovery concepts for each of them. In [72], the authors study the pre-defined business processes which can substantially simplify the process design and the implementation of run time support. However, requiring that all the instances of the business process follow a fixed pattern (even if conditions are allowed) does not offer users sufficient flexibility to make changes to the process structures. These changes may be necessary due to the occurrence of exceptions, or other ad-hoc events. Some exceptions/events may be predictable in advance, and therefore can be incorporated into the process definition. However, not all exception/events can be predicted at the process definition time. When this happens, no corresponding exception-specific provision can be incorporated. The design and implementation of the system support for unpredictable exception/event handling is therefore a more complicated issue. In [72], they study a special case of changing the structure of a business process in the context of unpredictable exception/event, namely, redirecting the control flow at run time in an ad-hoc manner. This phenomenon is termed ad-hoc recovery. The authors concentrate on two aspects in supporting ad-hoc recovery, the kind of interface that should be used and the increased functionality that must be built

into the workflow database. For the latter, we also suggest implementation strategies to maximize the performance.

Failure handling is different with attack recovery in two aspects. On one hand, they have different goals. Failure handling tries to guarantee the atomic of workflows. When failure happens, their work find out which tasks should be aborted. If all tasks are successfully executed, failure handling does nothing for the workflow. Attack recovery has different goals, which need to do nothing for failure tasks even if they are malicious because failure malicious tasks have no effects on the workflow system. Attack recovery focuses on malicious tasks that are successfully executed. It tries to remove all effects of such tasks. On the other hand, these two systems active at different times. Failure handling occurs when the workflows are in progress. When the IDS reports attacks, the malicious tasks usually have been successfully executed. Failure handling can do nothing because no failure occurred. Attack recovery is supposed to remove the effects of malicious tasks after they are committed.

Failure handling of transaction, e.g., [16, 18, 72], aims at guaranteeing the atomicity of database systems when some transactions failed. For instance, when a disk fails (media failure), most traditional recovery mechanisms (media recovery) focus on recovering the legitimate state of the database to its most recent state upon system failure by applying backup load and redo recovery [12]. 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. In [50], the authors present a

simple and efficient method, called ARIES (Algorithm for Recovery and Isolation Exploiting Semantics), which supports partial rollbacks of transactions, fine granularity (e.g., record) locking and recovery using write-ahead logging (WAL). ARIES introduces the paradigm of repeating history to redo all missing updates before performing the rollbacks of the loser transactions during restart after a system failure. All updates of a transaction are again logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart of of nested rollbacks. Removing inconsistency induced by malicious transactions is usually based on the recovery mechanisms introduced in [50], 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. Besides the above recovery mechanism, checkpoint techniques [39] are widely used to preserve the integrity of data stored in databases by rolling back the whole database system to a specific time point. Checkpointing a database is a vital technique to reduce the recovery time in the presence of a failure. For distributed databases, checkpointing also provides an efficient way to perform global reconstruction. Since the need for global reconstruction is

infrequent in most distributed databases, a less restrictive and less resource-consuming approach to checkpoint distributed databases in an integrated distributed database system is recommended over a transaction consistent checkpoint approach. For a federated or multidatabase system, any type of global consistent checkpoint is difficult to achieve without violating local autonomy. However, all work, done by both malicious and innocent transactions, will be lost.

2.3 Intrusion Detection System (IDS) Research

One critical step towards TRACE and the database firewalling systems is Intrusion Detection (ID), which has attracted many researchers [13, 19, 20, 27, 30, 37, 59, 74]. The existing methodology of ID can be roughly classified into two groups. Firstly, anomaly detection [30, 38, 64, 66]. In [30], SRI International's real-time intrusion-detection expert system (IDES) contains a statistical subsystem that observes behavior on a monitored computer system and adaptively learns what is normal for individual users and groups of users. The statistical subsystem also monitors observed behavior and identifies behavior as a potential intrusion (or misuse by authorized users) if it deviates significantly from expected behavior. The multivariate methods used to profile normal behavior and identify deviations from expected behavior are explained in detail. The statistical test for abnormality contains a number of parameters that must be initialized and the substantive issues relating to setting those parameter values are discussed. Anomaly detection is an essential component of protection mechanisms against novel attacks. The authors [38] propose to use several information-theoretic measures, namely, entropy, conditional entropy, relative conditional entropy, information gain, and

information cost for anomaly detection. These measures can be used to describe the characteristics of an audit data set, suggest the appropriate anomaly detection model(s) to be built, and explain the performance of the model(s). In [64], the authors present IDAMN (intrusion detection architecture for mobile networks), a distributed system whose main functionality is to track and detect mobile intruders in real time. IDAMN includes two algorithms which model the behavior of users in terms of both telephony activity and migration pattern. The main novelty of our architecture is its ability to perform intrusion detection in the visited location and within the duration of a typical call, as opposed to existing designs that require the reporting of all call data to the home location in order to perform the actual detection. The algorithms and the components of IDAMN have been designed in order to minimize the overhead incurred in the fixed part of the cellular network. In [66], the authors introduce a new intrusion detection approach that identifies anomalous sequences of system calls executed by programs. Since their work, anomaly detection on system call sequences has become perhaps the most successful approach for detecting novel intrusions. A natural way for learning sequences is to use a finite-state automaton (FSA). However, previous research seemed to indicate that FSA-learning is computationally expensive, that it cannot be completely automated, or that the space usage of the FSA may be excessive. The new approach presented in this paper overcomes these difficulties. Their approach builds a compact FSA in a fully automatic and efficient manner, without requiring access to source code for programs. The space requirements for the FSA is low–of the order of a few kilobytes for typical programs. The FSA uses only a constant time per system call during the learning as well as detection period. This factor leads to low overheads for intrusion detection. Unlike

many of the previous techniques, this FSA-technique can capture both short term and long term temporal relationships among system calls, and thus perform more accurate detection.

Secondly, misuse detection [20, 27]. In [20], the authors argue that model-based anomaly detectors can retrofit efficient attack detection ability to vulnerable programs. These detectors restrict a process execution using a pre-computed model of normal, expected behavior. The authors construct models of behavior using static binary analysis. While previous statically-constructed models have traded attack detection ability for performance, the new Dyck model is the first statically-constructed model that balances security and performance, and it demonstrates that the previous trade-off was not a fundamental limitation of static analysis. They further improve the Dyck model by incorporating into the model information about data values used in the program and about the execution environment in which the program runs. The paper [27] presents a new approach to representing and detecting computer penetrations in real time. The approach, called state transition analysis, models penetrations as a series of state changes that lead from an initial secure state to a target compromised state. State transition diagrams, the graphical representation of penetrations, identify precisely the requirements for and the compromise of a penetration and present only the critical events that must occur for the successful completion of the penetration. State transition diagrams are written to correspond to the states of an actual computer system, and these diagrams form the basis of a rule based expert system for detecting penetrations, called the state transition analysis tool (STAT). However, the majority of the existing ID research focus on identifying attacks on OS and computer networks. In general, IDSs monitor system

activities to discover attempts to gain illicit access to systems or corrupt data objects in systems.

Existing methodologies of ID are in two categories either based on *statistical profile* [30, 64, 66] or on *known patterns of attacks*, called *signatures* [20, 26, 27, 60, 67]. In [60], A new approach to representing computer penetrations is introduced called penetration state transition analysis. This approach models penetrations as a series of state transitions described in terms of signature actions and state descriptions. State transition diagrams are written to correspond to the states of an actual computer system, and these diagrams form the basis of a rule-based expert system for detecting penetrations, referred to as STAT. In [26], the authors present the design and implementation of a real-time intrusion detection tool, called Us-TAT', a State Transition Analysis Tool for UNIX. This is a UNIX-specific implementation. State Transition Analysis is a new approach to representing computer penetrations. In STAT, a penetration is identified as a sequence of state changes that take the computer system from some initial state to a target compromised state. In [26], the development of the first USTAT prototype, which is for SunOS 4.1.1, is discussed. Us-TAT makes use of the audit trails that are collected by the C2 Basic Security Module of Sun OS, and it keeps track of only those critical actions that must occur for the successful completion of the penetration. This approach differs from other rule-based penetration identification tools that pattern match sequences of audit records. In [67], the authors address the operational security problems, which are often the result of access authorization misuse, and can lead to intrusion in secure computer systems. They present a model that tracks both data and privilege flows within secure systems to detect context-dependent intrusions caused by operational security

problems. The model allows the uniform representation of various types of intrusion patterns, such as those caused by unintended use of foreign programs and input data, imprudent choice of default privileges, and use of weak protection mechanisms. As with all pattern-oriented models, this model cannot be used to detect new, unanticipated intrusion patterns that could be detected by statistical models.

Intrusion detection can supplement protection of network and information systems by rejecting the future access of detected attackers and by providing useful hints on how to strengthen the defense. These ID systems indeed make the system attack-aware but not attack-resistant. In addition, they focus on identifying attacks on OS and computer networks and cannot be directly applied to detect malicious transactions and data corruption. In general, intrusion detection systems have a few noticeable limitations: (1) Intrusion detection makes the system attack-aware but not attack-resistant. that is, intrusion detection itself cannot maintain the integrity and availability of the database in face of attacks. (2) Achieving accurate detection is usually difficult or expensive. The false alarm rate is high in many cases. (3) The average detection latency in many cases is too long to effectively confine the damage.

Some works on database IDS [13, 37, 73, 74, 78] that are suitable to detect malicious transactions or abnormal behaviors in database systems are published recently. In [13], the authors propose such a mechanism based on mining database traces stored in log files. The result of the mining process is used to form user profiles that can model normal behavior and identify intruders. An additional feature of my approach is that we couple my mechanism with Role Based Access Control (RBAC). Under a RBAC system permissions are associated with roles, usually grouping several users, rather than with

single users. This ID system is able to determine role intruders, that is, individuals that while holding a specific role, have a behavior different from the normal behavior of the role. An important advantage of providing an ID mechanism specifically tailored to databases is that it can also be used to protect against insider threats. Furthermore, the use of roles makes this approach usable even for databases with large user population.. The ID approach proposed in [13] is based on mining database traces stored in log files. The result of the mining process is used to form user profiles that can model normal behavior and identify intruders.

A database user session is a sequence of queries issued by a user (or an application) to achieve a certain task. Analysis of task-oriented database user sessions provides useful insight into the query behavior of database users. In [78], the authors describe novel algorithms for identifying sessions from database traces and for grouping the sessions different classes.

In [37], the authors identify that there is a growing security concern on the increasing number of databases that are accessible through the Internet. Such databases may contain sensitive information like credit card numbers and personal medical histories. Many e-service providers are reported to be leaking customers information through their websites. The hackers exploited poorly coded programs that interface with back-end databases using SQL injection techniques. They develop an architectural framework, DIDAFIT (Detecting Intrusions in DAtabases through FIngerprinting Transactions) that can efficiently detect illegitimate database accesses. The system works by matching SQL statements against a known set of legitimate database transaction fingerprints. The

authors explore the various issues that arise in the collation, representation and summarization of this potentially huge set of legitimate transaction fingerprints. They design an algorithm that summarizes the raw transactional SQL queries into compact regular expressions. This representation can be used to match against incoming database transactions that explore poorly written programs using SQL injection techniques efficiently. A set of heuristics is used during the summarization process to ensure that the level of false negatives remains low. This algorithm also takes into consideration incomplete logs and heuristically identifies potential malicious transactions. This framework can efficiently detect illegitimate database accesses. Careless development of web-based applications results in vulnerable code being deployed and made available to the whole Internet, creating easily-exploitable entry points for the compromise of entire networks. To ameliorate this situation, F. Valeur et al. [73] propose an approach that composes a web-based anomaly detection system with a reverse HTTP proxy. The approach is based on the assumption that a web sites content can be split into security sensitive and non-sensitive parts, which are distributed to different servers. The anomaly score of a web request is then used to route suspicious requests to copies of the web site that do not hold sensitive content. By doing this, it is possible to serve anomalous but benign requests that do not require access to sensitive information, sensibly reducing the impact of false positives. F. Valeur et al. [74] develop an anomaly-based system that learns the profiles of the normal database access performed by web-based applications using a number of different models. These models allow for the detection of unknown attacks with reduced false positives and limited overhead. In addition, our solution represents an improvement with respect to previous approaches because it reduces the possibility of

executing SQL-based mimicry attacks. However, all of the above approaches are unable to detect the damage spread by executing normal transactions.

2.4 Intrusion Tolerant Database Research

The need for intrusion tolerance has been recognized by many researchers in such contexts as information warfare [23]. Recently, extensive research has been done in general principles of survivability [22, 33, 75], survivability of networks [48], survivable storage systems [77], survivable application development via middleware [56], persistent objects [46], and survivable document editing systems [71].

2.4.1 Self-Healing Software Systems

Secure survivable architectures are typically very application- or domain-specific. In [21], the authors propose "fault injection analysis" applied to software. They are concerned with the survivability of the infrastructure to software flaws, anomalous events, and malicious attack. In the past, finding and removing software flaws has traditionally been the realm of software testing. Software testing has largely concerned itself with ensuring that software behaves correctly — an intractable problem for any non-trivial piece of software. The authors present "off-nominal" testing techniques that are not concerned with the correctness of the software, but with the survivability of the software in the face of anomalous events and malicious attack. Where software testing is focused on ensuring that the software computes the specified function correctly, they are concerned that the software continues to operate in the presence of unusual system events or malicious attacks. The off-nominal testing approach uses fault injection analysis to

determine the effect of unusual or malicious attacks against software. Fault injection is the process of perturbing or corrupting a data state during program execution. Fault injection analysis is the process of determining the effect of that perturbation. The analysis may consist of simply measuring whether the perturbed state affected a particular output, or the analysis may determine whether system attributes such as safety, security, or survivability have been affected [12].

In [70], the authors apply a low-level approach: they propose an intrusion detection and recovery model at the storage layer. Self-securing storage turns storage devices into active parts of an intrusion survival strategy. From behind a thin storage interface (e.g., SCSI or CIFS), a self-securing storage sen,er can watch storage requests, keep a record of all storage activity, and prevent compromised clients from destroying stored data. This paper describes three ways self-securing storage enhances an administrator's ability to detect, diagnose, and recover from client system intrusions. First, storage-based intrusion detection offers a new observation point for noticing suspect activity. Second, post-hoc intrusion diagnosis starts with a plethora of normally-unavailable information. Finally, post-intrusion recovery is reduced to restarting the system with a pre-intrusion storage image retained by the sensor. Combined, these features can improve an organization's ability to survive successful digital intrusions.

In [34], the authors propose a formalized feedback driven model for individual COTS applications. They address the problem of information system survivability, or dynamically preserving intended functionality & computational performance, in the face of malicious intrusive activity. A feedback control approach is proposed which enables

tradeoffs between the failure cost of a compromised information system and the maintenance cost of ongoing defensive countermeasures. Online implementation features an inexpensive computation architecture consisting of a sensor-driven recursive estimator followed by an estimate-driven response selector. Offline design features a systematic empirical procedure utilizing a suite of mathematical modeling and numerical optimization tools. The engineering challenge is to generate domain models and decision strategies offline via tractable methods, while achieving online effectiveness. The authors illustrate the approach with experimentation results for a prototype autonomic defense system which protects its host, a Linux-based web-server, against an automated Internet worm attack.

SABER proposed in [32] is a generalized, application-neutral architecture that encompasses a broad array of tools. The authors present SABER (Survivability Architecture: Block, Evade, React), a survivability architecture that blocks, evades and reacts to a variety of attacks by using several security and survivability mechanisms in an automated and coordinated fashion. Contrary to the ad hoc manner in which contemporary survivable systems are built-using isolated, independent security mechanisms such as firewalls, intrusion detection systems and software sandboxes-SABER integrates several different technologies in an attempt to provide a unified framework for responding to the wide range of attacks malicious insiders and outsiders can launch. This coordinated multi-layer approach will be capable of defending against attacks targeted at various levels of the network stack, such as congestion-based DoS attacks, software-based DoS or code-injection attacks, and others. Our fundamental insight is that while multiple lines of defense are useful, most conventional, uncoordinated approaches fail to exploit the

full range of available responses to incidents. By coordinating the response, the ability to survive successful security breaches increases substantially.

The APOD project [4] uses a combination of intrusion detection, firewalls, TCP stack probes, virtual private networks, bandwidth reservation, and traffic shaping mechanisms, to allow applications to detect attacks and contain the damage of successful intrusions by changing their behavior. They also discuss the use of randomizing techniques, such as changing the TCP ports applications listen to.

[69, 45] explore the notion of collaborative security with specific application to coordinating IDS alerts for worms and scanning attacks across administrative domains. In [69, 45], the authors describe the Worminator project that detects, reports, and defends against early pre-attack cyber-events–specifically network observables–that are precursors to malicious activities during a later attack stage.

Indra [29] is another scheme that provides a peer-to-peer approach to intrusion detection. While the spread of the Internet has made the network ubiquitous, it has also rendered networked systems vulnerable to malicious attacks orchestrated from anywhere. These attacks or intrusions typically start with attackers infiltrating a network through a vulnerable host and then launching further attacks on the local network or Intranet. Attackers rely on increasingly sophisticated techniques like using distributed attack sources and obfuscating their network addresses. On the other hand, software that guards against them remains rooted in traditional centralized techniques, presenting an easily-targeted single point of failure. Scalable, distributed network intrusion prevention techniques are sorely needed. The authors propose Indra: a distributed scheme based

on sharing information between trusted peers in a network to guard the network as a whole against intrusion attempts.

In addition, a collaborative approach to containing the spread of worms has been the focus of current research [49, 51, 61]. Vigilante [31] proposes the concept of Self-Certifying Alerts, which are exchanged between hosts as a result of a newly detected attack. The recipient can verify the validity of the alert and use an appropriate protection mechanism. In Vigilante, every host checks for all attacks all the time, in contrast to our more general load-sharing-capable approach. Furthermore, we propose a software-based protection mechanism (as opposed to their use of filtering) that both protects against attacks and also maintains application availability, thus providing an element of real software healing. In [52], the authors study algorithms for the assignment of distinct software packages (whether randomized or inherently different) to individual systems in a network, towards increasing the intrinsic value of available diversity. Their goal is to limit the ability of a malicious node to compromise a large number (or any) of its neighbors with a single attack. Unfortunately, their abstraction does not translate well to the end-to-end semantics of the Internet, where any host can contact another without (in most cases) needing to pass through a series of other hosts. Their work can be viewed as a situation where a community of nodes collaboratively diversifies.

In [54, 14], Gamma is proposed as an architecture for instrumenting software such that information that can lead to future improvements of the code can be gathered in a central location, without imposing excessive overhead to any given code instance. Their technique, software tomography, has been combined with a dynamic software update mechanism that allows code producers to fix bugs as they are detected.

2.4.2 Intrusion Tolerant Database Systems

In [22], the authors argue that intrusion detection and response research has so far mostly concentrated on known and well-defined attacks, and this narrow focus of attacks accounts for both the successes and limitation of commercial intrusion detection systems (IDS). Intrusion tolerance, on the other hand, is inherently tied to functions and services that require protection. They presents a state transition model to describe the dynamic behavior of intrusion-tolerant systems. This model provides a framework from which we can define the vulnerability and the threat set to be addressed. They also show how this model helps to describe both known and unknown security exploits by focusing on impacts rather than specific attack procedures. By going through the exercise of mapping known vulnerabilities to this transition model, they identify a reasonably complete fault space that should be considered in a general intrusion-tolerant system. Survivability architectures enhance the survivability of critical information systems by providing a mechanism that allows the detection and treatment of various types of faults. In [33], the authors discuss four of the issues that arise in the development of such architectures and summarize approaches that they are developing for their solution. The addition of stringent security specifications to the list of requirements for an application poses many new problems in DBMS design and implementation, as well as database design, use, and maintenance. Tight security requirements, such as those that result in silent masking of withholding of true information from a user or the introduction of false information into query answers, also raise fundamental questions about the meaning of the database and the semantics of accompanying query languages. In [75],

some applications can be given increased resistance to malicious attack even though the environment in which they run is untrustworthy. The authors call any such application "defense-enabled". This paper explains the principles behind defense enabling and the assumptions on which it depends. This kind of semantics can be used as a helpful retrofit for the proposals for a "multilevel secure" database model (a particularly stringent form of security), and may be useful for less restrictive security policies as well. They also propose a family of query languages for multilevel secure relational database applications. A major attack can significantly reduce the capability to deliver services in large-scale networked information systems. In this project, we have addressed the survivability of large scale heterogeneous information systems which consist of various services provided over multiple interconnected networks with different technologies. The communications network portions of such systems are referred to as multi-networks. We specifically address the issue of survivability due to physical attacks that destroy links and nodes in multi-networks. The end goal is to support critical services in the face of a major attack by making optimum use of network resources while minimizing network congestion. This is an area which is little studied, especially for large-scale heterogeneous systems. A detailed overview is addressed in [48]. Survivable storage systems [77] must maintain data and access to it in the face of malicious and accidental problems with storage servers, interconnection networks, client systems, and user accounts. These four component types can be grouped into two classes: server-side problems and client-side problems. The PASIS architecture addresses server-side problems, including the connections to those servers, by encoding data with threshold schemes and distributing trust amongst sets of storage servers. Self-securing storage addresses client and user account problems by

transparently auditing accesses and versioning data within each storage server. Thus, PASIS clients use threshold schemes to protect themselves from compromised servers, and self-securing servers use full access auditing to protect their data from compromised clients. Together, these techniques can provide truly survivable storage systems.

The original rollback mechanism called lazy cancelation has aroused great interest. In [40], the author study these rollback mechanisms. The general tradeoffs between aggressive and lazy cancelation are discussed, and by a conservative-optimal simulation is defined for comparative purposes. Within the framework of aggressive cancelation, we offer some observations and analyze the rollback behavior of tandom systems. The lazy cancelation mechanism is examined using a metric called the sensitivity of output message. Both aggressive and lazy cancelation are shown to work well for a process with a small simulated load intensity. Finally, an analytical model is given to analyze message preemption, an important factor that affects the performance of rollback mechanisms. Results indicate that message preemption has a significant effect on performance when (1) the processor is highly utilized, (2) the execution times of messages have high variance, and (3) rollbacks occur frequently.

The authors in [56] consider two aspects of survivability namely survival by adaptation and survival by protection. They show how the quality objects (QuO) distributed adaptive middleware framework enables users to introduce these aspects of survivability in a flexible and systematic manner. They describe in [56] a toolkit for developing adaptive applications and demonstrate how more survivable applications can be built using the toolkit. Fleet [46] is a middleware system implementing a distributed repository for

persistent Java objects. Fleet is primarily targeted for supporting highly critical applications: in particular, the objects it stores maintain correct semantics despite the arbitrary failure (including hostile corruption) of a limited number of Fleet servers and, for some object types, of clients allowed to invoke methods on those objects. Fleet is designed to be highly available, dynamically extensible with new object types, and scalable to large numbers of servers and clients. In [71], the authors design and implement a tool protect the end-to-end document integrity by wrapping the tools that manipulate those documents and mediating their operation to cryptographically integrity mark those documents as they are being saved, to check those cryptographic integrity marks as those documents are loaded, and to record an application-level history of the changes to the document. Corrupted documents (those failing to match their cryptographic integrity mark) are automatically repaired by replaying the recorded history of the application-level changes to the document. That recorded history is also used to identify all modifications (including date and author) to any selected portion of the document.

Some research has also been done in database intrusion tolerance. In [3], a fault tolerant approach is taken to survive database attacks where (a) several useful survivability phases are suggested, though no concrete mechanisms are proposed for these phases; (b) a color scheme for marking damage (and repair) and a notion of integrity suitable for partially damaged databases are used to develop a mechanism by which databases under attack could still be safely used.

Fault tolerant approaches [3] are introduced to survive and recover a database from attacks and system flaws. A color scheme for marking damage and a notion of integrity suitable for partially damaged databases are proposed to develop a mechanism

by which databases under attack could still be safely used. For traditional database systems, *Data oriented attack recovery* mechanisms [57] recover compromised data by directly locating the most recent untouched version of each corrupted data, and *transaction oriented attack recovery* [41] 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 [11], the authors consider the problem of malicious and intended corruption of data in a database, acting outside of the scope of the database management system. Although detecting an attacker who changes a set of database values at the disk level is a simple task (achievable by attaching signatures to each block of data), a more sophisticated attacker may corrupt the data by replacing the current data with copies of old block images, compromising the integrity of the data. To prevent successful completion of this attack, they provide a defense mechanism that enormously increases the intruders workload, yet maintains a low system cost during an authorized update. This algorithm calculates and maintains two levels of signatures (checksum values) on blocks of data. The signatures are grouped in a manner that forces an extended series of block copying for any unauthorized update. Using the available information on block sizes, block reference patterns and amount of concurrently active transactions in the database, they calculate the length of this chain of copying, proving that the intruder has to perform a lot of work in order to go undetected. Therefore, this checksum technique makes this type of attack very unlikely. Storage jamming can degrade real-world activities that share stored data. In addition, storage jamming is not prevented by access controls or cryptographic techniques. Verification to rule out

storage jamming logic is impractical for shrink-wrapped software or low-cost custom applications. Detection mechanisms do offer more promise. In [47], the authors model storage jamming and a detection mechanism, using Unity logic. They find that Unity logic, in conjunction with some high-level operators, models storage jamming in a natural way and allows us to reason about susceptibility, rate of jamming, and impact on persistent values. *Storage jamming* [47] is used to seed a database with dummy values, access to which indicates the presence of an intruder.

Attack recovery has different goals from media failure recovery, which focuses on malicious and affected transactions only. Previous work[2, 3, 6, 41, 43, 57] of attack recovery heavily depends on exploiting the system log to find out the pattern of damage spreading and schedule repair transactions. In [2], the authors adopt an information warfare perspective, which assumes success by the attacker in achieving partial, but not complete, damage. In particular, they work in the database context and consider recovery from malicious but committed transactions. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of benign transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. Recovery is complicated by the presence of benign transactions that depend, directly or indirectly, on the malicious transactions. The authors present algorithms to restore only the damaged part of the database and identify the information that needs to be maintained for such algorithms. The initial algorithms repair damage to quiescent databases; subsequent algorithms increase availability by allowing new transactions to execute concurrently with the repair process. In [3], the authors consider the problem of surviving information warfare attacks on

databases. They adopt a fault tolerance approach to the different phases of an attack. To maintain precise information about the attack, they mark data to reflect the severity of detected damage as well as the degree to which the damaged data has been repaired. In the case of partially repaired data, integrity constraints might be violated, but data is nonetheless available to support mission objectives. Moreover, the authors define a notion of consistency suitable for databases in which some information is known to be damaged, and other information is known to be only partially repaired. They present a protocol for normal transactions with respect to the damage markings and show that consistency preserving normal transactions maintain database consistency in the presence of damage. However, the analysis of system log is very time consuming and hard to satisfy the performance requirement of on-line recovery. In addition, the dynamic algorithm proposed in [2] leaks damage to innocent data while repairing the damage on-the-fly.

In [44], a similar idea of recovering from bad transactions is proposed to automatically identify and isolate the ill-effects of a flawed transaction, and then preserving much more of the current database state while reducing the service outage time. This technique requires both a write log and a read log. Although a write log is quite common in modern DBMS, maintaining a read log poses a serious performance overhead and therefore is not supported in existing DBMS. In addition, it requires the system be off-line to mark the identified invalid data and repair them. In [17], an advanced dependency tracking technique is proposed. This approach tracks and maintains inter-transaction dependency at run time to determine the exact extent of damage caused by a malicious attack. The drawbacks of this approach are 1) it does not support on-line damage repair

and thus results in long system outage, 2) it does not support *Redo* transaction and thus results in permanently data lost. In our proposed approach, all dependency relations are dynamically collected at run time and corrupted data is automatically identified and isolated without the analysis of the system log. In addition, our approach supports on-the-fly repair thus provides remarkable system throughput improvement with limited service outage.

In [3], a fault tolerant approach is introduced to survive and recover a database from attacks. A color scheme for marking damage and a notion of integrity suitable for partially damaged databases are proposed to develop a mechanism by which databases under attack could still be safely used. Unlike the color scheme approach, which is based on an assumption that each data object has an accurate damage mark, our method focuses on dynamically assigning the damage mark and dealing with the negative impact of inaccurate damage marks. To overcome these limitations, a broader perspective has been introduced, namely an intrusion tolerance database system (ITDB) [43]. Unlike the color scheme approach, ITDB focuses on dynamically assigning the damage mark and dealing with the negative impact of inaccurate damage marks. Some work has been done on OS-level database survivability. In [11], checksums are smartly used to detect data corruption.

Failure handling aims at guaranteeing the atomicity of database systems. Checkpoint techniques [39] 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. Attack recovery has different goals from failure handling. It focuses on malicious transactions that have

been successfully executed. For traditional database systems, *Data oriented attack recovery* mechanisms [57] recover compromised data by directly locating the most recent untouched version of each corrupted data, and *transaction oriented attack recovery* [41] mechanisms do attack recovery by identifying the transactions that are affected by the attack through read-write dependencies and rolls back those affected transactions.

Many database attack recovery methods, such as [57] and [41], stops the database service during repair. To overcome this limitation, in [2], an advanced transaction-oriented attack recovery algorithm is provided that unwinds not only the effects of each malicious transaction but also the effects of any innocent transaction. However, the algorithm [2] does not prevent the damage from spreading during the detection latency or the on-the-fly recovery process. To overcome this limitation, in [42], an innovative multiphase damage containment approach is proposed, which proactively *contains* (i.e., blocks accesses to) all the data records damaged during the detection latency instantly after the malicious transaction is detected. This approach can guarantee that no damage caused by malicious transaction B_i will spread to any new update. However, the limitation of multiphase containment is that it can cost substantial data availability loss due to the *initial containment* phase of this method in which a lot data items can be mistakenly contained. Moreover, it may take a substantial amount of time for those mistakenly contained data items to be uncontained. As a result, current database survivability techniques cause the following dilemma: they either stop (or greatly restricts) the database service during repair, which causes unacceptable availability loss or denial-of-service for mission critical applications, or may cause serious damage spreading during

on-the-fly recovery where many clean data items are accidentally corrupted by legitimate new transactions.

Chapter 3

The TRACE Damage Management System

3.1 Introduction

Transactional processing systems (e.g., database systems) have become increasingly sophisticated and been critical to most cyber infrastructures, such as banking, e-Commerce, military combat-field command centers, etc. Data availability and integrity are crucial for these infrastructures. However, it is well known that system vulnerabilities cannot be completely eliminated, and such vulnerabilities can be exploited by skillful attackers.

As Web services and Internet applications gain popularity and are embraced in industry to support today's E-Business world, surviving the back-end DBMSs from E-Crime is becoming even more crucial because of the increasingly critical role that they are playing. The cost of attacks on the DBMSs are often at the magnitude of several millions of dollars. Unfortunately, traditional DBMS protection mechanisms do not defend the DBMSs against some new threats that have come along with the rise of Internet, both from external source, e.g., SQL Slammer Worm[1], SQL Injection[2], as well as from malicious insiders. Existing DBMS security techniques, e.g., authentication based access control, integrity constraints, and roll-back recovery mechanisms, are designed to

[1] http://www.cert.org/advisories/CA-2003-04.html
[2] http://www.owasp.org/index.php/OWASP_Top_Ten_Project

guarantee the correctness, integrity, and availability of the stored data, but are very limited in dealing with data corruption and do not deal with the problem of malicious transactions. As we will explain shortly in this chapter, once a DBMS is attacked, the damage (data corruption) done by these malicious transactions has severe impact on it because not only is the data they write invalid (corrupted), but the data written by all transactions that read these data may likewise be invalid. In this way, legitimate transactions can accidentally spread the damage to other innocent data. Techniques, such as implemented in Oracle *flashback*[3] and [36], can handle corrupted data, but are costly to use and have serious impact on the compromised DBMSs. These techniques can seriously impair the database availability because not only the malicious transaction, but all the transactions committed after the malicious transaction are rolled back.

Although a good number of research projects have been done to tackle the emerging data corruption threats, existing mechanisms are still quite limited in meeting four highly desired requirements: (R1) *near-zero-run-time overhead*, (R2) *zero-system-down time*, (R3) *zero-blocking-time* for read-only transactions, (R4) *minimal-delay-time* for read-write transactions. As a result, these proposed approaches introduce two apparent issues: 1) substantial run time overhead, 2) long system outage.

To overcome the above limitations, we propose *TRACE*, a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. The service outage is minimized by (a) cleaning up the compromised data on-the-fly, (b) using multiple versions to avoid blocking read-only transactions,

[3]http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.html

and (c) doing damage assessment and damage cleansing concurrently to minimize delay time for read-write transactions. Moreover, TRACE uses a novel marking scheme to track causality without the need to log read/write 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[2]. In summary, TRACE is the *first* integrated database tracking, quarantine, and recovery solution that can satisfy all the four highly desired requirements shown in Table 1. In particular, TRACE is the first solution that can simultaneously satisfy requirements R1 and R2. In addition, TRACE is the first solution that satisfied R4.

Proposals	R_1	R_2	R_3	R_4
[2]		✓	✓	
[17]	✓			
[44]	✓			
TRACE	✓	✓	✓	✓

Table 3.1. Existing Approaches and Their Limitations.

3.2 Background

As we will explain shortly in chapter 3.3 in details, once a DBMS is attacked, the damage (data corruption) done by these malicious transactions has severe impact on the DBMS because not only are 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. When a database system is attacked, the immediate negative effects (basically, the data damage) caused particularly by this attack are usually relatively limited. However, the negative effect can enlarge its impact largely because the data damage can be spread to other innocent data stored in the database system due to the nature of transactional read/write dependency of the database systems. we name it *damage spreading*. The database system armed with existing security technologies cannot continue providing satisfactory services because the integrity of some data objects is compromised. Simply identifying and repairing these compromised data objects by operating undo and redo transactions still cannot ensure the database integrity due to the damage spreading. A damage spreading example is given in Figure 3.1. Suppose T_b is a fraudulent transaction and its writeset contains corrupted data objects. After transaction T_{12} and T_{32} (also called *affected transactions*) read a data object o_i that has been updated by transaction T_b (in w_{T_b}), data objects that are updated by T_{12}, T_{32} will also be compromised. In this way, the damage will spread out and we say that the transaction T_{12}, T_{32}, and T_{23} are affected directly or indirectly by T_b.

3.2.1 Existing Solutions and Their Limitations

To see why existing data corruption tracking/recovering mechanisms are limited in satisfying the four requirements, here we briefly summarize some main limitations of three representative database damage tracking/recovering solutions [2, 17, 44]. In ITDB [2], 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

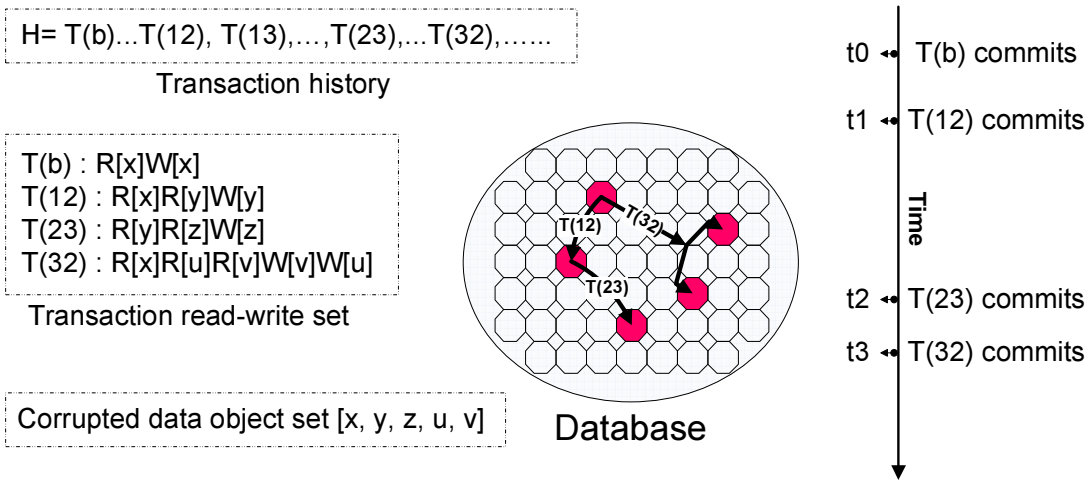


Fig. 3.1. An Example of Damage Spreading

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 [17], 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 [44], another inter-transaction dependency tracking technique is proposed to identify and isolate ill-effects of the malicious transactions. In order to maintain the data dependency, this technique also needs to record a read log, which is not supported in existing DBMS and will pose a serious performance overhead . Additionally, it only provides off-line post-corruption database repair. Table 3.1 lists the major mechanisms

and their limitations, and the four requirements will shortly be further explained in Section 2.

3.3 Preliminaries and Problem Statement

In this section, we first review a few critical security threats that can cause the data corruption problem, then introduce a set of formal definitions used in this study.

3.3.1 The Threat Model

We deal with the data corruption problem caused by transaction level attacks in database systems in this study. Transaction level attacks are not new. They have been studied in a good number of researches [3][17][68]. 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 and stronger user authentication on its own does not protect against these forms of attack.

- Second, the attacks can be done through erroneous 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 [55], 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.

In this study, we use "attack" to denote both the malicious attacks and human errors.

3.3.2 An Motivated Example using SQL Injection Attack

EXAMPLE 1. **A motivating Example of SQL Injection Attack.** *We show an example of the SQL injection attack that convinces the database application to run SQL code that is not intended. SQL Injection Attack. Although end-users do not interact with back-end database servers directly, if user input is not sanitized correctly, it is possible*

that unauthorized users can leverage the applications to corrupt the integrity of data objects stored in back-end databases. As an illustration, consider the transaction templates used in the real clinic OLTP application:

1. **Templates:** update Customer set TSales=TSales+ $amts, . . . ,

   Year_Accu_Amt= Year_Accu_Amt + $amty where Cust_Num = '$pid';

   . . .

   update Customer set Point =Point - $points,... where Cust_Num = '$pid';

2. **Injection:** $pid=' OR Cust_Num like '100%

3. **Result:** update Customer set TSales=TSales+ $amts, . . . where Cust_Num = ' ' OR Cust_Num like '100%';. . .

The transaction script shown above is originally to update the purchase accumulated at that moment of a customer and calculate the reward points accordingly. However, if the input parameters are not escaped correctly before inserting into the query templates, it allows an attack to change the query structure by injecting a piece of SQL statement. Thus, the *where-clause* is always true and some data objects that the malicious transaction is not supposed to gain access to are modified. We use this SQL injection attack to simulate the malicious transaction and evaluate our *TRACE* system in the experiments.

3.3.3 Basic Concepts of the Database System

A *database* system is a set of data objects, denoted as DB={ o^1, o^2, ..., o^n }. A *transaction* T_i is a partial order with ordering relation $\lhd_i$ [12], where

1. $T_i \subseteq \{(r_i[o^x], w_i[o^x]) \mid o^x$ is a data object$\} \cup (a_i, c_i)$;

2. if $r_i[o^x], w_i[o^x] \in T_i$, then either $r_i[o^x] \lhd_i w_i[o^x]$, or $w_i[o^x] \lhd_i r_i[o^x]$;

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 data structure called a *log*.

Formally, let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions. A complete history H over T is a partial order with ordering relation $\lhd_H$, where:

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

3.3.4 Problem Statement

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.

3.3.4.1 Dependency Relations

To accomplish the above 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 $T_i \lhd T_j$. Note, we assume* strict 2PL *scheme is applied as in most of the commercial DBMSs.*

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

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 2. *For example, given the transaction* $T_i : o^c = o^a + o^b$, $R_{T_i} = \{o^a, o^b\}$ *and* $W_{T_i} = \{\underline{o^c}\}$*; the transaction* T_j*:* $o^f = o^d + o^c$, $R_{T_j} = \{\underline{o^c}, o^d\}$, $W_{T_j} = \{o^f\}$. *Obviously, we have* $T_i \rightarrow T_j$ *because transaction* T_j *reads the data object* o^c *written by transaction* T_i. *If data objects (e.g.,* o^c*) contained in the write set* W_{T_i} *are corrupted, write set* W_{T_j} *is affected because of the dependency relation of* T_i *and* T_j, $R_{T_j} \cap W_{T_i} = o^c$. *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 3.1,* $T_b \rightarrow T_{12} \rightarrow T_{23}$, $T_{12} \lhd T_{13}$, $T_b \rightarrow T_{32}$. *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. A solid directed edge indicates a preceding relation and a dash edge indicates a data dependency relation. Note, not all preceding relations are drawn in Figure 3.1 for the sake of clarity.*

3.3.4.2 Problem Statement

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).*

Problem Statement: How to satisfy the four requirements listed in Table 1 without violating the above correctness criteria?

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 any 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.

3.4 Overview of 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 3.2 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 [62].

3.4.1 Assumptions

When a DBMS is attacked, data corruption can be detected by a good number of existing *Intrusion Detection System* (IDS) techniques [13, 19, 37, 59, 74]. Although intrusion detection techniques and tools cannot do the damage tracking/quarantine/cleansing work as the proposed new damage mitigation technique, TRACE, will do, intrusion detection is a basic component of any intrusion recovery solution. In TRACE, we adopt the following work [13, 59] into our solution. All these techniques experience certain false positive rate largely because they are influenced by the detection latency. The longer time the IDS spends, the more accuracy the system can achieve, however, the more damage the database will suffer. Additionally, since false positives can mislead TRACE to revert updates done by innocent transactions, TRACE will typically get a data corruption alarm verified before cleansing any data records. Verifying a data corruption alarm is not always a difficult thing, e.g., by techniques such as storage jamming [47], checksum based corruption verification [11], or by a *Database Administrator* (DBA) who understands the applications being served by the database. TRACE does not rely on the precision of timely reporting by the intrusion detection. As long as an attack is reported, TRACE identifies all affected transactions and repairs them. Only if the reporting delay is significant, more transactions in the system will be affected, which needs more efforts

to make the cleansing done. The *detection window* indicates the interval between when an intrusion takes place and when it is detected. In addition, we assume no blind write is performed in TRACE.

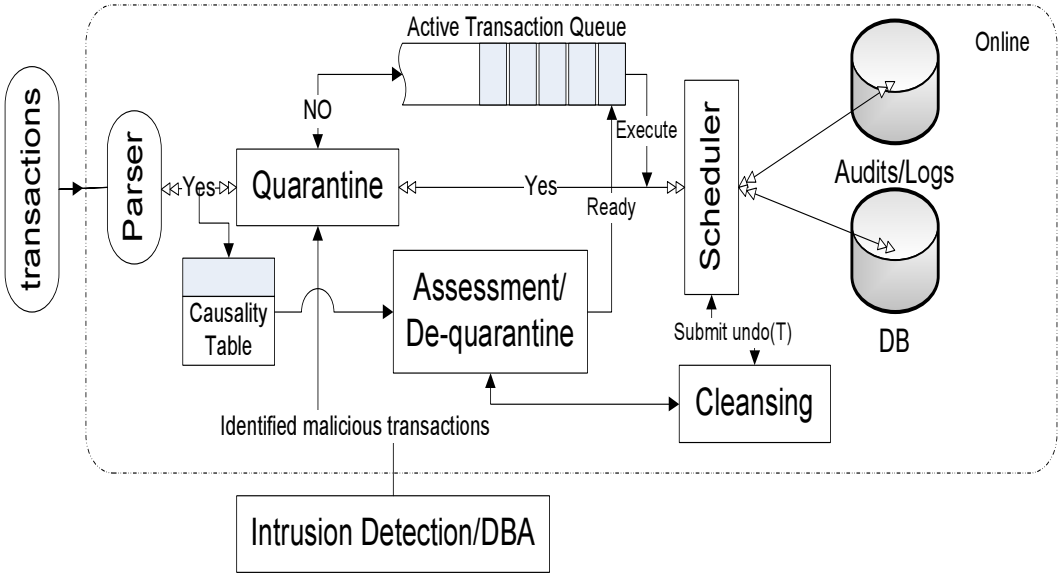


Fig. 3.2. The TRACE System Workflow

3.4.2 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 3.3) the inter-transaction dependencies to perform the

damage tracing. Figure 3.3 shows the creation of Causality Table entries of the damage spreading example in Figure 3.1. 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).

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

Fig. 3.3. An Example of the Causality Table Construction

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 transaction 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*.

In Figure 3.3, 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 3.1. 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.

To construct the Causality Table entries for each corresponding updated data record, TRACE needs to obtain tags associated with each data record, e.g. timestamp, transaction id (*xid*). Figure 3.6 gives the record layout with additional marking bytes.

3.4.3 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. TRACE uses superior technique to do such coordination so that the availability loss can be minimized. In the following, we overview how each cleansing operation functions with the focus on how they coordinate with one another.

3.4.3.1 Damage Quarantine

Damage quarantine is to prevent the normal transactions from accessing any invalid data objects, and then stop damage propagating and further reduce the repairing cost. When malicious transactions are detected by IDS, TRACE immediately sets

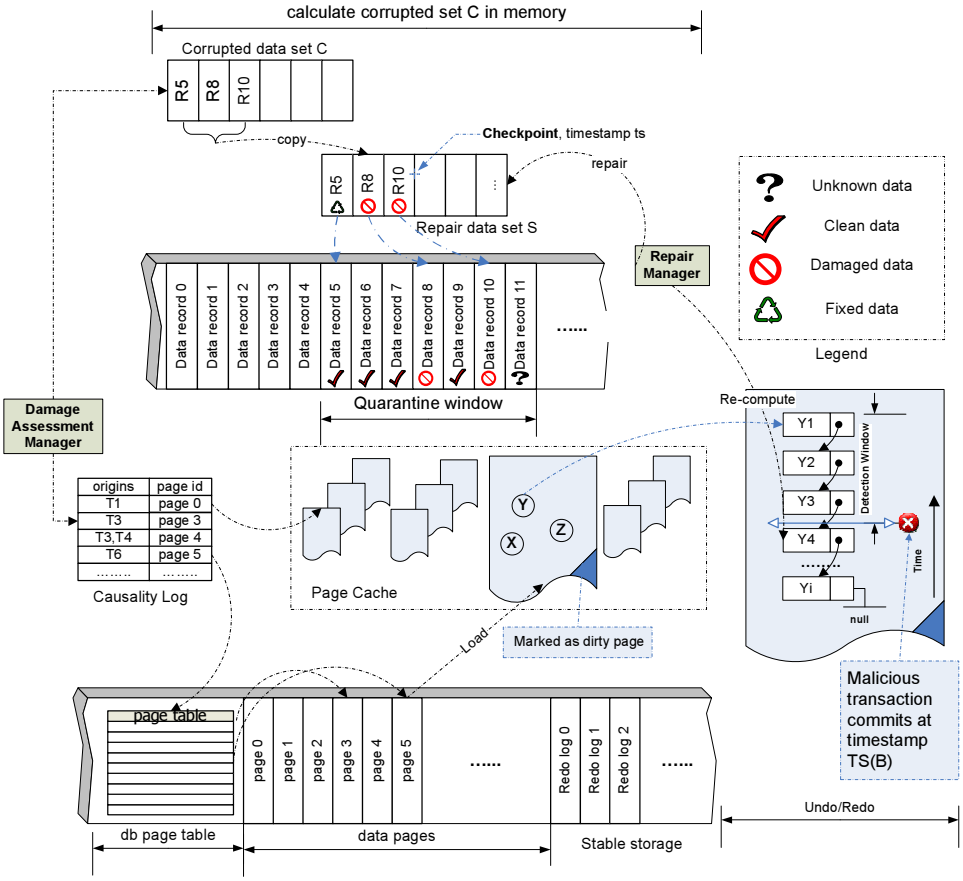


Fig. 3.4. Identify/Repair Corrupted Data Records Overview

up a time-based quarantine window (a time interval between the malicious transaction timestamp ts_b and the time TRACE receives the detection report). TRACE uses the timestamp ts_b to block the access to the data objects contained in the window. All access to the data records that are last updated later than the timestamp ts_b will be blocked. In general, after a DB is attacked, majority data objects are still valid and available. Access to the data records updated/created earlier than the timestamp ts_b will still be allowed. As a result, requirement R2 can be satisfied.

Algorithm 1: Damage Quarantine Pseudo Code

```
   Input: ts_b, t_d, T_i
 1 begin
 2   qw_s ← ts_b; qw_e ← t_d;
 3   while o_i ∈ O_T do
 4     if ts_{o_i} ∈ [qw_s,qw_e] then
 5       Deny the access of transaction T_i;
 6     else
 7       Grant the access of transaction T_i;
 8 end
```

3.4.3.2 Damage Assessment

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

The abstract damage assessment algorithm includes two steps: 1) the Intrusion Detection System (IDS) reports a malicious transaction. The malicious transaction identifier (T_b) is the initial mark of damaged data records. During scanning the causality table, TRACE knows it has found all data records corrupted by the malicious transaction T_b when it encounters an entry whose mark (tid) has an associated timestamp later than the malicious transaction timestamp ts_b. At this point, TRACE obtains the initial corrupted data set $C(T_b)$. 2) Then, TRACE processes the causality table starting from the entries whose origins set O contains the mark T_b. In general, for each entry in CT associated with transaction T_j (mark tid_j), if the entry's origins set $O_{T_j} \cap C_{T_i} \neq \emptyset$ (C_{T_i} stands for the known corrupted transactions tids in C up to T_i), TRACE puts the data records with T_j (tid_j) into corrupted set $C(T_j)$, and then adds T_i's origins set O_{T_i} into T_j's origins set O_{T_j} in CT. TRACE stops the assessment process when any one of the following two conditions is true: 1) TRACE reaches the last entry of CT; 2) TRACE reaches an entry whose timestamp is equal to the time point when TRACE starts quarantine procedure. For condition 2, any entry in CT beyond this time point is valid because only transactions request accessing to valid data are allowed to execute after the quarantine window is set up.

3.4.3.3 Valid Data De-Quarantine

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

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 ∈ CT do
3          if (o_i.ts < ts_b) && (o_i.mark==T_b) then
4              o_i → C(T_b);
5          else
6              break;
       /* calculate the corrupted data object set C(T_i)        */
7      forall o_j ∈ CT do
8          if O_{T_j}^{o_j} ∩ C_{T_i} ≠ ∅ then
9              o_j → C(T_j);
10             update O_{T_j} with O_{T_i};
11         if (∄ o_{j+1} ∈ CT) || (ts_{o_{j+1}} ≥ qw_e) then
12             break;
13 end
```

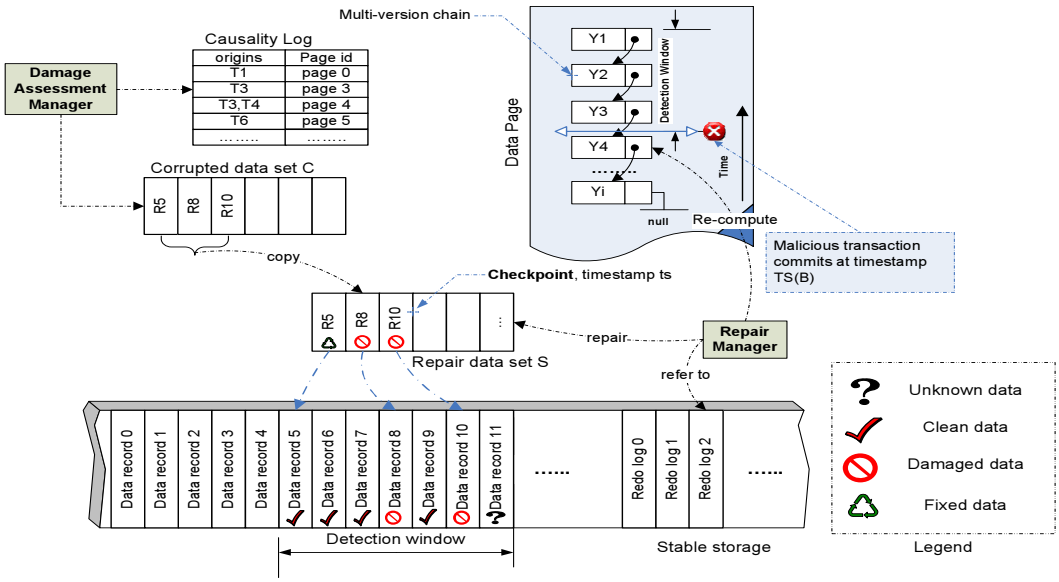


Fig. 3.5. Example of Repairing Corrupted Data Records

Because of the over-quarantined valid data records in the time-based quarantine window, TRACE needs to gradually filters 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 is not known. The submitted transaction is put into active transaction queue to wait until the data record's status is clear (e.g. the data record 11 in Figure 3.5). 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 3.5), the access is not allowed. 3) If the data record is in S_r and the status is valid (e.g., the data record 5 (R5)), the data record has been fixed and is free to access. To guarantee the correctness of condition 1), we introduce a '*checkpoint*' to the repair set S_r. Each time TRACE copies the newly identified corrupted data records in the corrupted data set C to the repair set S_r, TRACE sets a 'checkpoint' in S_r. Among the data records in S_r, there is a data record whose timestamp ts_i is the greater than others, but smaller than the 'checkpoint'. If an incoming transaction requests a data record whose timestamp is smaller than ts_i and the data object is not included in the repair set S_r at this 'checkpoint', the data record is clean and is allowed to access. This is because TRACE ensures all corrupted data records before this 'checkpoint' have been identified and copied into S_r. After a corrupted data object is fixed, the data object's status is reset to clean.

Algorithm 3: De-quarantine Valid Data Objects Pseudo Code

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

3.4.3.4 Repairing On-The-Fly

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

PostgreSQL uses a non-overwrite scheme to keep every version of updated data objects. TRACE uses this feature to look up 'before/after' images of damaged data objects without consulting the system log. Each data object o^x has a multi-version record with the form $< o^{x(v_1)}, o^{x(v_2)}, ..., o^{x(v_n)} >$, where each $v_i, (1 \leq i \leq n)$ is a version number of the data object o^x. When we find an invalid data object o^x, if the data

Algorithm 4: On-the-fly Repair Pseudo Code

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

object has been corrupted multiple times, TRACE will find a correct version of the data object and performs the undo transaction only once to remove the invalid data object. A normal transaction that needs to read/write the data object $o^{x(v_k)}$ 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 object $o^{x(v_k)}$, multi-version data objects break the dependency relations between the repairing 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.

3.5 Design and Implementation of TRACE atop PostgreSQL

To build the TRACE that offers the feature of identifying/repairing the corrupted data objects and meets the four requirements, we make several changes to the source code of a standard PostgreSQL[4] 8.1 database.

[4]http://www.postgresql.org/

3.5.1 Implementing the Marking Scheme

We maintain in *Causality Table* (CT) the inter-mark dependencies to perform the damage tracing (addressed in section 3.4.3). We create an entry for each updated data record. *Figure 3.3*(b) reflects the creation of marks as shown in Figure 3.3(a) using a simple example. Field *TransID* (the mark) has transaction identifier (*xid* in PostgreSQL) as the key of the entry. Field *Origins* is a set of mark (xid_i) indicating a data record's origins. Field *Timestamp* tells when the entry is created. We initially set the *timestamp* field with the transaction identifier, and replace it when the transaction commits. Field *PageID* indicates the page where TRACE can look up for the corresponding data record. In Figure 3.3(b), we assume transaction T_0 has no origin and calculates based on local inputs. Mark T_6 has origin T_2 and T_3. The complete *origins* list of a data record updated by transaction T_6 is $\{T_3, T_2, T_1, T_0\}$ and T_4 is $\{T_2, T_1, T_0\}$ according to Figure 3.3(a). If mark T_1 is identified as malicious, data records with attached mark T_1 are invalid. Since T_2 has T_1 in its origins, data records attached with mark T_2 are also invalid. Thus, mark T_6 is invalid too.

First, to construct the Causality Table (CT) entries for each corresponding data record, we need to obtain information associated with a data record, e.g. timestamp, transaction id (*xid*). Thus, we need to modify the data structure of the data record defined in PostgreSQL. To avoid insufficient precision and give each transaction a unique time, we extend the timestamp value with an additional four byte sequence number (SN). TRACE marking scheme additionally uses another eight byte transaction identifier to indicate the transaction that last updates the data record and one bit in the record

header to denote a data record dirty/clean status. Figure 3.6 gives the record layout with additional marking bytes. Causality table process generally has three steps corresponding with a user transaction as follows:

- *Transaction Begin.* A CT entry is generated for a data record. Initially, the transaction id is assigned and entered into the *TransID* and *Timestamp* fields.
- *Transaction In Process.* New versions of data records are generated. When the transaction reads a data record o^x, it gets the transaction id that last updated o^x and obtains the page id. TRACE updates the *origins* and *Page ID* in CT, respectively.
- *Transaction Commit.* TRACE obtains a timestamp and updates the *Timestamp* field in CT after the user transaction commits.

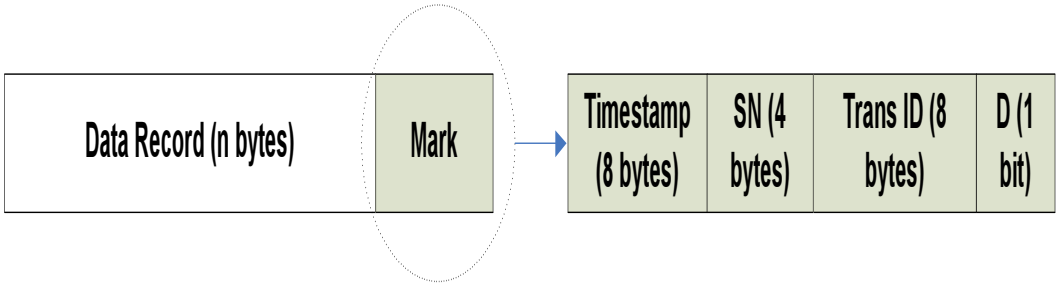


Fig. 3.6. Data Record Structure with Marking Bits

3.5.2 Maintaining Before Images

In PostgreSQL, when a data record is updated the old versions usually are not removed immediately after the *update/delete* operations. A versioning system is implemented by keeping different versions of data records in the same tablespace (e.g. in the same *data page*). Each version of a data record has several hidden attributes, such as *Xmin* (the transaction id *xid* of the transaction that generates the record version) and *Xmax* (the *xid* of the transaction that either generates a new version of this record or deletes the record). For example, when a transaction T_i performs an update on a data record o^x, it takes the valid version v of o^x, and makes a copy v^c of v. Then, it sets v's *Xmax* and v^c's *Xmin* to the transaction id xid_i. A data record is visible by a transaction if *Xmin* is valid and *Xmax* is not. Different versions of the same data record are chained by a hidden pointer as shown in Figure 3.7. TRACE utilizes these chained 'before' images to do the corrupted data record repairing (addressed in section 3.5.3). To make the hidden versions visible, we modify the index scan and sequential scan functions in *executor* module of PostgreSQL to identify the *xid* of transactions that generated (or deleted) data record versions which match our needs. Then we can go through the multi-version chain to find every historical version of a data record.

3.5.3 Damage Assessment Module

To assess the damage, TRACE locates the data record by the page id stored in CT. If the data page is not in memory, TRACE loads the data page containing the corrupted data record back into the memory (as shown in Figure 3.7). Then, TRACE traverses the associated data record version chain backwards in time (using the hidden

previous version pointer of each on-page data record, and this could span over multiple data pages) to identify every invalid data record version and the valid data record version. Within the detection window, a data record can be updated multiple times. All updates corrupted both directly and transitively by the malicious transaction T_b must have timestamps associated with them later than the timestamp ts_b. As TRACE traverses the multi-version chain, it marks every invalid version by setting the *dirty* bit until it finds a data record version whose timestamp is less than the malicious timestamp ts_b. This version-data-status-identification scheme is a simple but effective solution to mark corrupted version data because the data records corrupted during the detection window are suspicious. Thus, the versions generated during this window cannot be trusted if there is no finer grained version-data-status-identification scheme provided. In fact, TRACE can be easily extended to do finer grained version-data-status-identification, when it is in favor of the performance/complexity tradeoffs. We will address the finer grained scheme in another section.

TRACE does not keep all invalid data record versions in the repair set S_r (implemented as a hash table with the transaction id as the hashed key). For example, in Figure 3.7, 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.

3.5.4 Quarantine/De-Quarantine Module

To implement the damage quarantine in PostgreSQL, we modify the executor module source code. The plan tree of PostgreSQL is created to have an optimal execution

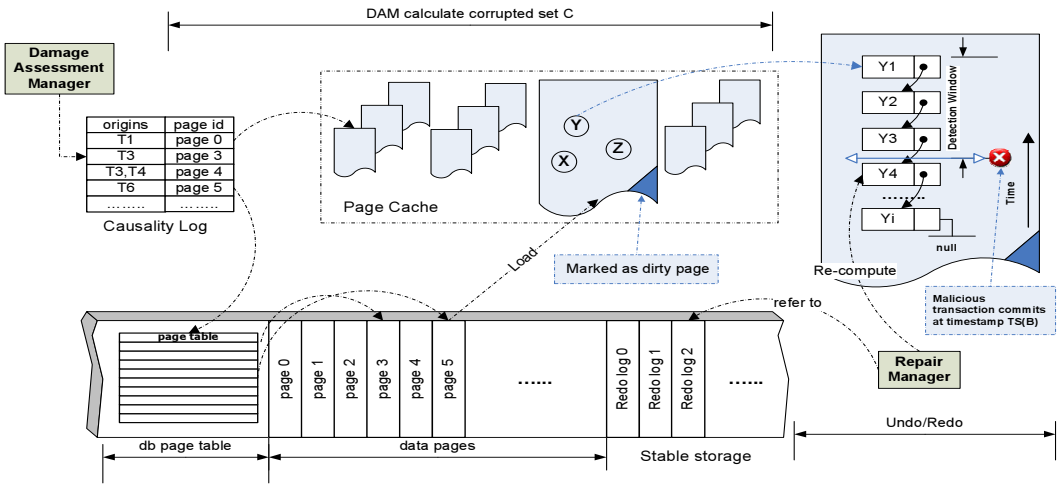


Fig. 3.7. TRACE Assessment/Repair with Causality Table in PostgreSQL

plan, which consists of a list of nodes as a pipeline. Normally, each time a node is called, it returns a data record. Starting from the root node, upper level nodes call the lower level nodes. Nodes at the bottom level perform either sequential scan or index scan. We make changes to the function of the bottom level nodes as well as the return results from the root node. By default, the executor module of PostgreSQL executes a sequential scan to open a relation, iterates over all data records. We change the executor module to check the timestamp attached to each data record while scanning the data records in the quarantine phase. If a data record satisfies the query condition and its timestamp is later than the timestamp ts_b, the executor knows the incoming transaction requests a corrupted data record. Therefore, it either discards the return result from the root node or asks the damage assessment and de-quarantine modules for further investigation, and then puts the transaction to active transaction queue to wait.

In de-quarantine phase, we modify the executor function to check whether each scanned data record from the sequential scan is already in the repair set S_r or not.

A similar change has been introduced for B-tree index scan nodes. During the normal database time, this procedure is transparent and bypassed without affecting performance. We maintain the repair set S_r as a mirror of the corrupted data set C is to enable damage assessment, de-quarantine and repairing modules run concurrently without the access conflict.

3.5.5 On-The-Fly Repairing

In this section, we present how we implement the repairing method for the identified corrupted data records in PostgreSQL. The goal is to remove the negative effects and restore the database to the previous state in the recent past. This unwinds only the work of those transactions affected by the malicious transactions.

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 3.7). 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. Due to the following reasons, TRACE will guarantee serializability: (a) all repairs are done within the quarantined area, so the repairs will not interfere the execution of new transactions; (b) our de-quarantine operations ensure serializability by doing atomic per-transitive-closure de-quarantine.

3.5.6 Garbage collection

Causality Table is a disk table that has the format $<$*TransID*, *origins*, *timestamp*, *Page ID*$>$. We build a B-tree kind index ordered by *TransID* (transaction id *xid*) on top of the causality table and maintain in the main memory, which permits fast access to the related information to assess the damage. However, if we do not remove historical entries from the causality table, it intends to become very large and the index maintained in main memory accordingly become hideous. To keep the causality table relatively small, high performance, and without losing the track of cascading effect, we garbage collect the causality table entries which are no longer of an interest of the damage assessment. As we discussed in section 3.4, 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 σ. The expected value of the detection delay is:

$$E(x) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} e^{-(x-T)^2/2\sigma^2}\, dx = T \tag{3.1}$$

and the variation of the detection delay is:

$$Var(X) = E[(X - T)^2] = \sigma^2 \tag{3.2}$$

Thus, we have the cumulative distribution function

$$F(\alpha) = \Phi(\frac{\alpha - T}{\sigma}) \tag{3.3}$$

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 is 0.0013 based on the above analysis (as shown in figure 3.8).

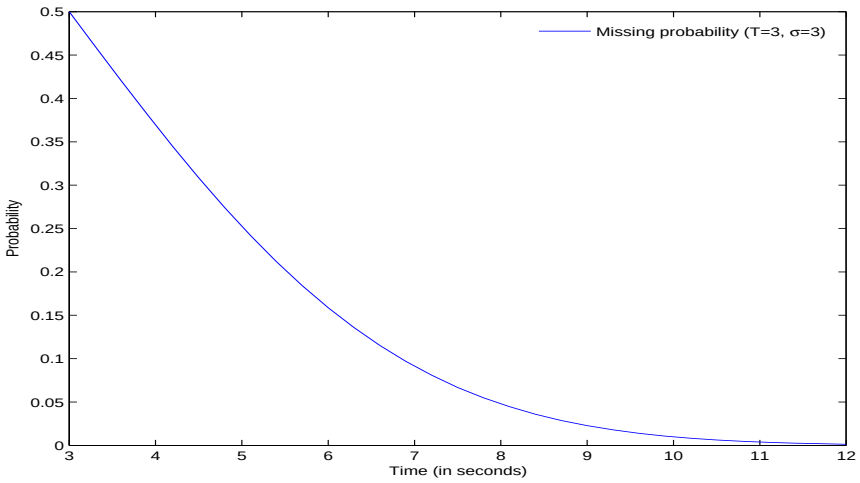


Fig. 3.8. An Example of the relationship between the process window and the missing probability

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 mark entries that stay in the causality table longer than a selected kT because the probability that the mark entry is involved in a recent negative impact to the database is small enough. In the experiments, for the sake of the simplicity, 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.

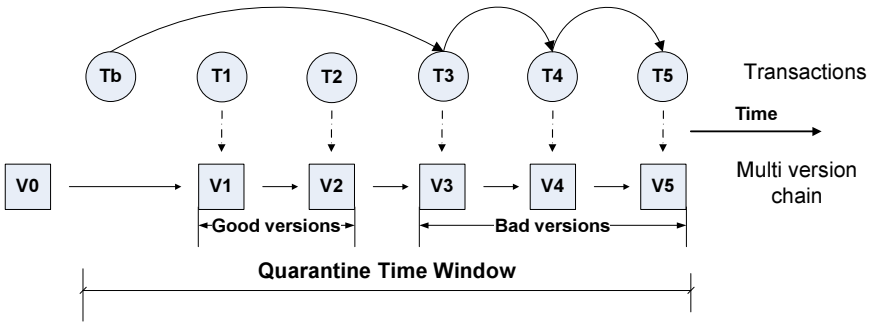
3.6 TRACE-FG: A Fine-Grained Damage Management Scheme

As we have mentioned in previous section, original TRACE system uses a time base quarantine window to block, asses the corrupted data records. It strongly relies on the assumption that all data versions generated during the quarantine window of a data record are suspicious if the quarantined data record is identified as corrupted. This simple version status identification scheme could cause 1)long system service delay, 2) substantial legitimate work loss. Thus, users may experience either data availability loss or deny-of-service.

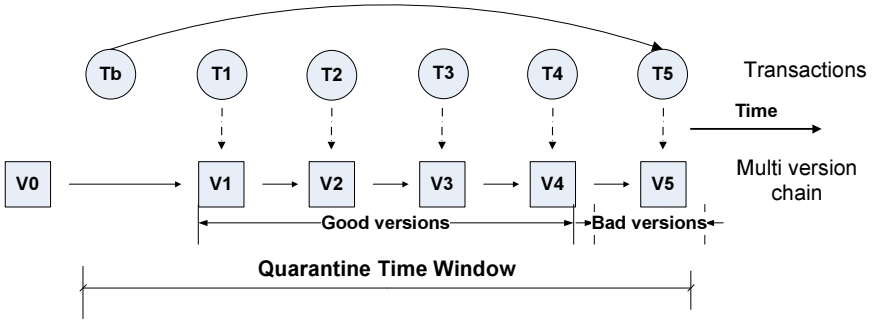
To overcome the limitations of original TRACE system, in this section, we propose an enhanced self-healing system with a fine-grained clean data version identification scheme, TRACE-FG. We elaborate the system from the following major components: damage assessment, damage De-quarantine/repair.

3.6.1 A Fine-grained Version-data-status-identification Method

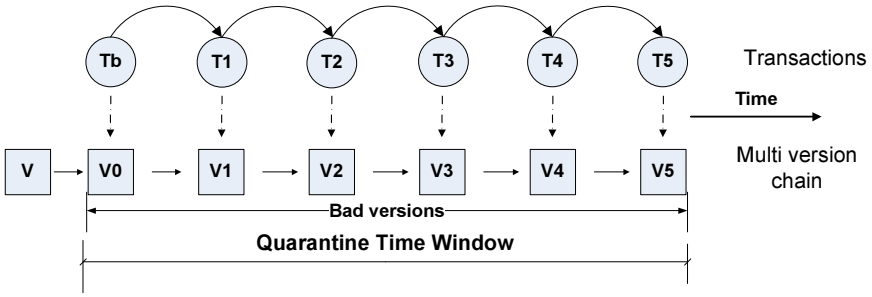
In original TRACE system, the Damage Assessment assumes that all versions of the corrupted data records within the quarantine window are also corrupted. Then, when repairing the corrupted data records, TRACE needs to traverse backwards the version chain to identify the right 'before image', which generally is the one whose timestamp ts is immediately smaller than the malicious transaction's ts_b. However, this assumption heavily reduces the legitimate work that could have been saved. Let us consider the scenario shown in the Figure 3.9(a) as an example. Within the quarantine time window, 5 versions (v_1-v_5) are generated by transactions $T_1 - T_5$. Let us assume that the malicious



(a) Case 1: Average Case Scenario



(b) Case 2: Best Case Scenario



(c) Case 3: Worst Case Scenario

Fig. 3.9. Cases of Assessment Procedure Limitations

transaction with timestamp ts_b has affected transaction T_3 at time ts_{T_3}, T_4 and T_5, respectively. Let us also assume that timestamp ts_{T_3} is 2 minutes later than ts_b. During the 2 minutes, 2 versions of v_0 have been produced. It is clear that these two data versions are clean, but TRACE will treat the 2 clean versions as dirty. Figure 3.9(b) shows another example in which TRACE will consider the entire version chain (v_0-v_5) as dirty. If a fine-grained version-data-status-identification scheme is provided, more legitimate work can be saved (such as using v_2 in Figure 3.9(a) and v_4 in Figure 3.9(b) as the right 'before image' instead of v_0).

Based on the example shown in Figure 3.9(a), we propose a simple solution of version-data-status-identification. Suppose the Damage Assessment module identifies the data record x (version v_5) is indirectly corrupted by the malicious transaction T_b through the transaction T_5. TRACE-FG then looks up the multi version chain backwards. It first checks the data version 4, and knows the transaction T_4 is the one who generates version 4. To determine if this version is the right 'before image' of the corrupted data version 5, TRACE-FG needs to check if the transaction T_4's origin set contains the malicious transaction T_b (such information is stored in the causality table). For case 1 shown in the Figure 3.9(a), the arrow on the transaction T_4 indicates the transaction T_4 has already been corrupted by the malicious transaction T_b, then TRACE-FG will go backwards further to check the version 3 and version 2 respectively. When TRACE-FG checks the version 2, it finds that the origin set of transaction T_2 does not contain T_b. TRACE-FG instantly knows that the version 2 is the correct 'before image' of the data record x, and then stop further check up. The case 1 shown in the Figure 3.9(a) is the average case scenario of this solution. The case 2 shown in the Figure 3.9(b) demonstrates the

best case scenario in which TRACE-FG only needs to go backwards one step along the multi version chain. The case 3 shown in the Figure 3.9(c) demonstrates the worst case scenario in which TRACE-FG needs to check up every version data until it reaches the original version v_0. The worse case scenario is actually the time based quarantine window approach. Ideally, we can see that this simple solution can save more legitimate work in term of the correct data version that should be restored. We summarize the solution in Algorithm 5.

Algorithm 5: Fine-grained Version Status Checking Pseudo Code

```
   Input: V, T_b, qw_e
1  begin
       /* check up every version v in the multi version chain V
          */
2      while (v_i ← V) ≠ null && (v_i.ts >= ts_b) do
3          if O_{T_i}^{v_i} ∩ T_b ≠ ∅ then
4              i++; continue;
5          else
6              break;
7      return v_i;
8  end
```

3.6.2 Fine-grained De-quarantine and Repair

Like the original TRACE system, TRACE-FG treat the read-only transactions and read-write transactions separately. For the read-only transactions, if the requested data record is corrupted, TRACE-FG will de-quarantine a right data version to the read-only transactions. For example, considering the case shown in Figure 3.9(a), TRACE-FG will pick up the version v_2 and release it to the transactions. It is clear that this chosen

version v_2 will become the 'current' version after repairing. Thus, releasing it prior to the repair procedure is harmless. To repair the corrupted data record, TRACE-FG issues an undo transaction piggybacked with the chosen version v_2 as the right 'before-image'. For read-write transactions, TRACE-FG responses more carefully. If the write set of the read-write transaction contains any corrupted data records, TRACE-FG holds up the transaction in the ATQ until the requested data records are repaired by undo operations. If only the read set of the read-write transaction contains corrupted data records, TRACE-FG treats it as a read-only transaction and grants the access with right version data. By this way, TRACE-FG can not only achieve high system throughput as the system being attacked, but also ensure high data availability.

3.7 Experimental Results

3.7.1 Evaluation of TRACE System

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

We construct two database applications based on the TPC-C benchmark and the clinic OLTP. The OLTP application defines 119 tables with over 1140 attributes belonging to 9 different sub-routines. In this work, we use 2 of the 9 sub-routines which contain 10 tables and over 100 attributes (refer to the Chapter 3.9.2 for details). For more detailed description of TPC-C benchmark and the OLTP application we refer the reader to [25, 78]. Transaction workloads are based on above two applications. A transaction includes both read and write operations. The experiments conducted in this paper run on Debian GNU/Linux with Intel Core Due Processors 2400GHz, 1GB of RAM. We choose PostgreSQL 8.1 as the host database system and compile it with GCC 4.1.2. The TRACE subsystem is implemented using C.

Parameters	Values
Number of warehouses	15
Districts per warehouse	30
Clients per district	5000
Items per warehouse	100000
Orders per district	5000

Table 3.2. TPC-C Parameters and Their Values

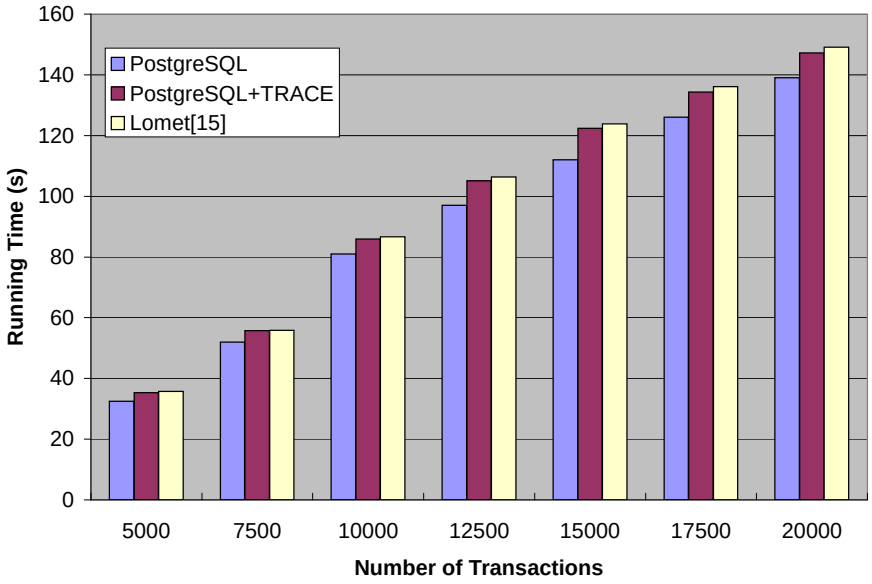
3.7.1.1 System Overhead

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 each application. For TPC-C application, we set up

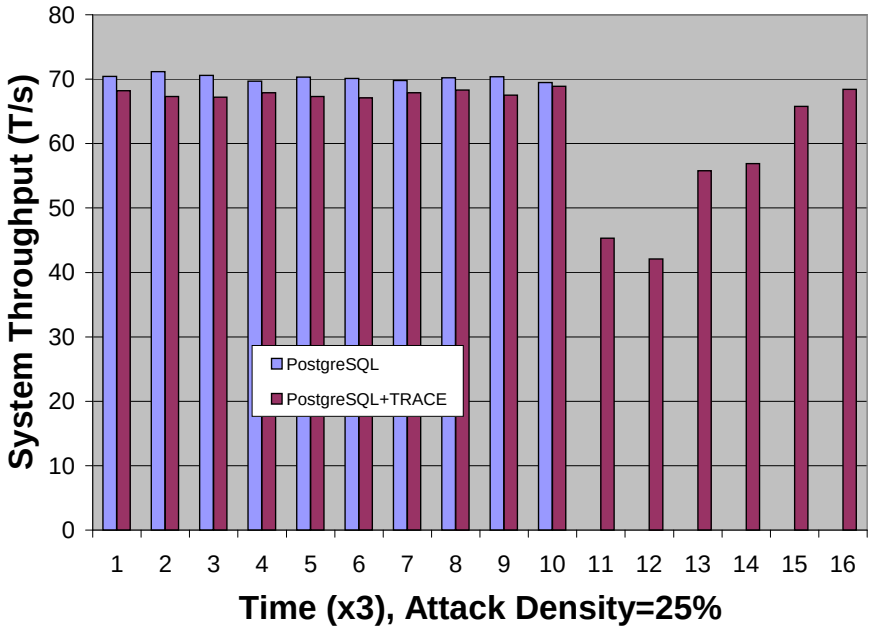
each transaction containing no more than 5 update statements. Figure 3.10(a) shows the comparison of system overhead of TRACE and the raw PostgreSQL system on TPC-C based application. Because TRACE provides additional functionalities, it has system overhead on the PostgreSQL by the size of transaction 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 [44]. The results shown in the Figure 3.10(a) demonstrate that two methods are comparable and do not cause significant system overhead. In comparison with [2], which adds approximate 25%-35% run time overhead to the system, TRACE achieves a great improvement.

3.7.1.2 Zero System Down Time

To evaluate the performance of TRACE on sustaining a good data service while recovering, we demonstrate in Figure 3.18 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



(a) Run Time Overhead



(b) Throughput of TRACE vs. PIT

Fig. 3.10. Evaluation of TRACE System

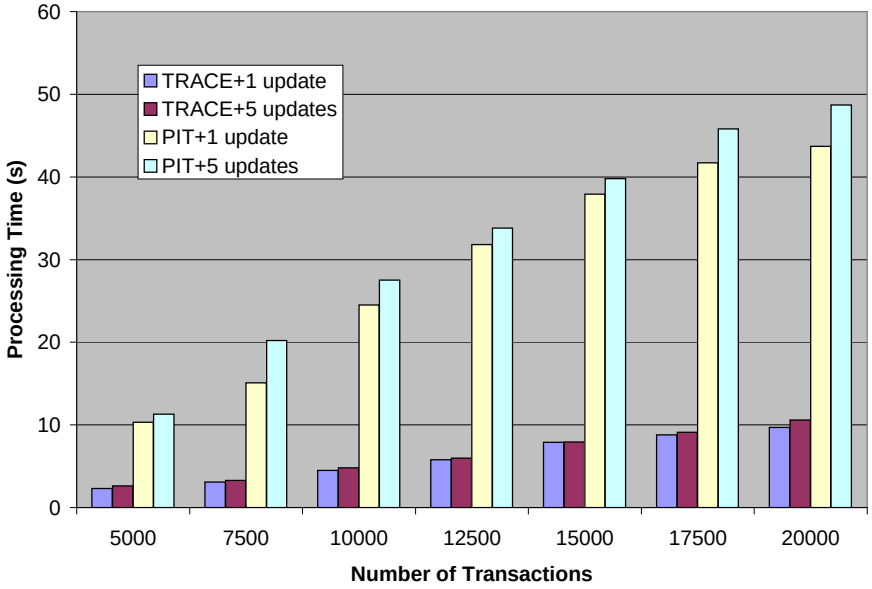
T_y, transaction T_x will still be filtered out (held in the active transaction queue) if transaction T_y is filtered out due to accessing compromised data because the result directly from transaction T_y is dirty. In Figure 3.18, we present an approximate 40 seconds system running-time window. Until 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.

3.7.1.3 TRACE vs. 'point-in-time' recovery

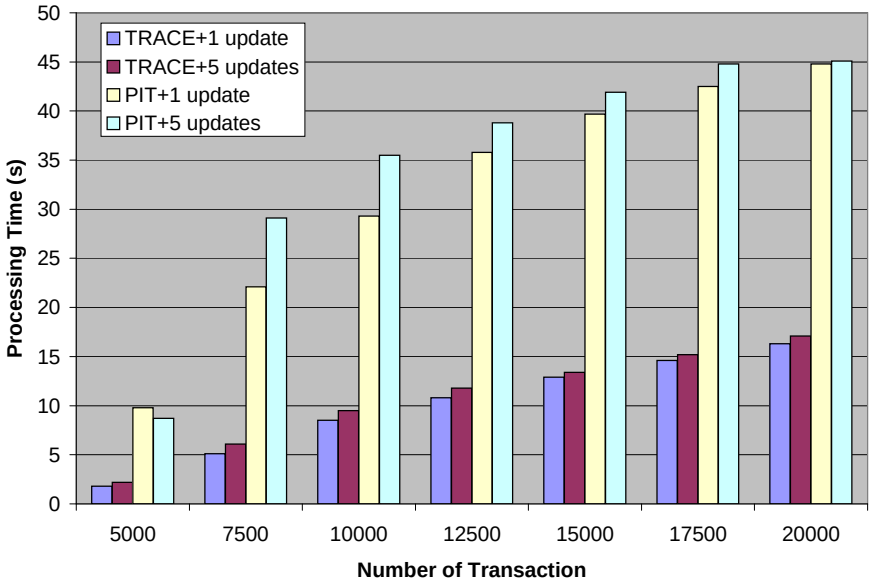
Next, we evaluate the performance of TRACE cleansing procedure and the ordinary PostgreSQL 'point-in-time' recovery procedure. We perform two experiments with 20,000 transactions having different ratios of corrupted data records within the database

system, lightly damaged (15% of the total data records, 15,000/100,000–corrupted/total), heavily damaged (35% of the total data records, 35,000/100,000–corrupted/total).

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 3.11(a) and Figure 3.11(b) 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 remove the negative effects caused by malicious transactions and the affected transactions, and then de-commit much less transactions than the standard recovery procedure applied in



(a) TRACE vs. PIT Recovery on Light Damage



(b) TRACE vs. PIT Recovery on Heavy Damage

Fig. 3.11. Evaluation of TRACE System

PostgreSQL. Thus, TRACE recovery mechanism saves a great amount of system down time (up to 80% of the system down time is saved).

3.7.1.4 System CPU Time Distribution

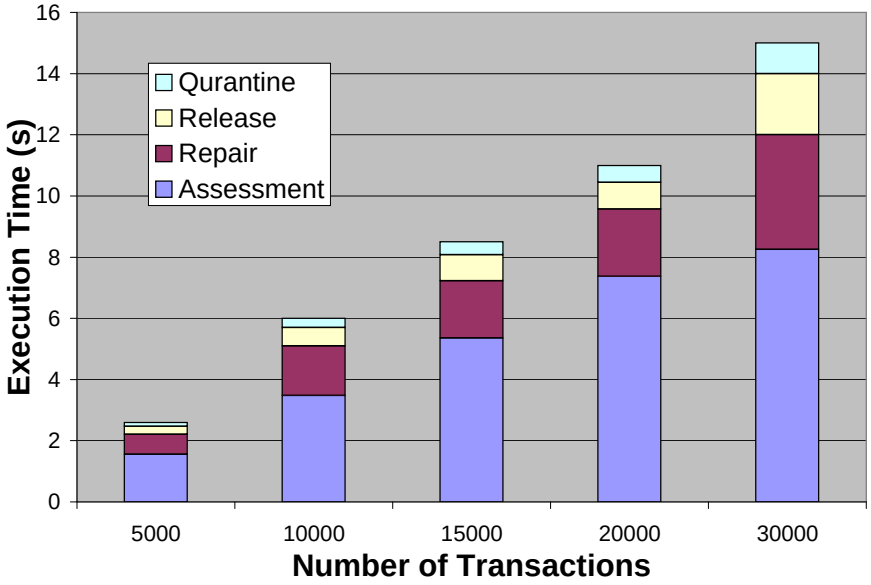
To understand how well each key component of TRACE collaborate with others, we evaluate the TRACE components by studying the CPU time occupied by each component. We run the experiment 5 times with the attacking density d=10% (25%), and 5000, 10000, 15000, 20000, and 30000 transactions, respectively. We demonstrate in Figure 3.12(a) 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 3.12(b) 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.

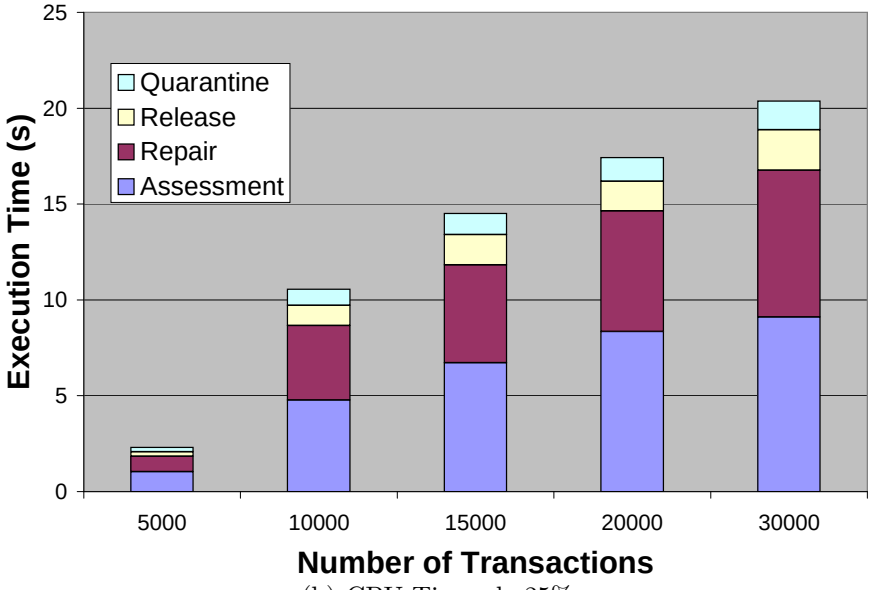
3.7.1.5 Reduced Service Delay Time and Saved Legitimate Transactions

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 OLTP application to obtain stable results.

The experimental results are shown in Figure 3.13(a) and Figure 3.13(b). Figure 3.13(a) 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

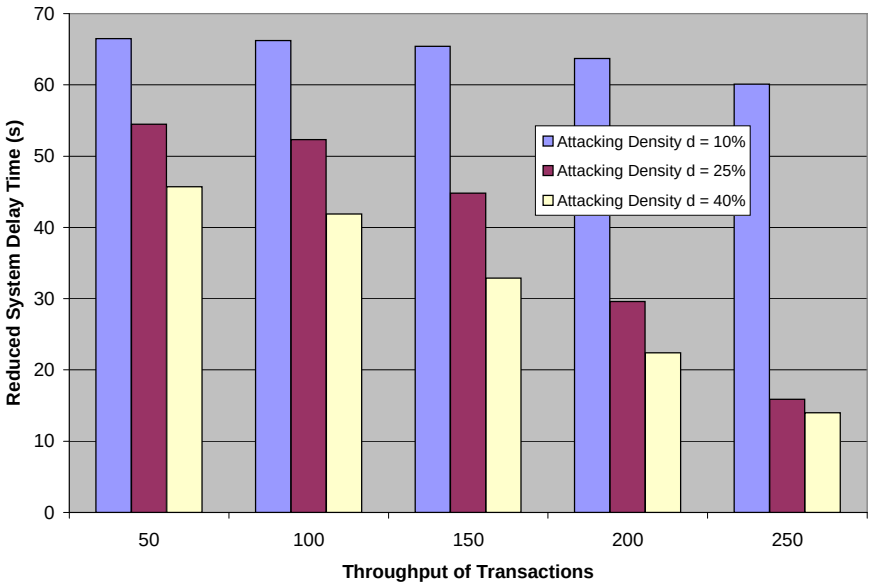


(a) CPU Time, d=10%

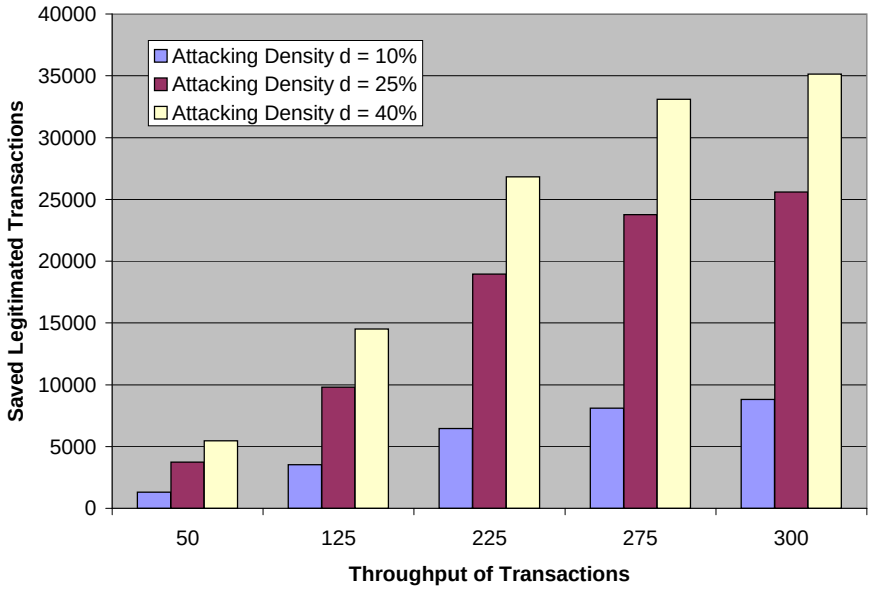


(b) CPU Time, d=25%

Fig. 3.12. Evaluation of TRACE System



(a) Reduced System Service Delay Time with Different d



(b) Reduced # of De-Committed Transactions with Different d

Fig. 3.13. Evaluation of TRACE System

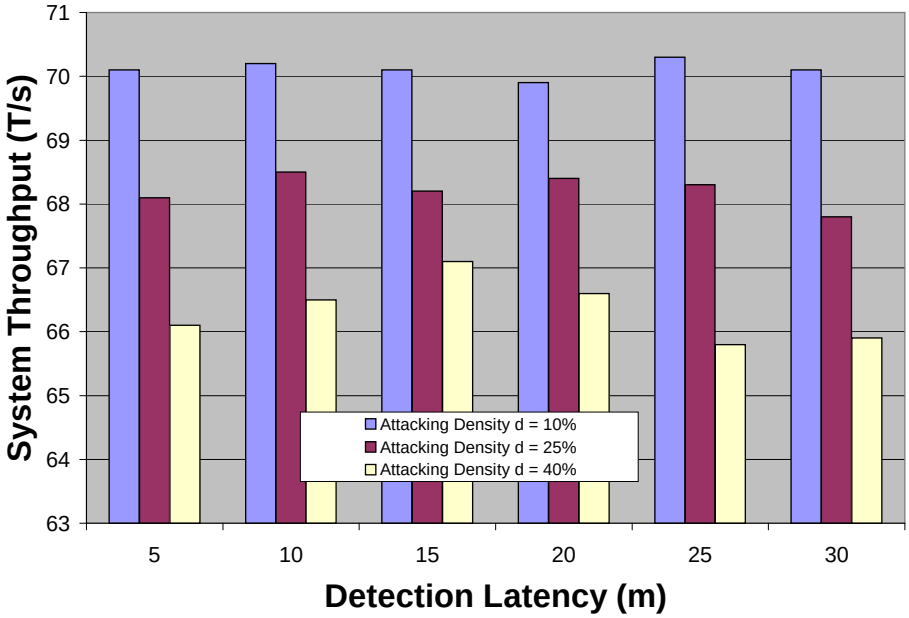
spends less time to analyze the causality table and has much less corrupted data records to repair. As the d and throughput increase, TRACE causes the database system running busy in identifying the corrupted data records. However, even the percentage decreases, TRACE still saves a great amount of system outage time and makes the system stay online. Figure 3.13(b) 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.

3.7.1.6 IDS Impacts on TRACE System

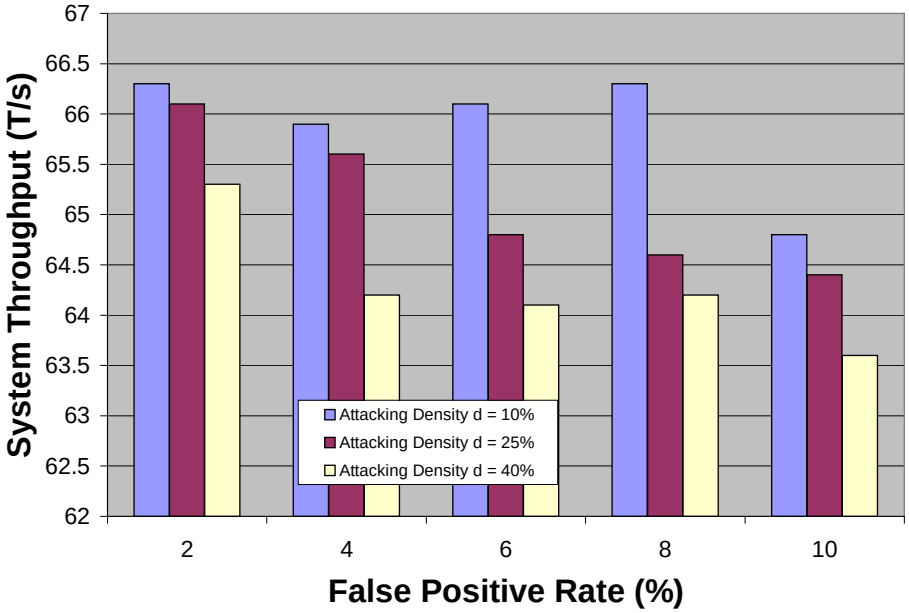
When we evaluate the performance of TRACE on the requirement R3 and R4, we also consider the impact of the detection latency and the false positive rate of the IDS. To study the IDS impact on TRACE, we implement the system proposed in [37]. We demonstrate in Figure 3.15(a) and **??** the detection latency impact on TRACE, in Figure 3.15(b) and **??** the false positive rate impact on TRACE, respectively. Figure 3.15(a) 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 3.15(a), 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 3.15(b) 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 **??** 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 **??** 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



(a) DL on Read-Only Transaction



(b) FPR on Read-Only Transaction

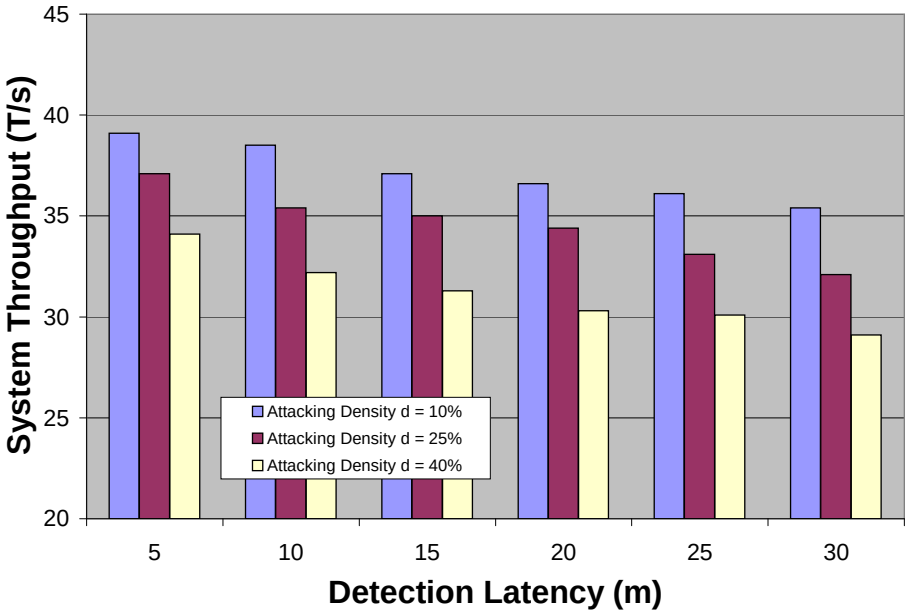
Fig. 3.14. Evaluation of TRACE System

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.

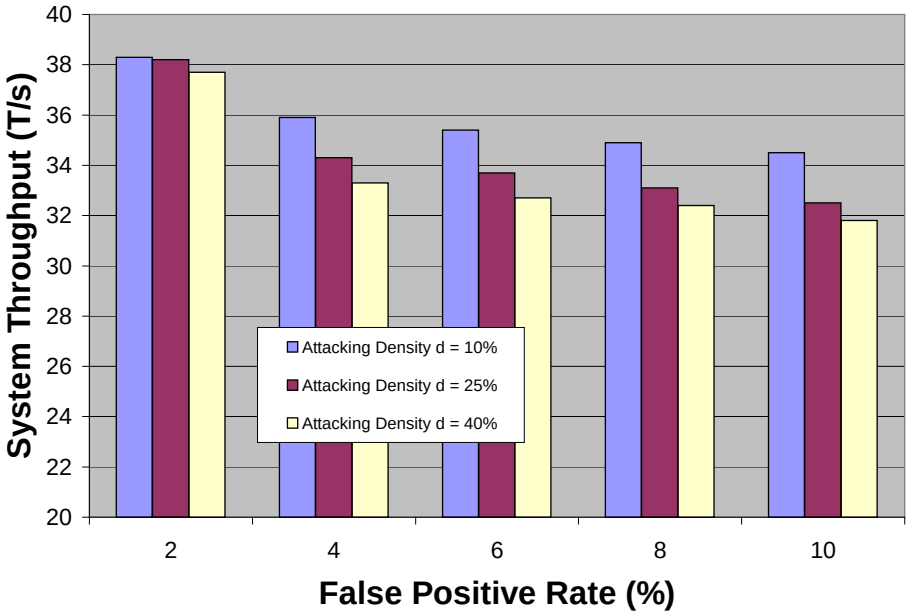
3.7.2 Evaluation of TRACE-FG System

We evaluate the fine-grained version status checking solution under the TRACE subsystem in PostgreSQL database system based on the TPC-C benchmark [25]. To measure the how well the proposed approach ensures the high level assurance of the database system, we focus on the data availability improvement of TRACE-FG over the original TRACE system. In addition, since both TRACE and TRACE-FG can restore the corrupted data records to a clean version, the data integrity improvement may not be clearly seen in a straightforward manner. We present the experimental results based on the following evaluation metrics: M1) Saved Data Versions; M2) Reduced System Service Delay Time; M3) Saved De-committed Transaction; M4) Improved System Throughput. First, we demonstrate the performance of the fine-grained scheme in terms of the saved data versions. Second, we demonstrate the comparison of TRACE-FG, TRACE and the standard *'point-in-time'* (PIT) recovery method in terms of the reduced system service delay, saved de-committed transactions, and system throughput.

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 [25]. A transaction includes both read and write operations. The experiments conducted in this paper run on Debian GNU/Linux



(a) DL on Read-Write Transaction



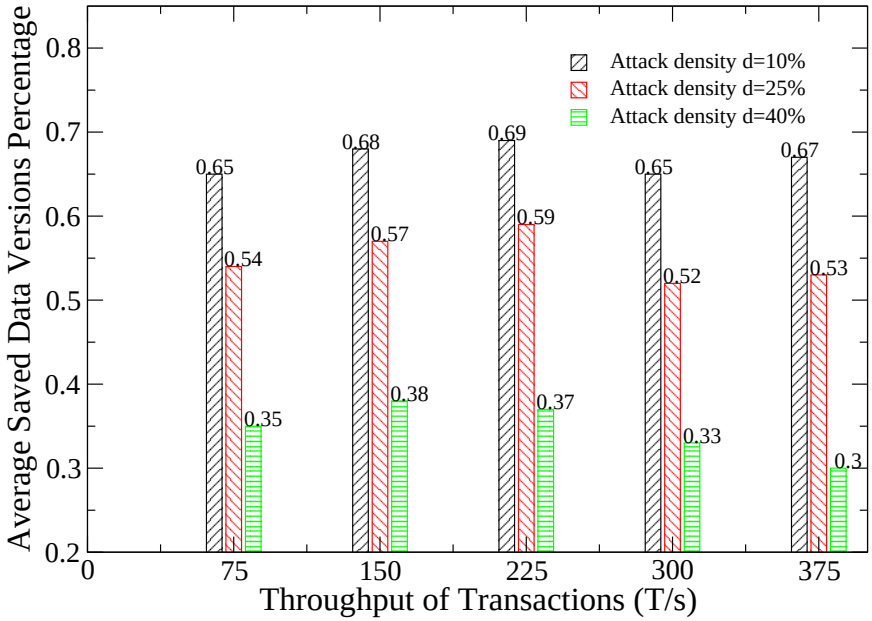
(b) FPR on Read-Write Transaction

Fig. 3.15. Evaluation of TRACE System

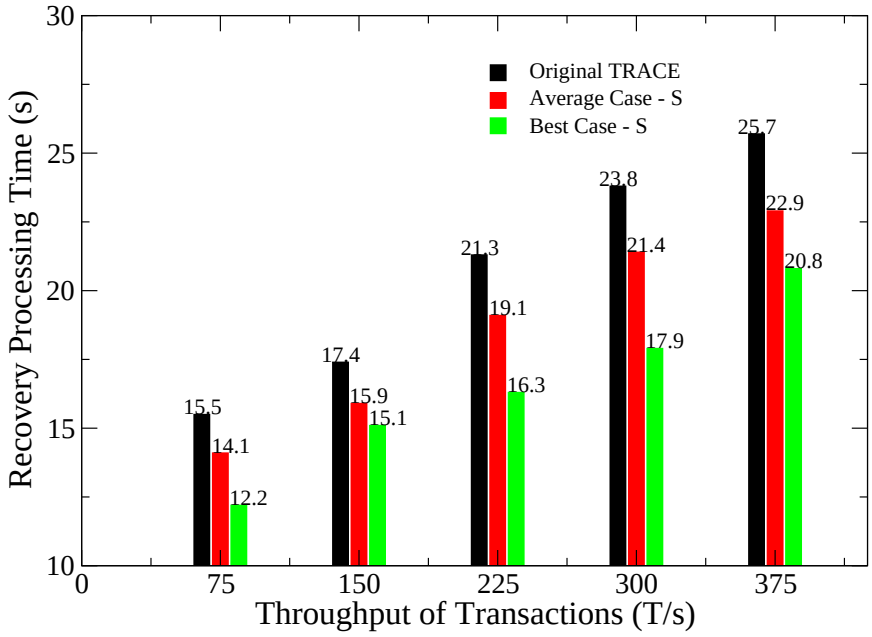
with Intel Core Due Processors 2400GHz, 1GB of RAM. We choose PostgreSQL 8.1 as the host database system and compile it with GCC 4.1.2. The TRACE subsystem is implemented using C.

3.7.2.1 Saved Data Versions

When we evaluate the performance of the fine-grained version status solution, we generate the workload according to the following criteria: 1) we restrict that each data record has at most 10 updates, which provides a fairly long version chain; 2) we set each transaction with at most 5 update statements. We demonstrate in Figure 3.16(a) the evaluation of the saved data versions. We define the saved data versions as the number of clean data versions identified. For example, considering the case 1 in the Figure 3.9(a), the number of saved clean data version is 2, the number of saved clean data version for case 2 shown in the Figure 3.9(b) is 4, and 0 for the case 3, respectively. Figure 3.16(a) shows the percentage of the average saved data versions at different throughput setting with respect to different attack density (d = 10%, 25%, and 40%). We define the percentage of the average saved data version as the total saved data versions during the entire recovery process over the total recovered data records. For example, for the case 1 in the Figure 3.9(a), the percentage of saved data versions is 40% (total 100 data versions identified, 40 clean data are saved). It is clear that more data versions are saved by the proposed solution along the X-axis. For each attack density setting, the system can maintain a relative stable percentage of saved data versions. At each transaction throughput setting, as the attack density d increase, less percentage of data versions are saved compared with light attack density.

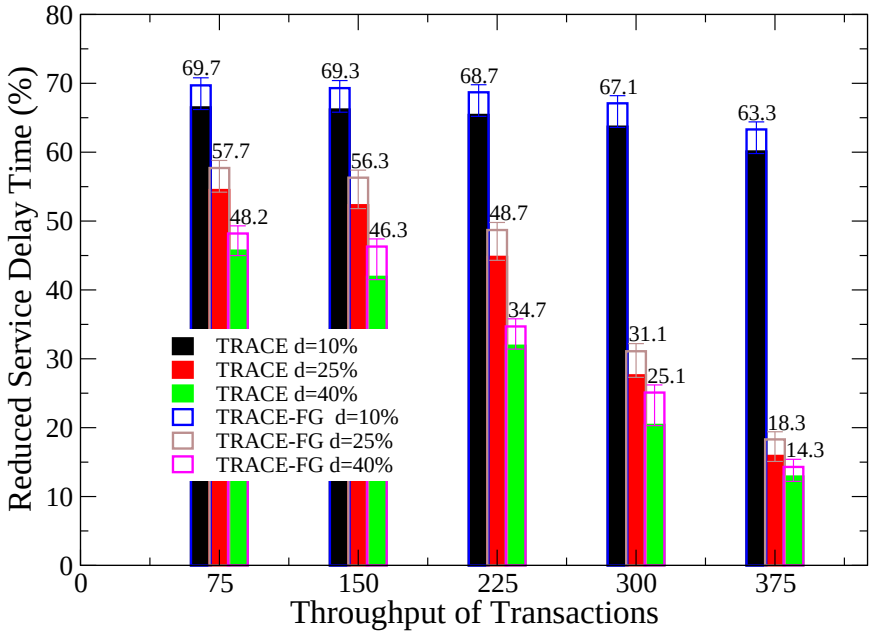


(a) Saved Data Versions

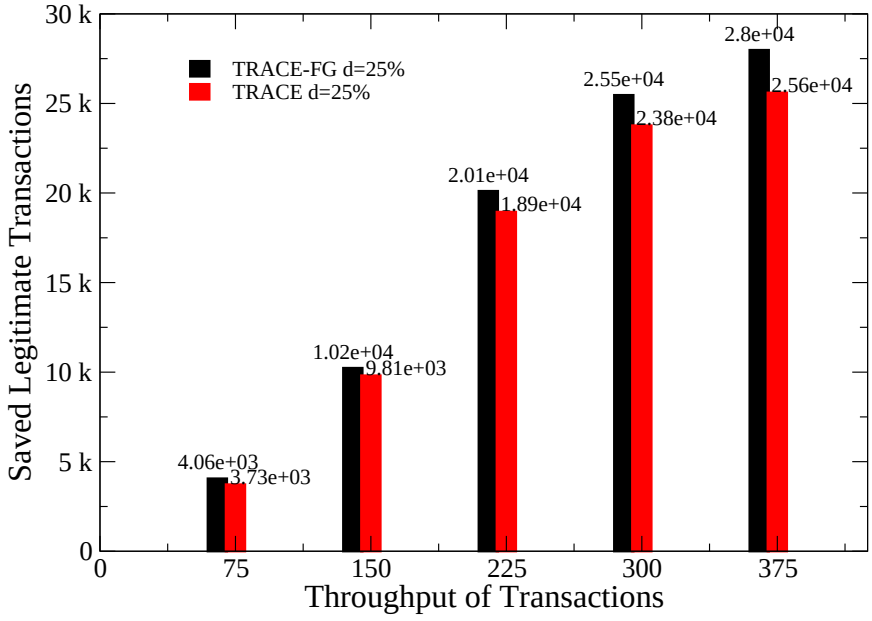


(b) System Recovery Time

Fig. 3.16. Evaluation of TRACE-FG System



(a) Reduced Service Delay Time with Different d



(b) Saved Legitimate Transactions d=%25

Fig. 3.17. Evaluation of TRACE-FG System

We also demonstrate in Figure 3.16(b) the performance of the proposed version status solution in terms of the recovery processing time. We generate the workload based on the same criteria shown above (e.g., each data record has at most 10 updates). With different throughput setting on X-axis, it is clear to see that the fine-grained version status solution outperforms the original TRACE system on average case (around 10% improvement on the system recovery). In the base case scenario, TRACE-FG can achieve 19% improvement compared with the original TRACE system. However, the best case scenario is extreme and may not happen in the real world all the time. The red bar in the Figure 3.16(b) indicates the average case, and the green bar indicates the best case of the proposed solution.

3.7.2.2 Reduced Delay Time/Saved De-committed Transactions

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

malicious/affected 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 3.17(a) and Figure 3.17(b). Figure 3.17(a) shows the reduced system service delay time w.r.t. different attacking density and throughput. We observe that, with TRACE-FG 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 as the attacking density d increases. The reason is when the attacking density and throughput is light, TRACE-FG, like the original TRACE system, spends less time to analyze the causality table and has much less corrupted data records to repair. As the d and throughput increase, TRACE-FG causes the database system running busy in identifying the corrupted data records. However, even the overall percentage decreases, TRACE-FG outperform the original TRACE (the white box indicates TRACE-FG achieves 5% improvement over TRACE on average) and saves a great amount of system outage time.

Figure 3.17(b) shows the saved De-committed transactions when the attacking density d is 25% and w.r.t. different throughput. We observe that TRACE-FG can save significant legitimated transaction work. The overall number of legitimate transactions that have been saved by TRACE-FG increases as the system throughput increases. The reason is when the attacking density and throughput is light, less data records are corrupted. TRACE-FG can save more innocent transactions than the original TRACE system (around 6% more over TRACE on average), 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.

3.7.2.3 TRACE-FG vs. TRACE and 'point-in-time' on System Throughput

To evaluate the performance of TRACE-FG on the metric M4, we demonstrate in Figure 3.18 the system throughput of the PostgreSQL with/without TRACE/TRACE-FG 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 3.18, 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/TRACE-FG because they 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/TRACE-FG, the system carries 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 doing the damage quarantine/assessment/repair procedure. During this partial time window (point 11 to point 16), the database armed with TRACE-FG can still achieve higher system throughput than TRACE. Overall, the goal of continuing providing service when

the system is under an attack is met with satisfactory system throughput performance. In addition, TRACE-FG outperforms the original TRACE system in terms of the system throug

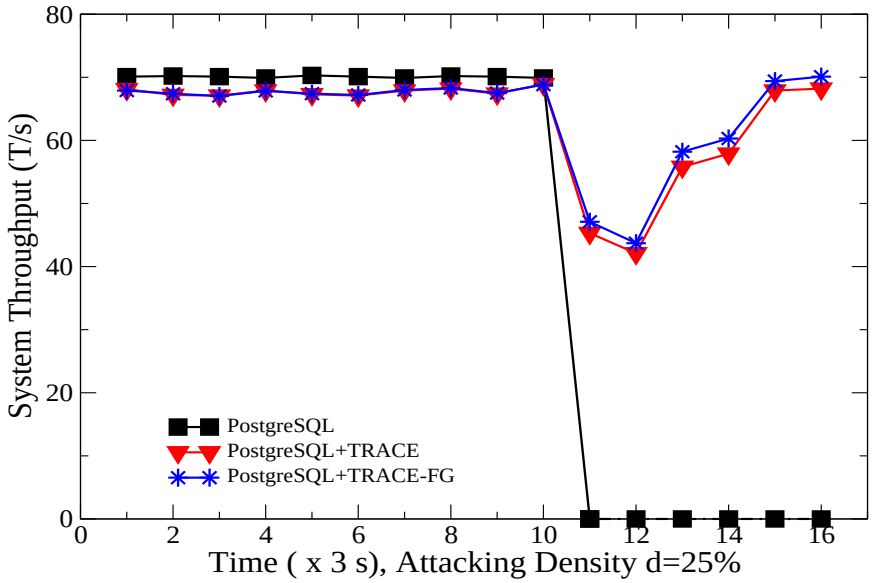


Fig. 3.18. System Throughput of TRACE-FG vs. TRACE and 'PIT' with d=%25

3.8 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 3.4.1, the impact of flase 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 let 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.

3.9 Appendix

3.9.1 TPC-C Transaction Read/Write Set Template

In TPC-C, the term database transaction as used in the specification refers to a unit of work on the database with full ACID properties, namely atomicity, consistency, isolation, and durability. A business transaction is composed of one or more database transactions. In TPC-C, a total of five types of business transactions are used to model the processing of an order (see [25] for the source codes of these transactions). The read and write Set templates of these transaction types are as follows.

- The **New-Order** transaction consists of entering a complete order through a single database transaction. The template for this type of transaction is:

  Input = warehouse number(w_id), district number(d_id), customer number(c_id);

  items id(ol_i_id), supply warehouses(ol_supply_w_id), and quantities(ol_quantity)

  Read_Set = { Warehouse.w_id.W_TAX;

  District.(w_id+d_id).(D_TAX, D_NEXT_O_ID);

Customer.(w_id+d_id+c_id).(C_DISCOUNT, C_LAST, C_CREDIT);

Item.ol_i_id.(I_PRICE, L_NAME, L_DATA);

Stock.(ol_supply_w_id+ol_i_id).(S_QUANTITY, S_DIST_xx,

S_DATA, S_YTD, S_ORDER_CNT, S_REMOTE_CNT) }

Write_Set = { x=District.(w_id+d_id).D_NEXT_O_ID; New-Order.(w_id+d_id+x);

Order.(w_id+d_id+x); R_1={ ol_i_id }; Order-Line.(w_id+d_id+x+R_1) }

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

  Input = warehouse number(w_id), district number(d_id), customer number(c_w_id, c_d_id, c_id) or customer last name(c_last), and payment amount(h_amount)

  Read_Set = { Warehouse.w_id.(W_NAME, W_STREET_1, W_STREET_2, W_STATE, W_YTD);

  District.(w_id+d_id).(D_NAME, D_STREET_1, D_STREET_2, D_CITY, D_STATE, D_ZIP, D_YTD); Customer.(c_w_id+c_d_id+c_id).(C_FIRST, C_LAST, C_STREET_1, C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_SINCE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_DATA);}

  Write_Set = { Warehouse.w_id.W_YTD; District.(w_id+d_id).D_YTD;

  Customer.(c_w_id+c_d_id+c_id).(C_BALANCE, C_YTD_PAYMENT, C_PAYMENT, C_PAYMENT_CNT); History.(c_id+c_d_id+c_w_id+d_id+w_id).*}

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

  Input = customer number(w_id+d_id+c_id)

  Read_Set = { Customer.(w_id+d_id+c_id).(C_BALANCE, C_FIRST, C_LAST, C_MIDDLE); x=Order.(w_id+d_id+c_id).O_ID;
  Order.(w_id+d_id+c_id).(O_ENTRY_D, O_CARRIER_ID);
  Order-line.(w_id+d_id+x).(OL_I_ID, OL_SUPPLY_W_ID,
  OL_QUANTITY, OL_AMOUNT, OL_DELIVERY_D)}

  Write_Set = {}

- The **Delivery** transaction processes ten new orders within one or more database transactions. The templates for this type of transaction are:

  Input = warehouse number(w_id), district number(d_id),
  and carrier number(o_carrier_id)

  Read_Set = { R_1=New-Order.(w_id+d_id).NO_O_ID;
  R_2=Order.(w_id+d_id+R_1).O_C_I);
  Order.(w_id+d_id+R_1).(O_CARRIER_ID, OL_DELIVERY_D, OL_AMOUNT);
  Customer.(w_id+d_id+R_2).(C_BALANCE, C_DELIVERY_CNT) }

  Write_Set = { R_1=New-Order.(w_id+d_id).NO_O_ID;
  R_2=Order.(w_id+d_id+x).O_C_ID; Order.(w_id+d_id+R_1).O_CARRIER_ID;
  Customer.(w_id+d_id+R_2).(C_BALANCE, C_DELIVERY_CNT);
  New-Order.(w_id+d_id+R_1); Order-Line.(w_id+d_id+R_1).OL_DELIVERT_D }

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

  Read_Set = { x=District.(w_id+d_id).D_NEXT_O_ID; R_1={x-1,..., x-19, x-20};

  R_2=Order-Line.(w_id+d_id+R_1+OL_NUMBER).OL_I_ID;

  Stock.(w_id+R_2).S_QUANTITY }

  Write_Set = {}

3.9.2 Clinic OLTP Application Transaction Read/Write Template

To test our ideas in the project, we use a clinic OLTP application as another test bed. The clinic is a private physiotherapy clinic located in Toronto. It has five branches across the city. It provides services such as joint and spinal manipulation and mobilization, post-operative rehabilitation, personal exercise programs and exercise classes, massage and acupuncture. In each day, the client applications installed in the branches make connections to the center database server, which is Microsoft SQL Server 7.0. In each connection, a user may perform one or more tasks, such as checking in patients, making appointments, displaying treatment schedules, explaining treatment procedures and selling products. The database trace log contains 81,417 events belonging to 9 different applications, such as front-end sales, daily report, monthly report, data backup, and system administration. Our target applications are the two front-end sales applications. We extract the transaction templates from the trace log and re-construct the two applications in our PostgreSQL server. The read and write Set templates of these transaction types are as follows.

- The **Schedule** transaction processes a client's request of scheduling the physical treatment. The templates for this type of transaction are:

  Input = customer number(customer_id); Contract Type(con_t); Branch number(b_id)

  Read_Set = { Contract.(contract_data+ contract_no+outstanding_balance); MemberCard.(card_name+card_level+accum_amount+one_time_amount +free_membership+p_discount+t_discount); TreatmentSchedule.(check_in_date +start_time+contract_no+b_id+contract_no+treatment_id); PostDatedRecord.(payment_method+is_credit+pay_date+pay_amount+b_id); InvoiceNo.(invoice_num);InvoicePay.(receipt_id+pay_amount); }

  Write_Set = { contract.(used_count); customer.(total_sales+year_accu_amount); InvoicePay.(receipt_id); TreatmentSchedule.(treatment_status)}

- The **Product** transaction processes a client's treatment and order information. The templates for this type of transaction are:

  Input = customer number(customer_id); Contract Type(con_t); Branch number(b_id)

  Read_Set = { Contract.(contract_date+contract_no+outstanding_balance+status); TreatmentSchedule.(check_in_date+start_time+contract_no+ treatment_id+checkin_customer+b_id); ContractTreatment.(used_count+contract_no+treatment_id+ purchased_count+is_exchangable); TelCount.(weather); Product.(pro_num+pro_name+pro_desc+service+unit_cost+itembonus

+salepoint+QtySold+lastyearsale+lastyearprofit+lastyearQtySold
+suggest_sale_price+ price_mode); InvoicePay.(Invoice_num+payment_branch);}
Write_Set = { contract.(outstanding_balance+post_dated_total+sales_id);
customer.(membercard_name+cardexpired_date+accu_base_date
+cust_name+phone+city+province+street+region+postal_code
+total_sales+account_b_id+cardeffective_date+
accu_base_date+thisyear_accu_amount+lastyear_accu_amount);
snControl.(buyref+useref); Treatment.(treatment_sold)}

- The **CheckOut** transaction processes a client's inventory and payment information. The templates for this type of transaction are:
Input = customer number(customer_id); Check in number(checkin_id); Branch number(b_id)
Read_Set = { Product.(Pro_num+Pro_name); TreatmentSchedule.(checkin_id+
contract_no+checkin_date+checkin_treatment+checkin_customer+
start_time+status+tFacility_id+ tFacility_name+customer_id+
treatment_id+b_id+QTY); contract.(contract_no+customer_id+contract_date+
contract_amount+net_sales_amount+outstanding_balance+b_id
+ sales+status+is_member+is_new+is_sponsor+con_t+
post_dated_total+invoice_num+treatment_id+status+purchased_count+
used_count+is_exchangable+unit_price+subtotal+bonus_point+original_price);
customer.(c_id+c_type+c_name+ship_city+ship_prov+ship_postal+out_balance+
last_order+credit_amount+total_sales +total_profits+bill_city+bill_street+bill_prov+

bill_postal+prepay_balance+tax+point+credit_balance+b_id+member_cardname+
thisyear_accu_amount+lastyear_accu_amount);
InvoicePay.(paydate+invoice_num+holder+pay_amount+
payment_discount+approved_num+receipt_id);
Invoice.(invoice_num+sub_total+tax+pay_method+credit_type+c_id+order_num+
total_amount+discount_amount+void_invoice+shipaddress
+outstanding+bonus+profit+taxable_amount+total_size+size_unit);
product.(inventory);}

Write_Set = { contract.(sales_id); contract.(cross_branh_use);
invoice.(last_mod_by+credit_balance); invoicepay.(delflag);
contract.(status+outstanding_balance); inventory.(quan_on_hand+quan_reserve+
ord_point+on_order+ord_date+warehouse_id+suggestorderQty);
product.(prod_name+prod_desc+unit_cost+measure+warranty+
onsales+service+tax+barcode+serial_control
+itembonus+GM_ucost+QtySold+lastyearsale+lastyearprofit+lastyearQtysold
+suggestsaleprice+pricemode+customersalesonly+autoprice);
product_desc.(dim_unit+weight_unit+height+weight+width+depth)}

Chapter 4

Preventive Damage Management through Filtering of Damage Spreading Transactions

4.1 Introduction

As we have mentioned, existing database security technologies have their limitations in handling data corruption caused by damage spreading. When a database is under an attack, rolling back and re-executing the affected transactions are the most used mechanisms during system recovery. This kind of mechanism either stops (or greatly restricts) the database service during repair, which causes unacceptable availability loss or denial-of-service for mission critical applications, or may cause serious damage spreading during on-the-fly recovery where many clean data items are accidentally corrupted by legitimate yet affected new transactions. To resolve this dilemma, we propose to answer the following question: When a database system is identified under an attack by either an intrusion detection system (IDS) or a database administrator (DBA), how can the server prevent spread of damage while continuously providing data services? In Chapter 3, we have proposed a light weighted damage management system TRACE that can solve this problem. From the demonstrated experimental results in Chapter 3.6, we notice that when a database system protected by TRACE is identified under an attack by either an intrusion detection system (IDS) or a database administrator (DBA), the server often

mistakenly quarantines a large number of clean data records to prevent spread of damage. Although the overhead imposed on the database system by TRACE is negligible, TRACE consumes a large amount of CPU-time to do the damage assessment, especially when the system is experiencing a dense data damage attack. Additionally, if the dense damage attack frequently occurs, the database integrated with TRACE will experiences low performance.

In this work, we take the first step to solve this problem and propose a novel proactive damage management approach denoted *database firewalling*. This approach deals with transaction level attacks. After a comprehensive investigation, we find that these data corruption related attacks often leave a fingerprint in the system after attacks are launched, namely *damage spreading pattern*. The idea of this approach is to extract robust damage spreading pattern out of previous attack histories. When specific damage spreading patterns repeatedly appear under the same *type* of attacks, we can extract these patterns. And then, we can re-use these mined patterns to quickly predict the data objects that could have been corrupted during the intrusion detection latency window even before the time-consuming damage assessment procedure starts. Thereafter, we can instantly set up firewalling rules to police data access requests from newly arrived transactions and block only the data objects that match the pre-mined patterns. In this way, spread of damage can be dramatically reduced (if the patterns are good), while the great majority of data objects in the database will be continuously accessible during the entire online recovery process. We present an association rule based mining algorithm to discover the frequent damage spreading pattern. In addition, we design a Bayesian network based algorithm to dynamically estimate the data integrity using the

discovered frequent spreading patterns. With these features, the database firewalling is able to enforce a set of policy of transaction filtering to dynamically filter out the potential *spreading transactions*. Extensive experimental studies have been conducted to evaluate the database firewalling prototype. It is shown by empirical results that this prototype can meet the requirements of continuously providing services in accordance with differential requirements of data quality while the database is under an attack.

4.2 Background and Preliminaries

In this chapter, we first recall the critical security threats that we have addressed in the previous chapter 3, which can cause the data corruption problem, then introduce a set of formal definitions used in this topic.

4.2.1 The Threat Model

In this paper, we deal with data corruption problem in DBMS. Data corruption can be caused by a variety of ways. First, the attacks can be done *through web applications*. Among the $OWASP^2$ top ten most critical web application security vulnerabilities, three out of the top 6 vulnerabilities can directly enable the attacker to launch a malicious transaction, some of which are listed as follows: 1)*Injection Flaws*; 2)*Unvalidated Input*; 3)*Cross Site Scripting (XSS) Flaws*. Second, the attacks can be done *through insider attacks*. TRACE focuses on handling data corruption caused by Injection Flaws and insider attacks through transaction level attack. Transaction level attacks have been well studied in a good number of researches [3, 17, 68].

4.2.2 The System Model

A *database* system is a set of data objects, denoted as DB=$\{o_1, o_2, \ldots, o_n\}$. A transaction T_i is a partial order with ordering relation $<_i$ as defined in [12]. The (usually concurrent) execution of a set of transactions is modeled by a structure called a history. Formally, let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions. A complete history H over T is a partial order with ordering relation $<_H$, where: (1) $H = \cup_{i=1}^{n} T_i$; (2) $<_H \supseteq \cup_{i=1}^{n} <_i$.

Given a set of similar attacks A = $\{A_1$,A_2,...,$A_k\}$ that had happened, the *attack history* for A_i denoted as H_i^A is the transaction sub-history whose write set contains damaged data object sets. An attack history H_i^A, as shown in Table 4.1, starts when A_i happens and ends when A_i is detected and it over T is also a partial order with ordering relation $<_H$. $H_i^A = \{T_i < r_i, w_i >, ..., T_k < r_k, w_k >, ...\}$, where, T_k is affected by T_i directly if $r_k \cap w_i \neq \varnothing$, or T_k is affected by T_i indirectly if $r_k \cap w_j \neq \varnothing$, $i < j < k$, and T_j is affected by T_i directly or indirectly, write set w contains only bad data objects.

TransID	ReadSet	WriteSet	Timestamp	PatientID
0001	<A B D>	<B C>	050617	P_1
0002	<A C>	<D>	050890	P_2
...				
0010	<E>	<A>	061098	P_1
...	...	...	...	...

Table 4.1. An Example of Attack History H_i^A

We propose a fine-grained mechanism to protect data objects stored in a database. Each data object o_i is uniquely represented by a triplet <t.id, c.id, r.id>, where *t.id*,

c.id and *r.id* denote the *table*, *column*, and *row*, respectively. An itemset is defined as a non-empty set of data objects. We denote an *access sequence* $a_s = <s_1 s_2 ... s_m>$ by an ordered list of itemsets $s_m=(o_1 o_2 ... o_n)$, where $s_m \subseteq w_m$ is an itemset updated by a transaction T_m at transaction time t, and o_n is a data object. In Table 4.2, each row represents an access sequence. Although the access of objects largely depends on the goal of an attacker, how the damage spreads is determined by the predefined transaction scripts once the attack is done. Given the attack histories H_i^A, we observe that attack histories of similar attacks have similar damage spreading patterns in terms of accessing similar sets of data objects. Hence, our research problem becomes to discover the damage spreading patterns and use these patterns to estimate the integrity of data objects when a new attack is detected.

4.3 Overview of Approach

Patient	Access Sequence		
PID	Transaction time t		
P_1	<(B C)[a]	(A)	(D E)>
P_2	<(D)	(B)>	
P_3	<(A C)	(B D)	(E)>
P_4	<(A D)>		
P_5	<(B)	(D)	(A)>

[a] Each character denotes a data object represented by a triplet (t.id,r.id,c.id).

Table 4.2. An Example of An Attack History H_i^A Grouped By Patient ID and Sorted By Transaction Time

4.3.1 Rational

In general, IDSs monitor system activities to discover attempts to gain illicit access to systems or corrupt data objects in systems. Existing methodologies of IDS [13, 19, 37, 59, 74] indeed make the system attack-aware but not attack-resistant. Although intrusion detection techniques and tools cannot do the damage tracking/quarantine/cleansing work as the proposed damage mitigation technique will do, intrusion detection is a basic component of any end-to-end intrusion recovery solution.

Achieving accurate detection is usually difficult or expensive. For the best case, if an IDS can instantly response to the malicious intrusion with accurate reporting of data damage, database systems using existing technologies such as proactive containment and recovery may experience great data availability lost because the accurate data damage assessment is expensive to perform. In general, these techniques use a time window based approach to first quarantine the suspicious data objects. During this time window, a substantial innocent data objects are blocked while the accurate damage assessment is proceeding. With the database firewalling technique, after the proactive containment procedure, the database can immediately process the incoming data access request, thus increase the data availability. However, for the average case, the IDS usually has relatively long detection latency. It needs a substantial time to generate the accurate alerts. In this case, general proactive containment plus repair approach will cost great damage spreading. With the database firewalling technique, the database could response to the half-cooked suspicious transactions and performs probability based filtering before the

accurate alert is generate. In the following sections, we will go into the details of the database firewalling approach.

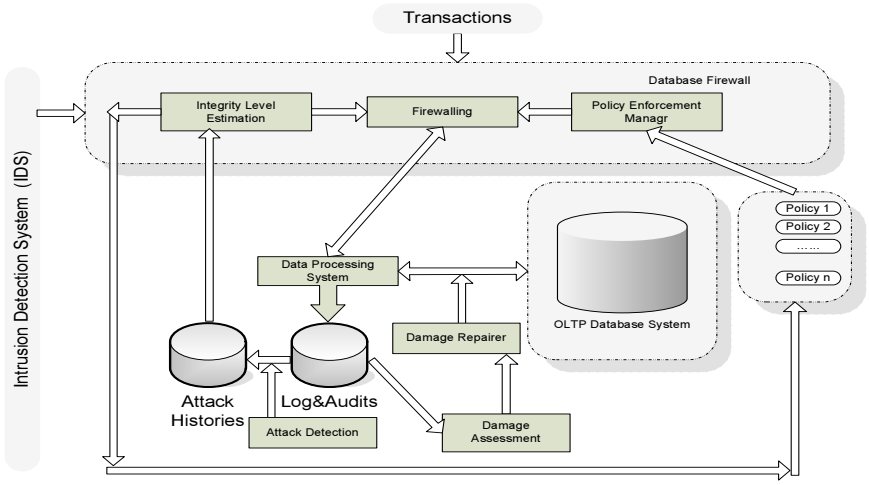


Fig. 4.1. Database Firewall Architecture

4.3.2 Database Firewall Architecture

As shown in Figure 4.1, the database firewall architecture is built upon the top of a traditional "off-the-shelf" DBMS. The database firewall consists of two sets of components: *offline* and *online* as shown in Figure 4.2. Offline components include the pre-built Bayesian Networks (BN), the pattern *Miner*, and the stored attack histories. For offline components, after the database system is recovered from an attack, existing discovered damage spreading patterns are updated by mining the attack histories that are combined the newest attack history with previous attack histories stored in *Attack Histories*. Then, the pre-built BNs will be updated accordingly. Online components

include the transaction parser, the *Filter*, and the integrity *Estimator*. For online components, once a new attack is detected by external component IDS and is mapping to a particular type of attack, the pre-built Bayesian network corresponding to that kind of attack is loaded into the Estimator, and then the prediction of the integrity of data objects is estimated based on the Bayesian network. Note, because IDS is an external component, our approach can work with any existing database IDS techniques. For example, we adopt the following work [13, 59] into our solution.

The detailed mechanism of the database firewall is as follows. When there are no attacks detected, the firewall is transparent to the database system and bypassed. The incoming transactions are processed by the *data processing system* as usual. As soon as an attack is identified by *Attack Detector* and the damaged data by malicious transaction T_i^B is located by *Damage Assessor* using stored transaction log and audit files, the *Integrity Estimator* starts to estimate the set of dirty data objects based on the built Bayesian network. When the *Damage Assessor* locates the damage, *Damage Repairer* repairs the damage data objects reported by *Damage Assessor* and *Integrity Estimator* using some specific *cleaning* transactions. *Policy Enforcement Manager* works as a *proxy* for decision making of data objects access. *Firewalling Manager* functions when *Attack Detector* detects malicious transactions. For example, upon the time when a malicious transaction T_i^B is detected, *Attack Detector* notifies *Firewalling Manager* to set up a new access policy to abort every active transaction and deny new coming ones at this time point and to build up firewalls to prevent damage from spreading. After the firewalls are built up, *Firewalling Manager* triggers *Integrity Estimator* to start estimating the integrity of data objects and consequently forces *Access Policy Manager*

to set up access rulesets to restrict access to the data objects that are confined in firewalls according to a new policy. The workflow of the database firewall is shown in Figure 4.2.

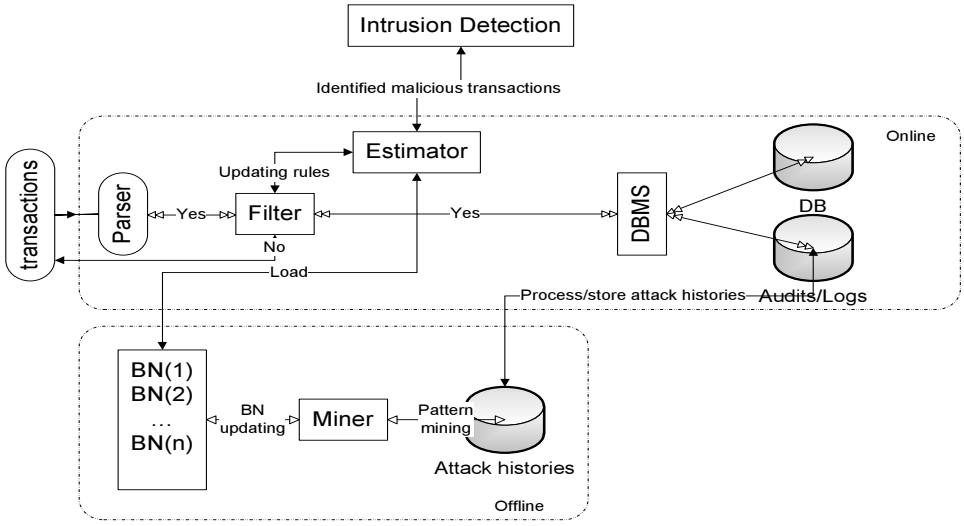


Fig. 4.2. The Database Firewall Workflow

This procedure may cause some availability loss at the stage of loading the Bayesian network and at each step of estimation, while the time consuming part of our db firewall solution, namely the pattern mining and Bayesian network generating part, typically does not cause any availability loss because it is done in offline manner. At each step of integrity estimation, the firewalls update themselves in response to the changes of data integrity level. Accordingly, any new transaction submitted by a user will comply with the newly updated policy. Along with scanning the new attack history, *Integrity Estimator* adaptively predicts the integrity of compromised data until the final solution is reached.

4.4 Mining Damage Spreading Patterns

In this section, we present how the miner shown in Figure 4.2 works.

4.4.1 Two Types of Damage Spreading Patterns

Figure 4.3 shows an example of the *frequent spreading patterns* generated based on a clinic OLTP application after the *SQL Injection Attack*. We observe that there are two types of spreading patterns. We name them *one-hop* spreading, and *multi-hop* spreading, respectively. In most applications, if we cluster the read and update operations on a database, we would typically find that most updates are "bounded" within the scope of a certain database *entity* (e.g., a patient in a health care database). That is, when the value of an attribute of an entity is changed, all the inputs used in the update are typically read from the same entity's attributes. As a result, the set of reads and updates clustered around each entity form a cluster of operations called an *island* in our model. Nevertheless, it should be noticed that due to the various relationships between entities, as well studied in the ER model, cross-island updates are not uncommon, and in many applications cross-island updates might even be dominant. In such an update, one entity's attributes are used in changing the value of an attribute of another entity. As shown in Figure 4.3, damage spreading within each circle (an island) illustrates a one-hop spreading pattern, such as $<(o_1^a, ..., o_x^a)_w>$. Damage spreading crossing islands illustrates a multi-hop spreading pattern, such as island $a \Rightarrow$ island b.

Based on this observation, we discover that the attack histories of similar attacks contain similar damage spreading patterns when any of the following conditions are met.

1) The transaction semantics are not corrupted due to an attack. 2) If the transaction semantics are indeed corrupted due to an attack, the similar attacks corrupt the semantics the same way. It can be seen that once such frequent spreading patterns are identified, pattern-based integrity estimation can be applied and an access policy can be quickly and dynamically enforced to filter out potential damage spreading transactions while the database system continuously provide services in the face of an attack.

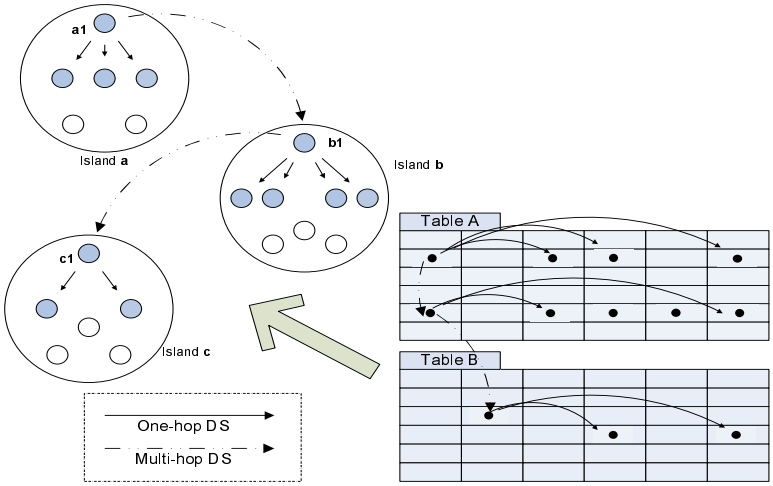


Fig. 4.3. An Example of Two Types of Damage Spreading

4.4.2 Mining The Damage Spreading Patterns

In previous section, we identified two types of *damage spreading patterns* from the attack histories of a database. In this section, we present the approach for discovering the frequent damage spreading patterns.

4.4.2.1 Basic Definitions

In our analysis of the attack histories, each attack history H_i^A is converted to records in the form of $[tA^i, tTime, tItemset_w]$, where $tTime$ denotes the transaction time and tA^i denotes an attribute of an entity. In an attack history, all the data objects updated by transactions associated with the same value of attribute tA^i of an entity can be viewed as a sequence together. For instance, as shown in Table 4.2, in the clinic application, an attack history is converted by tA^i=*patient-id*. To evaluate the importance of a pattern, the support is used as a metric parameter. The definition of support for a pattern may very from one application to another. In this paper, the definitions of support are defined as follows:

DEFINITION 6. ***Support of Damaged Data Object.*** *The* support *of a frequent damaged data object o_x is denoted by the ratio of the number of data object sets having o_x to the total number of data object sets in an attack history.*

DEFINITION 7. ***Support of Damage Spreading Pattern.*** *The* support *of a frequent damage spreading pattern X, including both one-hop and multi-hop, is denoted by the ratio of the number of attack histories containing X to the total number of attack histories.*

In practice, we find that some data objects are damaged in a random manner. Scanning the entire set of attack histories to find their support ratio is time consuming and does not benefit our pattern mining process. Thus, Definition 6 of the minimum support is defined to filter out these uninteresting corrupted data objects.

One-Hop Spreading Pattern	Mapping
<(A)(B)>	1
<(A)(E G)>	2
...	...

Table 4.3. One-Hop Spreading Pattern Mapping

Histories	After Mapping
$H_1^{A'}$	{1} {2} {3}
$H_2^{A'}$	{3} {4} {5}
$H_3^{A'}$	{1} {2} {4} {5}
$H_4^{A'}$	{1} {2} {3} {4}
$H_5^{A'}$	{1} {4} {5}
⋮	⋮

Table 4.4. An Example of Attack Histories $H_i^{A'}$ After Mapping

	Access Sequence (sorted by t)			
Histories	t^1	t^2	$...t^k$	$S^M > 40\%$
H_1^A	<(A)(B)>	...	...	<(A)(B)>
H_2^A	...	<(C D)(A)(E F G)>	...	
H_3^A	<(A H G)>	...	...	
H_4^A	...	<(A)(E G)(B)>	...	<(A)(E G)>
H_5^A	<(B)>	...	...	
⋮	⋮	⋱	⋮	

Table 4.5. An Example of Using Attack Histories to Mine One-Hop Spreading Patterns

4.4.2.2 Finding One-Hop Spreading Patterns

Given a cluster of attack histories H_i^A that are caused by the same (type of) attack, the problem of mining one-hop access patterns is to find within an entity the large access sequences among all sequences that satisfy a user-specified minimum support. A large *access sequence* is denoted as $a_s^l = <s_1, s_2, \ldots, s_i>$, where each itemset s_i satisfies the specified minimum support threshold. The support of itemset s_i is similar to the definition 2 of support and is defined as the fraction of attack histories that contain it. Each such large access sequence a_s^l is denoted as a one-hop *frequent spreading pattern*. To dig out the large access sequences from attack histories, the histories are transformed in the form as shown in Table 4.2 in which data objects are grouped by an attribute tA^i and sorted by transaction time $tTime$. Note, in transformation, we assume that access sequence a_s = <(A A)> = <(A)>, and similarly a_s =<(A)(A B)> = <(A)(B)>, and the large access sequence a_s^l = <(A)(B C)> = <(A B)(C)>. Since we only consider the damaged data objects, in this way, we can reduce the redundancy.

EXAMPLE 3. *Table 4.2 demonstrates an attack history that is grouped by patient-id and sorted by transaction time. Table 4.5 shows the attack histories expressed as a set of access sequences. Consider the example of histories shown in Table 4.5, in accordance with the Definition 2, with the minimum support set to 40% (i.e., two histories of total five histories.), two access sequences are found: <(A)(B)>, <(A)(E G)>. A real damage spreading pattern from clinic application is shown as follows: <(Treat_Status, Used_Count) (Cust_Name, Year_Accu, Accu_Base) (Point.x) (Total_Sales, Quan_OnHand)*
>

The data objects (A) and (B) in *damage spreading pattern* <(A)(B)> are accessed both in histories 1 and 4, and the data objects (A) and (E G) in *damage spreading pattern* <(A)(E G)> are supported by histories 2 and 4. In history 2, itemset (C D) will be first filtered out due to support definition 1. In history 4, since we try to dig out access sequences that are not necessarily contiguous, although another access sequence is in between items (A) and (B), <(A)(B)> still counts as a large access sequence. Obviously, access sequence <(A H G)> is not a large access sequence because only history H_3 supports it. Although access sequences <(A)>, <(B)>, <(A)(E)> and <(A)(G)> have satisfied the minimum support, they are not counted as a pattern because they are not the large access sequences.

4.4.2.3 Finding Multi-Hop Spreading Patterns

In this section, we will present the idea of identifying multi-hop spreading patterns. Without loss of any generality, we assume that the one-hop spreading patterns are mapped to a set of contiguous integers as shown in Table 4.3. Table 4.4 illustrates an example of attack histories after mapping. $H_i^{A'}$ denotes an attack history i after mapping and is sorted by the transaction time of the first data object set s_m of each large access sequence (one-hop) a_s^l. By doing this, the detailed information of one-hop spreading patterns is hidden and the focus is moved to finding out how the damage migrates among the islands.

EXAMPLE 4. *Consider the attack histories shown in Table 4.4. The large one-hop spreading patterns found have been replaced by the set of integers. The minimum support is specified to be 40%. Applying the same steps used in mining one-hop patterns, the*

large multi-hop spreading patterns would be following: $< \{1\}\{2\}\{3\} >$, $< \{1\}\{2\}\{4\} >$, $< \{1\}\{4\}\{5\} >$ *and* $< \{3\}\{4\} >$.

To find the one-hop and multi-hop spreading patterns, the Apriori algorithm [**?**] is adopted. Given the minimum support, the algorithm of mining the one-hop is shown as below, where $(a_s.o_i)^s$ is the support ratio of a data object o_i in the access sequence a_s against the minimum support (definition 1), and $(a_s)^s$ is the support ratio of an access sequence against the minimum support (definition 7). The set of candidate patterns is denoted as C, and I_k denotes the final list of large access sequences (one-hop spreading pattern). Similarly, the multi-hop can be mined by repeating steps 9 through 16 using the after-mapping histories.

4.5 Integrity Level Estimator

In previous section, we show how to mine frequent damage spreading patterns. In this section, we propose a novel database firewall approach to use these patterns to "throttle" damage spreading or propagation (during on-the-fly attack recovery).

Old methods (such as finding damaged data objects using dependency relation [2]) can achieve high accuracy of damaged data objects. However, these methods consume too much time and do not prevent the potential spreading transactions from accessing damaged data objects. Thus, it downgrades the availability of data objects and the system integrity and would further exacerbate the situation of denial of service. In this section, we propose an approach using probabilistically reasoning based on Bayesian networks [58].

Algorithm 6: Mining Frequent Damage Spreading Patterns

Input: Histories H_i^A
Result: Frequent one-hop damage spreading patterns
/* * */
[f]for each attack history, grouped by tA^i and sorted by time t
begin
  forall $a_s \in H_i^A$ **do**
    if $(a_s.o_i)^s < min\_sup^1$ **then**
      remove o_i from a_s, H_i^A;
    else if $(a_s)^s \geq min\_sup^2$ **then**
      $a_s \rightarrow I_1$;
    else
      remove a_s from $H_i^{'}$;
  for $k \rightarrow 2$ **to** $I_{k-1} \neq \varnothing$ **do**
    C $= I_{k-1} \bowtie I_{k-1}$;
    foreach $a_s \in C$ **do**
      if $(a_s)^s < min\_sup^2$ **then**
        remove a_s from C;
      else
        do nothing;
    $I_k = C$;
end

4.5.1 Bayesian Network Based Analysis on Data Integrity

In many cases, it is not possible to give out precise data integrity information (either good (clean) or bad (dirty)) in response to customer's queries without any delay after a database system is detected to be under an attack. Among various techniques used for analyzing the integrity level of data objects, the Bayesian network-based predictive reasoning is adopted in our database firewall framework. Through the analysis of the mined spreading patterns, we probabilistically determine the integrity of data objects from previous experience on the fly of an attack.

Conditional probabilities are the key for reasoning the possible data integrity. For example, if we know there is a $P(A|B) = 90\%$ about two data objects A and B, it is equivalent to the logic $B \Longrightarrow A$, which means if the data object B is damaged, the data object A is also damaged with 90% certainty, where, the integrity level of a data object is between[0,1]. Note, in our study, we denote the integrity of the good data object as 0 and the integrity of the bad data object as 1. However, often times we can not get the inference above directly. We choose Bayes's Rule $P(A|B) = \frac{P(A)P(B|A)}{P(B)}$ as our basis for probabilistic reasoning. A Bayesian network is a space-efficient data structure that encodes all of the information in the full joint probability distribution for the set of random variables defining a domain, where we denote each one-hop pattern found in the pattern-mining phase a variable within the domain. Consider a task of specifying an arbitrary joint distribution, $P(x_1, ..., x_n)$, for n random variables. To store $P(x_1, ..., x_n)$ explicitly would need a table with 2^n entries, which tremendously consumes a large number of resources. However, by adjusting the minimum support at the stage of

mining frequent damage spreading patterns, we can filter out uninteresting patterns and thus reduce the space of the domain. Each variable remaining depends on merely a small subset of other variables. Thus, economy of required storage space for joint probabilities can be achieved.

To build the Bayesian network, directed acyclic graphs (DAGs) have been used by researchers to facilitate the decomposition of a large distribution function into several small subsets and to represent causal or temporal relationships. Suppose there is a distribution P defined on n random variables. By using the chain rule of probability, we can have the following equation:

$$P(x_1, ..., x_n) = \prod_j P(x_j | x_1, ..., x_{j-1}) \tag{4.1}$$

$$P(x_1, ..., x_n) = P(x_j | parents(j)) \tag{4.2}$$

$$P(x_1, ..., x_n) = \prod_i P(x_i | parents(i)) \tag{4.3}$$

Since each variable is conditionally dependent of only some of its predecessors, the equation above can be rewritten to equation (2), where x_j is the variable that is only dependent on a select group of predecessors named $parents(j)$. In general, we can construct the Bayesian network as a carrier of conditional independence relationships along the order of construction as equation (3).

4.5.2 Integrity Estimation Using Bayesian Network

A Bayesian network consists of two components. The first component is a directed acyclic graph G whose node corresponds to a one-hop *damage spreading pattern* discovered at the pattern mining phase. The second component is an edge in the graph that denotes a direct dependence of the one-hop *damage spreading pattern* on its $parents(i)$. The DAG encodes a set of conditional independence assumptions that each node is conditionally independent of its predecessors given its parents.

EXAMPLE 5. *Consider the example shown in Table 4.4 in section 4.4.2.3, with the minimum support specified to be* 40%*, the large multi-hop damage spreading sequences are:* $<\{1\}\{2\}\{3\}>$, $<\{1\}\{2\}\{4\}>$, $<\{1\}\{4\}\{5\}>$ *and* $<\{3\}\{4\}>$. *To construct the DAG from these multi-hop patterns, we determine the select group of predecessors of each node if they satisfy the following condition: there is a rule as* $\{parents_i\} \Rightarrow \{\text{node}\}$ *and the confidence of the rule is larger than or equal to a user specified parameter c, then an arc between* $parents_i$ *and node is drawn accordingly. We identify the following rules from above patterns:* $\{1\} \Rightarrow \{2\}(conf = 75\%), \{1\} \Rightarrow \{3\}(conf = 50\%), \{1\} \Rightarrow \{4\}(conf = 75\%), \{1\} \Rightarrow \{5\}(conf = 50\%), \{2\} \Rightarrow \{3\}(conf = 66\%), \{2\} \Rightarrow \{4\}(conf = 66\%), \{3\} \Rightarrow \{4\}(conf = 66\%), \{4\} \Rightarrow \{5\}(conf = 75\%)$.

The rules with confidence less than c=50% are discarded from all rules generated. Figure 4.4 shows an example Bayesian network constructed from above rules. In addition, the conditional probability tables computed based on previous attack histories is drawn as well in Figure 4.4.

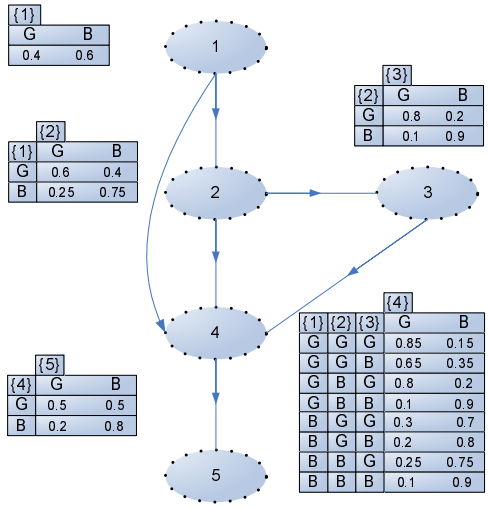


Fig. 4.4. An Example of Bayesian Network with the Damage Spreading Probability Table

Bayesian network is adopted in our framework to predict the integrity of data objects when some data object has been observed as damaged. The idea of is, given evidence about the state G or B of a one-hop pattern, where G and B denote *Good*, and *Bad* for short, the state of data objects in other one-hop pattern can be inferred by equation 4.4:

$$P(X) = \sum_{Y} P(X, Y) \tag{4.4}$$

where Y stands for all unknown variables excluding variable X, and $P(X, Y)$ may be expanded using the product rule shown as the equation (4.3). Because Bayesian network defines a joint probability it is clear that $P(X, Y)$ can be computed from the DAG. For example, to find the probability that the 5 is damaged given that 1 is damaged, that is $P(\{5\} = B|\{1\} = B)$, it is necessary to marginalize over the unknown parameters. This amounts to summing the probabilities of all routes through the graph.

EXAMPLE 6. *Given the Bayesian network as shown in Figure 4.4, if we observe at time t that the data object* $o_i = (A) \in \{1\}$ *is damaged when scanning the new attack history* h_{new}*, we can compute the damage probability of the data object sets belonging to pattern* $\{5\}$ *by equation 4.4 as follows:*

$$\begin{aligned} P(5 = B|1 = B) &= \sum_{x,y,z} \{P(5 = B|4 = x) \\ &\times P(2 = y|1 = B) \\ &\times P(4 = x|1 = B, 2 = y, 3 = z) \\ &\times P(3 = z|2 = y)\} \\ &= 0.465 \end{aligned}$$

Suppose that data objects (C), (D) and (E) are in $\{5\}$, and <(A)(B)> $\Rightarrow$ <(C)(D)(E)> with probability 0.465. Then we denote the integrity level of data objects (C), (D) and (E) as 0.465, which means that there is a 0.465 certainty that these data objects are damaged. If along the scanning process of the new attack history, new evidences of damaged data are found, the integrity level is adjusted accordingly. Meanwhile, filtering rules are enforced and certain clean transactions are triggered to repair the dirty data according to the estimated integrity of the data objects. Finally, the integrity of data objects contained in Y as in algorithm 7 is known, and the corresponding Integrity Filtering List $\hat{I}_i$ (see Definition 3 in section 4.5.4) is updated for generating new filtering rules. This algorithm is incremental. As new evidence obtained, some interim results

from previous calculations can be reused to estimate the integrity of data objects instead of re-computing.

4.5.3 Algorithm of Integrity Estimation

In this section, we give the algorithm of estimating the data integrity in algorithm 2. The algorithm takes three arguments: G, X and Y, where G is the DAG, X is the evidence that have been observed, and Y is the data objects to be estimated. c_{node} denotes the children node of the DAG. Line 3 will check if current children node has its parents. If it has parents, $\sum_v p(c_{node} = v|pa_1 = v, pa_2 = v, \ldots)$ is calculated, where v has two possible values G and B. After all the routes from X to Y are computed, the probability Pr(Y|X=x) is computed by multiplying all interim results together. Finally, the integrity of data objects contained in Y is known, and the corresponding Integrity Filtering List $\hat{I}_i$ (see Definition 3 in section 4.5.4) is updated for generating new filtering rules. This algorithm is incremental. As new evidence obtained, some interim results from previous calculations can be reused to estimate the integrity of data objects instead of re-computing.

4.5.4 Updating the Filtering Rules On-The-Fly

A specific and strongly worded security policy is vital to the pursuit of internal data integrity. This policy should govern everything from acceptance of accessing data objects to response scenarios in the event that a security incident occurs, such as policy updating upon new attacking. Ideally, a database firewall policy dictates how transaction traffic is handled and how filtering ruleset is managed and updated. To limit the potential

Algorithm 7: Integrity Estimation

Input: G, X, Y
Result: Integrity Level List $\hat{I}_i$
/* * */
[f]X → evidence; Y → to be estimated
begin
  c_{node} = Y;
  while $c_{node}.pa \neq \varnothing$ **do**
    p[i] = $\sum_v p(c_{node} = v | pa_1 = v, pa_2 = v, \ldots)$;
    /* * */
    [f]v∈G,B
    $c_{node} = c_{node}.pa$;
    i++;
  Pr(Y|X=x) = $\prod_i p[i]$;
  $\hat{I}_i = \forall\ o_i \in Y \leftarrow Pr$;
end

damage spreading, firewall policy needs to create a ruleset to restrict the entrance of transactions that could compromise other data objects, but let other transactions enter to achieve maximum throughput.

Definition 8. **Integrity Filtering List.** *Integrity filtering list* $\hat{I}$= $\{(o_1, o_x, \ldots, o_y)^{i_1}, (o_2, o_z, \ldots, o_v)^{i_2}, \ldots\}$, *where i is an integrity level. Objects* o_x *with the same integrity level i are grouped together.*

For example, suppose a transaction $T_1(t) = r_1[o_x] r_1[o_y] w_1[o_y]$ needs to enter the database at time t. We know that the integrity of data object o_x has been estimated and considered as corrupted at this moment, our policy checker will screen and be aware if the request can be granted using the rulesets defined as follows:

Rule 1. ∀ *transaction* T, *if* ∃ *data object* $o_x \in R_T$, *and* $R_T \bigcap \hat{I} \neq \varnothing$, *and if* $W_T \neq \varnothing$, *DENY;*

RULE 2. $\forall$ *transaction* T, *if* $\exists$ *data object* $o_x \in R_T$, *and* $R_T \bigcap \hat{I} \neq \varnothing$, *and if* $W_T = \varnothing$, *and if* $i(o_x) <$ Q *then DENY, otherwise GRANT;*

RULE 3. $\forall$ *transaction* T, *if* $\exists$ *data object* $o_x \in R_T$, *and* $R_T \bigcap \hat{I} \neq \varnothing$, *and if* $W_T \neq \varnothing$, *and if* $i(o_x) <$ Q *then DENY, otherwise GRANT;*

RULE 4. $\forall$ *transaction* T, *if* $\nexists$ *data object* $o_x \in R_T$, *and* $R_T \bigcap \hat{I} = \varnothing$, *GRANT;*

Where, $i(o_x)$ is the integrity level of the object o_x and Q is Quality of Information Assurance associated with applications. R_T and W_T are the readset, writeset of the transaction T, respectively. For example, give $Q = 0.5$ of an application, it means if the integrity of data objects is 0.5 or higher, it is acceptable to the application. Note, what we have presented here is a sample ruleset. We should be aware that firewall rulesets tend to become more complicated with age.

4.6 Experimental Results

In this section, we present the experimental results. To assess the performance of our database firewalling approach, we conduct two empirical studies based on both synthetic and real datasets. The simulation model of our experimental studies is described in section 4.6.1.

Our current firewalling prototype has limitations. First, the integrity estimation mechanism is based on a mathematic model, the Bayesian Network, which is built on the mined damage spreading patterns to infer the goodness of data objects. Thus, the current approach will fail when the applications running in the databases do not contain apparently strong data dependency that can result in damage spreading patterns when

attacks happen. Second, the Bayesian Network providing a probability to the data objects could risk the database system in damage leakage when the ongoing attack is not exact the same as the previous attack. We demonstrate from the experimental results when the condition that the applications contain damage spreading patterns is satisfied, our approach performs very well in terms of increasing the data availability and transaction throughput.

4.6.1 Experiment Settings

In our experiments, we use two synthetic history sets generated based on the modified TPC-C benchmark [25]. TPC-C benchmark simulates activities of a warehouse supplier. The supplier operates a number of warehouse (W) each of which has its own stock. A warehouse is comprised of a number of districts, each of which has a number of clients. The TPC-C benchmark also describes the set of transactions that are issued during benchmarking runs: order placement, payment, order delivery, order status inquiry, and stock level inquiry. Orders can only be placed by clients already in the database. We populate the TPC-C application with the parameters shown in Table 4.6. We take the parameters shown in Table 4.7 to populate the synthetic attack histories. The specified settings for each parameters used to generate the history sets are included in Table 4.8.

In addition, a real dataset of transaction histories from a clinic OLTP application is employed to verify the mining algorithm. The dataset (400MB) contains 119 tables with 1142 attributes belonging to 9 different applications. For more detailed description of the dataset we refer the reader to [78]. Moreover, to verify the database firewall prototype, we re-construct the clinic application based on the application and query

templates and generate real_syn attack histories. We also assume that there is only one attack at a time. Additionally, similar attack is practically defined in our experiments as the attacks with the same transaction type, such as *Order, Payment* transaction type in the TPC-C benchmark.

Parameters	Values
Number of warehouses	15
Districts per warehouse	30
Clients per district	5000
Items per warehouse	100000
Orders per district	5000

Table 4.6. TPC-C Parameters and Their Values

Parameters	Notes
N	Number of data objects
T	Average numbers of data objects per transaction
C(P)	Number of customers(patients)
D	Number of transactions
H	Number of attack histories

Table 4.7. Parameters of the synthetic datasets

4.6.2 Mining Algorithms Testing

To verify the mining algorithms, we conducted experiments to measure the quality of mined one-hop frequent *damage spreading patterns* using two metrics, recall and precision. They are defined as follows. Given a set of true *damage spreading patterns A*

Name	N	T	C	D	H
Syn_1	1000	3	500	100,000	10
Syn_2	1000	5	1000	100,000	20
$Real$	1000	8	1000	100,000	10
$Real\_Syn$	1000	8	1000	100,000	20

Table 4.8. Parameter values of the datasets

and a set of obtained frequent *damage spreading patterns* B, the recall is $\frac{|A \cap B|}{|A|}$ and the precision is $\frac{|A \cap B|}{|B|}$.

Data Type	A	B	Recall	Precision	s (%)
Syn	300	279	0.93	1	15
Syn	600	587	0.98	1	10
$Real$	200	190	0.95	1	10
$Real_Syn$	300	291	0.97	1	10

Table 4.9. Measurement of the Mining Algorithm of One-hop Patterns

In Table 4.9, the first column shows we use two different kinds of dataset, one from the synthetic application and one from the real application. The last column is the minimum support s to mine the patterns. It can be seen that when the minimum support is set to 15%, using the synthetic dataset, 0.93 of recall rate can be achieved. When we lower s to 10%, higher recall rate (98%) can be obtained. Although the high recall rate is achieved, the false-negative patterns have the potential to cause the damage leakage. However, after further investigation of the missed patterns, we find that the chance these missed patterns cause damage leakage is small. First, although these missed patterns are true damage spreading patterns, they do not often occur in the damage histories and then are screened out because of the minimum support. The reason

they do not frequently involve in damage spreading is that transaction scripts always have conditional statements, such as *if-else* statement. Only when certain condition is satisfied, the branch will be gone through, and then the patterns occur. Second, these missed patterns are often on the downstream of a multi-hop spreading pattern (similar to the leaf node of a tree). Thus, even if they are damaged and not included in final answer, they do not cause damage leakage. The experiments to measure the multi-hop patterns achieve similar results that are not presented due to limited space. As a conclusion, we believe that the mining algorithm is satisfactory. However, since we deploy a relative simple SQL-injection attack, pattern mining algorithm achieve high recall rate. How the algorithm performs when the attack is sophisticated needs a further investigation. Furthermore, the algorithm is designed particularly for the TPC-C and clinic applications. A generic condition under which the mining algorithm can be useful remains unknown.

4.6.3 System Integrity Analysis

Estimated system integrity SI_e is defined as the ratio of the summation of the damage probabilities of the data objects n^d to the number of total data objects N in the database: $SI_e = 1 - \frac{\sum_{i=1}^{n} P}{N}$. For example, if the database contains $N = 100$ in which total $n^d = 20$ are bad data, the real system integrity should be $SI_r = 1 - \frac{\sum_{i=1}^{20} *1}{100}$=80% when the system is detected to be under an attack. To verify the estimation accuracy of our approach, we define the accuracy as the ratio of the number of data objects that are estimated (and are tested as they are later) to the number of damaged data objects in the new attack history.

Real System Integrity SI_r	Time (s)		%
	BN	DR	Accuracy
80%	50	260	94%
70%	70	270	92%
65%	90	300	87%

Table 4.10. Speed of System Integrity Estimation

Figure 4.5 shows the process of discovering system integrity. Our probability based approach is compared with the algorithm adopted from [2] that use the dependency relation of transactions. Along with the process of scanning the new attack history, at each time t_i a data object o_i or a set of data objects $(o_i, .., o_j)$ are found to be bad, the dependency relation based approach will only know what it has learned up to time t_i. However, with the Bayesian network constructed from previous attack histories, we are able to compute the damage probability of all data objects previously occurred in attack histories. Therefore, our approach should be faster than the dependency based approach in finding the system integrity. Results in Table 4.10 demonstrate that our Bayesian network based approach is much faster than the dependency relation based approach in terms of assessing the system integrity. We can also observe that the estimation accuracy of our approach is relative high when the new attack is similar to previous attacks. When the recall of the pattern mining is high, the estimation accuracy is also high because more knowledge of patterns can be used. However, when a different attack is launched, the accuracy will be low because the previous knowledge is no longer useful to estimate the integrity level of damaged objects. More efforts are needed to study this issue.

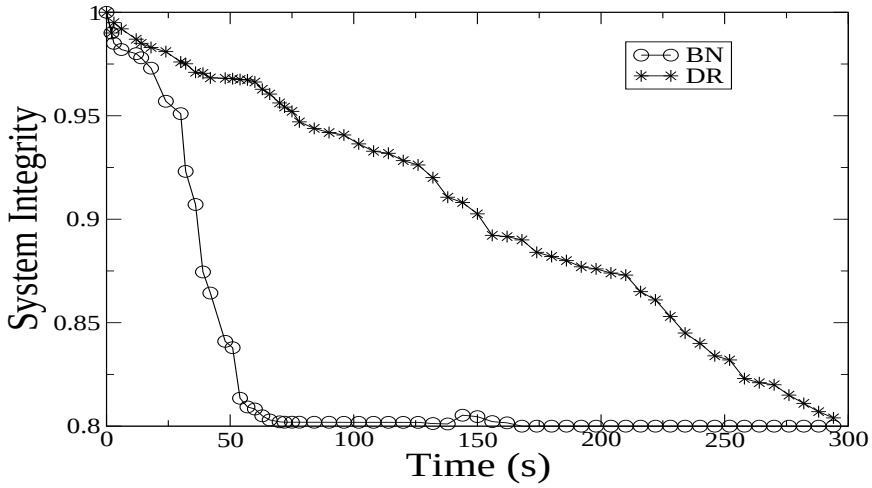
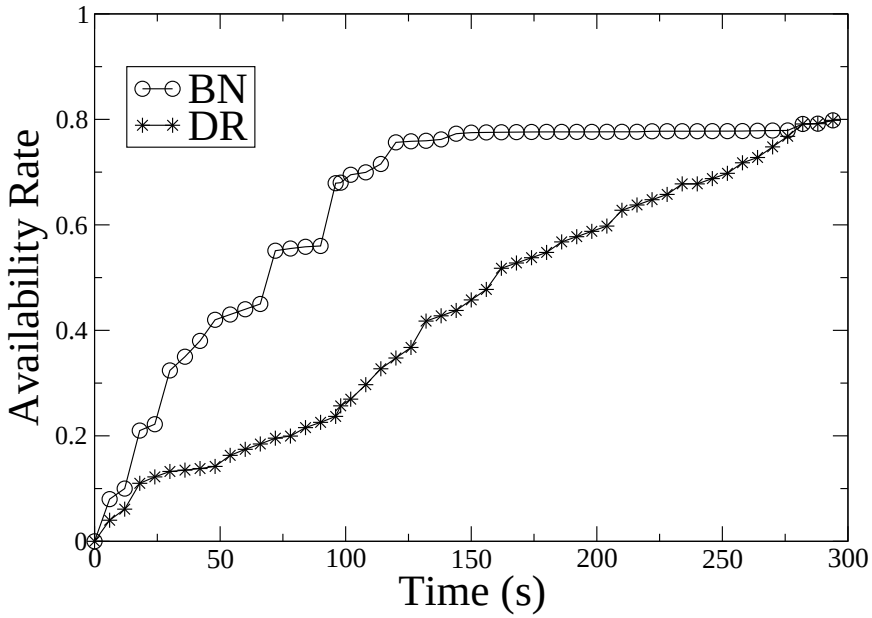


Fig. 4.5. System Integrity Estimation Process. Real System Integrity SI_r=0.8

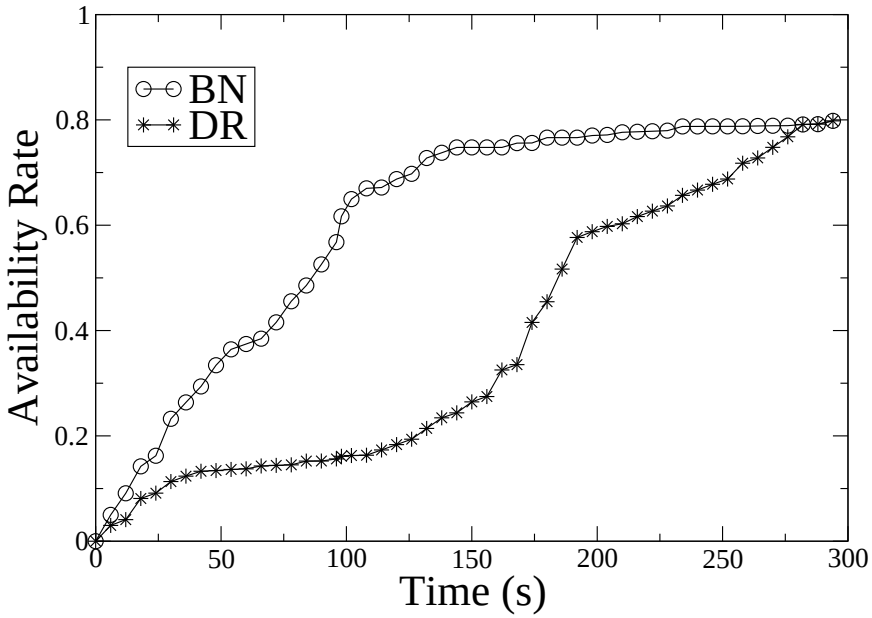
4.6.4 System Availability Analysis

System availability is an important metric to evaluate the performance of the database firewall framework. For conventional methods, e.g. rolling back, even if there are only a few damaged data objects, since the system is not accessible while re-executing, the system availability is zero. It is crucial to the database firewall to reduce this time period and maintain the system availability on a certain level. In our framework, we define two kinds of availabilities. The maximum system availability is defined as the rate of the number of good data objects to the number of the total data objects in the database $A_{max} = \frac{n^g}{N}$. The real-time system availability is defined as the rate of the number of good data objects discovered at time t to the number of the total data objects in the database $A_r = \frac{n^t}{N}$.

Figure 4.6.4 illustrates the results of the system availability with respect to time from both the synthetic and the real histories. The repairing procedure is not included



(a) Synthetic Dataset



(b) Real_Syn Dataset

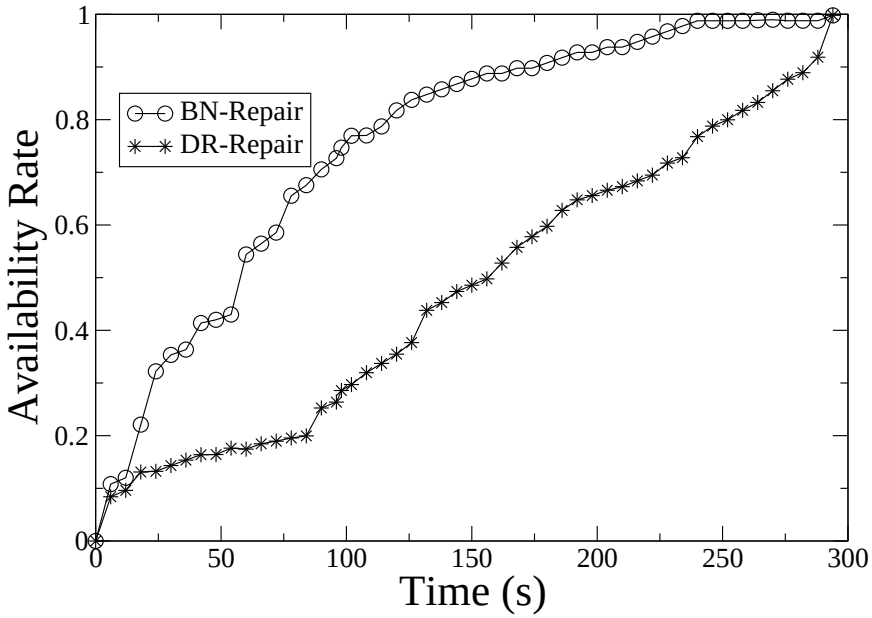
Fig. 4.6. System Availability without Repairing Procedure

in this testing. The approach based on the dependency relation [2] (denoted as DR, Bayesian network denoted as BN) is also implemented to compare with our approach. Let ε be a user specified parameter. If the estimated integrity of data objects is equal to ε or higher, the repairing procedure is fired up instead of waiting until it is 100 percent sure that the data objects are damaged. Let Q be the differential quality of information assurance required by different applications. It may vary from an application to another. If the estimated integrity of data objects is equal to Q or lower, these data objects are available to be accessed. Given $A_{max} = 0.8$, $Q = 0.15$ and $\varepsilon = 0.6$, it can be seen in Figure 4.6.4 that the database firewall approach can maintain a relative higher system availability A_r and also reach the maximum system availability $A_{max} = 0.8$ faster than the dependency relation based approach. Figure 4.7 demonstrates the results of the system availability when the repairing procedure [2] is introduced. Given $Q = 0.15$ and $\varepsilon = 0.6$, it can be seen that the maximum system availability $A_{max} = 1$ can be reach earlier than the dependency relation based approach. Note, we assume the repairing rate is constant.

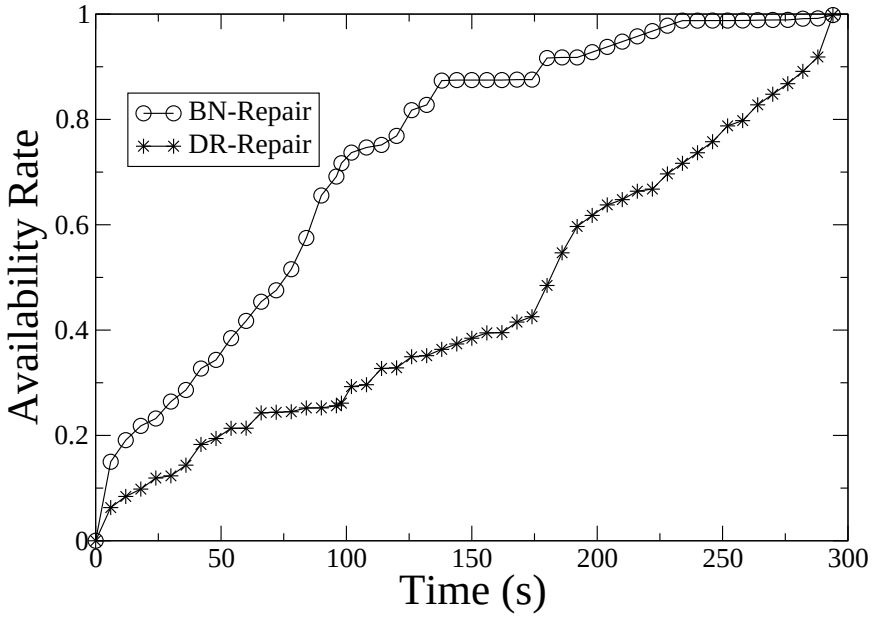
4.6.5 System Throughput Analysis

System throughput in terms of continuously providing data access while filtering out potential damage spreading transactions when the data system is under an attack is an important metric to measure the performance of the database firewalling framework.

In practice, the system throughput is defined as a ratio of the number of served transactions to the number of transactions requiring service: $Tr = \frac{n^s}{N}$. We assume that

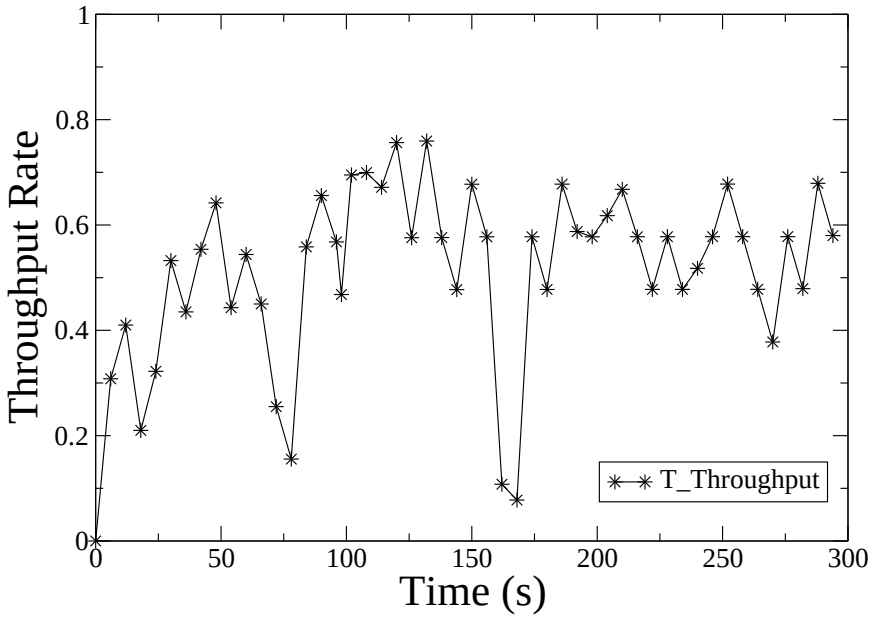


(a) Synthetic Dataset

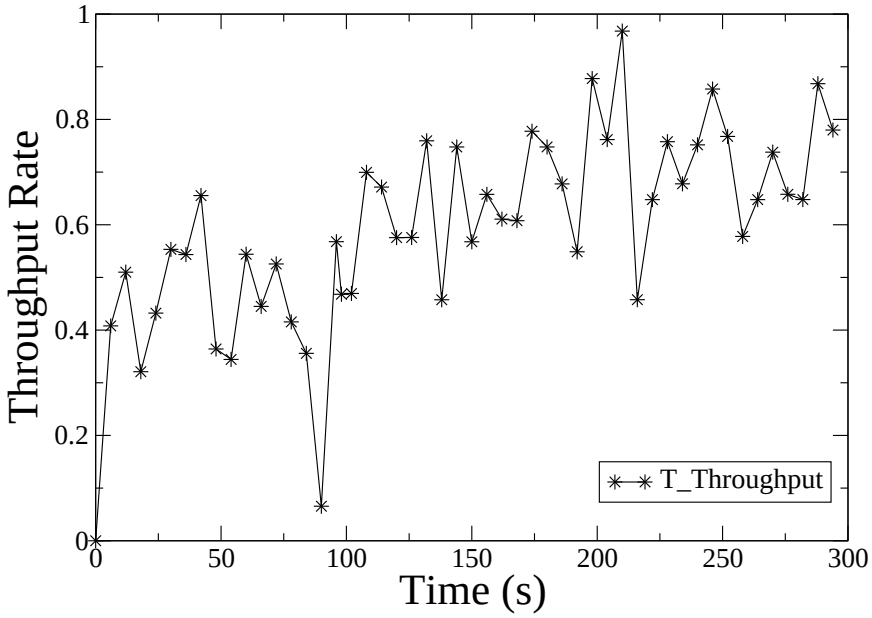


(b) Real_Syn Dataset

Fig. 4.7. System Availability with Repairing Procedure



(a) Synthetic Dataset without Repairing Procedure



(b) Real_Syn Dataset with Repairing Procedure

Fig. 4.8. System Throughput

the arrival rate of the transactions is constant. Figure 4.8 demonstrates the performance of the system throughput from both the synthetic and the real dataset without taking into consideration the damage leakage. To filter out potential damage spreading transactions, we assume the transaction dependence is tight. For instance, if transaction T_2 does not access compromised data but rely on the result of transaction T_1, transaction T_2 will still be discarded if transaction T_1 is filtered out due to accessing compromised data. This assume is reasonable because the result directly from transaction T_1 is dirty.

Given $A_{max} = 0.8, Q = 0.15$ and $\varepsilon = 0.6$,it can be seen that, in Figure 4.8(a), the system throughput reaches nearly zero at time $t = 75s$. The reason is that the data objects required by the transactions are all contained, and filtering policy does not allow those data objects being accessed by the transactions. In addition, in both Figure 4.8(a) and (b), although the system throughput vibrates up and down along the time axis, the average ratio of the system throughput is above 50%. Moreover, if the repairing process is introduced in Figure 4.8(b), the system throughput can be even higher.

Chapter 5

Conclusion

5.1 Summary

In my PhD study, I mainly focus on investigating critical security topics in database damage management. As we have seen, damage management is a very important problem faced today by many mission/life/business-critical applications and information systems that must manage risk, business continuity, and assurance in the presence of severe cyber attacks. Damage management is a broad research topic. In this study, I investigate the critical damage management techniques in database security research from the following dimensions: *damage spreading*, *damage quarantine and recovery*, and *correctness and quality of the damage management*.

Correct and high-quality damage quarantine and recovery mechanisms are highly desired in mission/life/business-critical applications because of the following main reasons: 1) experience with data-intensive applications, such as credit card billing, online banking, inventory tracking, and online stock trading, has shown that a variety of attacks successfully fool traditional database protection mechanisms. Because of data sharing and inter-operability between business process and applications, cyber attacks could even enhance their damage because of catastrophic cascading effects. 2) Due to the fundamental differences between failure recovery and attack recovery, the damage

management problem cannot be efficiently solved by failure recovery technologies which are very mature in handling random failures.

I design and implement TRACE as a zero-system-down-time database damage tracking, quarantine, and recovery solution with negligible run time overhead. The service outage is minimized by (a) cleaning up the compromised data on-the-fly, (b) using multiple versions to avoid blocking read-only transactions, and (c) doing damage assessment and damage cleansing concurrently to minimize delay time for read-write transactions. Moreover, TRACE uses a novel marking scheme to track causality without the need to log read operations. In this way, TRACE has near-zero run-time overhead. We build TRACE prototype into the DBMS kernel of PostgreSQL, which is currently one of the most advanced open-source DBMSs with transaction support, not layered on top. In summary, TRACE is the *first* integrated database tracking, quarantine, and recovery solution that can satisfy all the four highly desired requirements shown in Table 1. In particular, TRACE is the first solution that can simultaneously satisfy requirements R1 and R2. In addition, TRACE is the first solution that satisfied R4.

I also investigate the damage spreading problem. Typically, once a DBMS is attacked, the damage (data corruption) done by these malicious transactions has severe impact on the DBMS because not only are 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. When a database system is attacked, the immediate negative effects (basically, the data damage) caused particularly by this attack are usually relatively limited. However, the negative effect can enlarge its impact largely because the data damage can be spread

to other innocent data stored in the database system due to the nature of transactional read/write dependency of the database systems. The database system armed with existing security technologies cannot continue providing satisfactory services because the integrity of some data objects is compromised. Simply identifying and repairing these compromised data objects by operating undo and redo transactions still cannot ensure the database integrity due to the damage spreading.

In this damage spreading study, I answer the following question: when a database system is identified under an attack by either an intrusion detection system (IDS) or a database administrator (DBA), how can the server prevent spread of damage while continuously providing data services? I take the first step to solve this problem and invent a novel proactive damage management approach denoted *database firewalling*. This approach deals with transaction level attacks. Through a comprehensive investigation, I find that these data corruption related attacks often leave a fingerprint in the system after the attacks are executed, namely *damage spreading pattern*. The idea of this approach is to extract robust damage spreading pattern out of previous attack histories. When specific damage spreading patterns repeatedly appear under the same *type* of attacks, I can extract these patterns using newly designed data mining algorithms. Then, I can re-use these mined patterns to quickly predict the data objects that could have been corrupted during the intrusion detection latency window even before the time-consuming damage assessment procedure starts. Thereafter, I can instantly set up firewalling rules to police data access requests from newly arrived transactions and block only the data

objects that match the pre-mined patterns. In this way, spread of damage can be dramatically reduced (if the patterns are good), while the great majority of data objects in the database will be continuously accessible during the entire online recovery process.

5.2 Future Work

5.2.1 Research Plan

Today's software and hardware suffer from vulnerabilities that provide attackers with unauthorized access to computer systems. Attackers can leverage all possible system flaws to gain access locally or remotely. By gaining the unauthorized access to critical systems, the attackers can arbitrarily alter or damage a system. In this study, we have investigated some critical security issues in database damage management. By solving these problems, we anticipate to obtain good solutions in terms of performance and security. This dissertation has showed that our data management system in DBMS provides practical and effective attack tolerance for Database programs.

Although the pitfalls of insecure programs have been apparent for years, vulnerabilities continue to pervade todays computer systems. The continued use of attack intolerant systems and the persistence of broken programming models used to create new software unfortunately guarantees that attackers will continue to successfully take control of the critical computer systems. Many opportunities for continued research in data damage management system remain.

In this dissertation, I have published five papers and completed several others for consideration of publication. The proposed TRACE technique and the preliminary

results in Chapter 3 have been accepted to ESORICS Conference 2008 [10]. The fine-grained TRACE system idea has been accepted in HASE 2008 [7]. Its extended version is now accepted in EDBT 2009 [8], and a completed version will also be submitted for journal publication. The proposed Database Firewalling technique and the preliminary results in Chapter 4 have been published in Annual IFIP WG 11.3 Working Conference DBSec [9], and ASCAC Conference [6], respectively. Its extended version is now under review for Elsevier Computer Standards & Interfaces Journal.

I will continue to work on the dissertation to obtain better experimental results. In the following, I will list two tasks that could be addressed to further improve current damage management solutions, and extend beyond the database damage management.

5.2.1.1 Task 1. Non-Blocking Repair Using a Shadow Database.

As we have explained in Chapter 3, TRACE uses a novel technique to do causality logging, damage assessment, damage quarantine and recovery. Like all existing transaction-level recovery schemes, the TRACE recovery scheme still belongs to the paradigm of "clean-then-reexecute", which is a two-phase procedure. In phase one, a set of cleansing transactions are executed to revoke the effects of malicious and affected transactions. In phase two, the set of affected transactions are reexecuted. Existing schemes of how affected transactions are executed, in general, belongs to the category of "lock-competing reexecution". Hence, being-reexecuted transactions may often compete with new transactions on accessing data records. In this task, we propose to shift the paradigm from "lock-competing reexecution" to "non-blocking repair".

5.2.1.2 Task 2. Machine Learning based Preventive Damage Management.

In Chapter 4, we propose a naive damage spreading pattern mining method based on association rule algorithm and a probabilistic data integrity estimation approach based on Bayesian network. However, this mechanism has the inherent limitations from the naive association rule algorithm on deciding frequent item set and support. In this task, we propose to comprehensively study other machine learning approaches on damage spreading pattern mining and data object integrity estimation.

Bibliography

[1] M. R. Adam. Security-control methods for statistical database: A comparative study. *ACM Computing Surveys*, 21(4), 1989.

[2] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transaction on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[3] 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.

[4] Michael Atighetchi, Partha Pal, Franklin Webber, and Christopher Jones. Adaptive use of network-centric mechanisms in cyber-defense. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 183, Washington, DC, USA, 2003. IEEE Computer Society.

[5] V. Atluri, S. Jajodia, and E. Bertino. Multilevel secure databases with kernalized architecture: Challenges and solutions. *IEEE Trans. on Knowledge and Data Engineering*, 9(5):697–708, 1997.

[6] Kun Bai and Peng Liu. Towards database firewall: Mining the damage spreading patterns. In 22^{nd} *Annual Computer Security Applications Conference (ACSAC 2006)*, pages 449–462, 2006.

[7] Kun Bai and Peng Liu. A fine-grained damage management scheme in a self-healing postgresql system. In *HASE*, pages 373–382, 2008.

[8] Kun Bai and Peng Liu. A data damage tracking quarantine and recovery (dtqr) scheme for mission-critical database systems. In *EDBT*, pages 720–731, 2009.

[9] Kun Bai, Hai Wang, and Peng Liu. Towards database firewalls. In *IFIP WG 11.3 Working Conf. on Data and Applications Security (DBSec)*, pages 178–192, 2005.

[10] Kun Bai, Meng Yu, and Peng Liu. Trace: Zero-down-time database damage tracking, quarantine, and cleansing with negligible run-time overhead. In *ESORICS*, 2008.

[11] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Int'l Conf. on Extending Data Base Technology*, Mar 2000.

[12] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987. ISBN 0-201-10715-5.

[13] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion detection in rbac-administered databases. In *ACSAC*, 2005.

[14] Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–9, 2002.

[15] CERT. Cert advisory ca-2003-04 ms-sql server worm. http://www.cert.org/advisories/CA-2003-04.html, January, 25 2003.

[16] Qiming Chen and Umeshwar Dayal. Failure handling for transaction hierarchies. In Alex Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 245–254. IEEE Computer Society, 1997.

[17] Tzi-cker Chiueh and Dhruv Pilania. Design, implementation, and evaluation of an intrusion resilient database system. In *Proc. International Conference on Data Engineering*, pages 1024–1035, April 2005.

[18] Johann Eder and Walter Liebhart. Workflow recovery. In *Conference on Cooperative Information Systems*, pages 124–134, 1996.

[19] Prahlad Fogla and Wenke 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.

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

[21] Anup K. Ghosh and Jeffrey M. Voas. Inoculating software for survivability. *Commun. ACM*, 42(7):38–44, 1999.

[22] K. Goseva-Popstojanova, F. Wang, R. Wang, G. Feng, K. Vaidyanathan, K. Trivedi, and B. Muthusamy. Characterizing intrusion tolerant systems using a a state transition model. In *2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.

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

[24] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Trans. on Database Systems (TODS)*, 1(3):242–255, Sep. 1976.

[25] http://www.tpc.org/tpcc/. *TPC-C Benchmark*.

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

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

[28] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *ACM SIGMOD*, pages 474–485, May 1997.

[29] Ramaprabhu Janakiraman, Marcel Waldvogel, and Qi Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *In Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 226–231, 2001.

[30] 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.

[31] Manuel Costa Jon, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms?, 2004.

[32] Angelos D. Keromytis, Janak Parekh, Philip N. Gross, Gail Kaiser, Vishal Misra, Jason Nieh, Dan Rubenstein, and Sal Stolfo. A holistic approach to service survivability. In *SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pages 11–22, 2003.

[33] J. Knight, K. Sullivan, M. Elder, and C. Wang. Survivability architectures: Issues and approaches. In *the 2000 DARPA Information Survivability Conference & Exposition*, pages 157–171, CA, June 2000.

[34] O. Patrick Kreidl, Student Member, and Tiffany M. Frazier. Feedback control applied to survivability: a host-based autonomic defense system. *IEEE Transactions on Reliability*, 52:148–166, 2002.

[35] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *CCS'03*, pages 251–261, Washington, DC, USA, October 27-31 2003.

[36] Caribou Lake. Journal based recovery tool for ingres.

[37] S. Y. Lee, W. L. Low, and P. Y. Wong. Learning fingerprints for a database intrusion detection system. In *ESORICS*, 2002.

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

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

[40] 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.

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

[42] P. Liu and S. Jajodia. Multi-phase damage confinement in database systems for intrusion tolerance. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 191–205, Nova Scotia, Canada, June 2001.

[43] Peng Liu. Architectures for intrusion tolerant database systems. In *The 18th Annual Computer Security Applications Conference*, pages 311–320, 9-13 Dec. 2002.

[44] David Lomet, Zografoula Vagena, and Roger 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.

[45] S. Stolfo A. Keromytis T. Malkin M. Locasto, J. Parekh and V. Misra. Collaborative distributed intrusion detection. technical report cucs-012-04. Technical report, Columbia University Department of Computer Science, 2004.

[46] D. Malkhi, M. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.

[47] 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.

[48] D. Medhi and D. Tipper. Multi-layered network survivability - models, analysis, architecture, framework and implementation: An overview. In *the 2000 DARPA Information Survivability Conference & Exposition*, pages 173–186, CA, June 2000.

[49] Kostas Anagnostakis Michael, Michael B. Greenwald, Sotiris Ioannidis, Angelos D. Keromytis, and Dekai Li. A cooperative immunization system for an untrusting internet. In *In Proceedings of the 11th IEEE International Conference on Networks (ICON*, pages 403–408, 2003.

[50] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter 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.

[51] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. 2003.

[52] Adam J. O'Donnell and Harish Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *CCS '04: Proceedings of*

the 11th ACM conference on Computer and communications security, pages 121–131, 2004.

[53] ORACLE. Oracle database advanced application developer's guide, 2007.

[54] Alessandro Orso, Ro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *In Proceedings of the international symposium on Software testing and analysis*, pages 65–69. ACM Press, 2002.

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

[56] P. P. Pal, J. P. Loyall, R. E. Schantz, and J. A. Zinky. Open implementation toolkit for building survivable applications. In *2000 DARPA Information Survivability Conference (DISCEX)*, June 2000.

[57] 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.

[58] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press.

[59] Roberto Perdisci, Guofei Gu, and Wenke Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *ICDM*, pages 488–498, 2006.

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

[61] Phillip Porras, Linda Briesemeister, Keith Skinner, Karl Levitt, Jeff Rowe, and Yu-Cheng Allen Ting. A hybrid quarantine defense. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 73–82, 2004.

[62] Postgresql. http://www.postgresql.org/.

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

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

[65] Ravi Sandhu and Fang Chen. The multilevel relational (mlr) data model. volume 1, pages 93–132. ACM Press, New York, NY, USA, 1998.

[66] S. Sekar, M. Bendre, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

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

[68] Rumman Sobhan and Brajendra 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.

[69] Salvatore J. Stolfo. Worm and attack early warning. *IEEE Security and Privacy*, 2(3):73–75, 2004.

[70] John D. Strunk, Garth R. Goodson, Adam G. Pennington, Craig A. N. Soules, and Gregory R. Ganger. Intrusion detection, diagnosis, and recovery with self-securing storage. *Technical Report*, 2002.

[71] M. Tallis and R. Balzer. Document integrity through mediated interfaces. In *2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.

[72] Jian Tang and San-Yih Hwang. A scheme to specify and implement ad-hoc recovery in workflow systems. *Lecture Notes in Computer Science*, 1377:484–498, 1998.

[73] F. Valeur, G. Vigna, C. Kruegel, and E. Kirda. An Anomaly-driven Reverse Proxy for Web Applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006.

[74] Fredrik Valeur, Darren Mutz, and Giovanni 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.

[75] F. Webber, P. P. Pal, R. E. Schantz, and J. P. Loyall. Defense-enabled applications. In *2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.

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

[77] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, August 2000.

[78] Qingsong Yao, Aijun An, and Xiangji Huang. Finding and analyzing database user sessions. In *DASFAA*, pages 851–862, 2005.

Vita

Kun Bai

Education

The Pennsylvania State University State College, Pennsylvania 2004–2010

Ph.D. in Information Sciences and Technology

Area of Specialization: Database Security

University of Alberta Edmonton, Alberta, Canada 2001–2003

M.S. in Computer Science

Area of Specialization: Computer and Communication Networks

Huazhong University of Science and Technology Wuhan, Hubei, P. R. China 1993–1997

B.S. in Computer Science

Research Experience

Doctoral Research The Pennsylvania State University 2004–2010

Thesis Advisor: Prof. Peng Liu

Graduate Research The University of Alberta, Canada 2001–2003

Research Advisor: Prof. Ehab Elmallah

Teaching Experience

Teaching Assistant The Pennsylvania State University 2005, 2006

I taught computer security courses and graded assignments and exams.

Teaching Assistant University of Alberta, Canada 2001–2003

I taught OO programming and computer networking courses.