Data Recovery for Web Applications

by

İstemi Ekin Akkuş

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2009 by İstemi Ekin Akkuş

Abstract

Data Recovery for Web Applications

İstemi Ekin Akkuş

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2009

Web applications store their data at the server. Despite several benefits, this design raises a serious problem because a bug or misconfiguration causing data loss or corruption can affect a large number of users. We describe the design of a generic recovery system for web applications. Our system tracks application requests and reuses undo logs already kept by databases to selectively recover from corrupting requests and their effects. The main challenge is to correlate requests across the multiple tiers of the application to determine the correct recovery actions. We explore using dependencies both within and across requests at three layers, (i.e., database, application, client) to help identify data corruption accurately. We evaluate our system using known bugs and misconfigurations in popular web applications, including Wordpress, Drupal and Gallery2. Our results show that our system enables recovery from data corruption without loss of critical data incurring little overhead while tracking requests.

Acknowledgements

I would like to thank to my advisor Prof. Ashvin Goel for his guidance, time and patience. I would also like to thank to Prof. David Lie, Prof. Cristiana Amza and Prof. Raymond Kwong for being a member in my committee and for their corrections and suggestions to improve my work. I also want to thank to the Department of Electrical and Computer Engineering at the University of Toronto for its financial support.

Without my friends and colleagues, I would not have come this far. I want to thank Gokul Soundararajan for his help, thoughts and suggestions on my work; Vivek Lakshmanan for his suggestions and for listening to my random rants; Vladan Djeric for trying to cheer me up and give me different perspectives looking at things and Stan Kvasov, Lionel Litty, Lee Chew, Thomas Hart, Maxim Siniavine and Eric Chen for making my experience more enjoyable, be it with coffee breaks, sushi lunches, barbecue parties or bar crawls. I also want to acknowledge Adrian-Daniel Popescu, Adam Czajkowski, Bogdan Simion, Daniel Lupei, Madalin Mihailescu, Pouya Alagheband, Saeed Ghanbari, Jin Chen, Weihan Wang, David Tam, Zoe Chow and many others in the Computer Systems Lab.

Finally, I would like to thank my family. Even though we are thousands of miles apart, I have always felt their support and known that they would always be there for me, no matter what.

Contents

1 Introduction **1**

1.1 Contributions 7

1.2 Outline 7

2 Related Work **8**

3 Approach **11**

3.1 Application Model 11

3.2 Overview 13

3.3 Monitoring 15

3.4 Analysis 17

3.4.1 Database Dependencies 17

3.4.2 Application Dependencies 20

3.4.3 Client Dependencies 22

3.4.4 Tainting 23

3.4.5 Replaying Requests 24

3.4.6 Dependency Policies 25

3.4.7 Whitelisting 26

3.5 Recovery 28

3.6 Selective Recovery 28

4 Implementation **31**

4.1 Monitor 31

4.2 Analysis 33

4.2.1 Query Rewriting 34

4.2.2 Handling of Blind-Writes 35

4.2.3 Propagation of Taint 35

4.2.4 Whitelisting 36

4.3 Recovery 36

4.4 Limitations 37

5 Evaluation **39**

5.1 Experimental Setup 39

5.2 Recovery Accuracy 40

5.2.1 Correct Recovery Actions 40

5.2.2 Wordpress Functionality Bug: Renamed Link Categories 42

5.2.3 Drupal Data Loss Bug: Lost Comments 47

5.2.4 Drupal Data Loss Bug: Lost Voting Information 50

5.2.5 Gallery2 Functionality Bug: Removing Permissions Breaks the Application 52

5.2.6 Gallery2 Functionality Bug: Resizing Images Breaks Existing Links to Images 54

5.2.7 Remarks 56

5.3 Performance 56

5.3.1 Throughput Overhead 58

5.3.2 Disk Space Overhead 59

6 Conclusion **61**

Bibliography **61**

List of Tables

4.1 Modifications to existing software 32

4.2 Query rewriting examples with row-level tainting 33

4.3 Query rewriting examples with field-level tainting 34

5.1 Recovery accuracy for request-level and program-level dependency policies 43

5.2 Recovery accuracy of database-level dependency policies 44

5.3 Disk space overhead 59

List of Figures

3.1 Dependencies across different layers of the application. 14

3.2 Row-level and field-level tainting example 19

3.3 A request dependency graph. 21

5.1 Throughput results. 57

5.2 Latency results. 58

Chapter 1

Introduction

Web-based applications generally store persistent data on the server, enabling client mobility, simpler configuration and improved data management. These applications are increasingly being designed for extensibility and to support a plugin architecture, allowing third-party developers to rapidly introduce additional features and provide enhanced services and customization. However, this design can lead to an application bug or a single misconfiguration affecting a large number of users, potentially causing data loss or corruption. Third-party plugins may be poorly tested, may cause problems with other plugins, or even worse, corrupt user data. For example, administrators of the Wordpress blogging application [16] are generally advised to back up all data before installing any new plugins or new versions of the application [17].

Data corruption and recovery pose especially serious challenges for web applications, since these applications may store important user data and configuration settings. For example, Wordpress can be configured to store arbitrary user data, and can even embed other web applications, such as the powerful Gallery [39] photo application that allows storing and sharing personal photos with specific users. However, Wordpress has had several vulnerabilities [19, 20, 22] that can cause data loss or corruption. A typical solution to this problem is to restore data from a backup. However, this approach loses

updates that occur after the backup is performed, and then these updates must be recovered manually or via some ad-hoc methods. Furthermore, a backup solution does not help diagnose the external or application events that caused the problem. As a result, currently much of this diagnostic work needs to be done manually, which is time consuming and error prone.

An alternative to backups is to use application-specific recovery features. A common example is a simple undo functionality, available in web applications such as Google Mail and Google Docs. This feature allows the user to undo her last action, enabling recovery from simple misconfiguration problems or accidental clicking. While this feature is useful, its implementation requires significant modifications to applications, because the developer needs to carefully design and implement undo functionality for all actions available to users. This problem only becomes worse for applications with extensible functionality. As a result, while there has been significant interest in developing undo for web applications [8, 9, 10, 18], these applications often do not provide such a facility.

A more significant problem with the simple undo approach is that it is not useful for the user when the corruption is detected much later than it occured. In this case, the user may have to undo and redo various actions manually, or the undo may not be available (e.g., the user navigates away from the web page where the action occurred in Google Mail), or it may not work at all because the failure may have propagated and affected other users. For example, Gallery2 bug number 2016834 [12] prevents all users, including the administrator, from accessing the application interface and thus prevent her from undoing her last action, even if the undo feature was implemented.

Perhaps the most serious problem with application-specific recovery is that it depends on the correctness of the application. If a data loss or corruption is caused by a buggy application, undo may not even work correctly and may cause further corruption. For instance, bug number 67745 [6] in Drupal [4], a popular content management system, causes all comments on the site to be deleted if two administrators try to delete the same

comment. If undo were available, it would restore this comment, but all other comments would be lost because the developer did not expect them to be deleted.

In this thesis, we describe the design of a generic data recovery system for web applications. These applications generally store their persistent data in a database tier. Our main goal is to provide tools that will allow web application administrators to diagnose and recover from application failures that cause persistent data stored in the database tier to be corrupted. Specifically, we focus on helping the administrator identify the persistent data corrupted by an application bug or a misconfiguration, and selectively recovering this data without affecting the rest of the application.

A significant challenge with this selective recovery idea lies in identifying data corruption and its dependent effects accurately. For example, in a recent case, a major electronics retailer experienced a misconfiguration, so that the price of one of its products was entered incorrectly [3]. This price had to be fixed and all dependent purchases involving this product had to be cancelled. Another online retailer had to shut down its services after a similar pricing error [2] to determine the dependent effects of the pricing misconfiguration. Many similar examples [1, 5, 7] show that determining corruption and its dependent effects accurately is an important part of the recovery process.

These dependent effects can be captured using a combination of dependencies at the different tiers of the application. At the database tier, a query might read a row that was written by another query and update a third row, thereby creating a causal dependency between the queries. Similarly, a transaction might read a row updated by a previous transaction and update others. Foreign key constraints can also be the cause of a dependency, where an update or delete to a row in a table causes other referencing tables to be updated [15].

At the application tier, causal dependencies can occur either within a request or across requests. A request (i.e., HTTP request) is an action of the user at the presentation layer triggering application logic to execute. A causal dependency within a request

occurs when a request issues a query that reads a row, uses that information to do some computation and uses the computed values to update other rows. This dependency cannot be captured at the database layer itself. A recovery system should be able to handle these dependencies because if these dependencies are ignored, the application state may become inconsistent. For example, Gallery2 assigns a global ID to all objects in the application. The sequence ID counter is updated and the new sequence value is used when inserting a new object. If this dependency is ignored, the recovery system may revert the sequence ID counter to its old value without reverting the object insertion. This will cause a problem with future insertions which will attempt to use an already-in-use ID.

Dependencies across requests are created when a request reads rows that another request has updated. This kind of dependency is similar to transaction-level dependencies in the database. However, web applications often do not use a transaction for an entire request to improve performance. For example, a request that adds a new link associated with a category in the Wordpress blogging application will read the row with the category information and therefore is dependent on the previous request that created the category. Missing these dependencies may effect the accuracy of the recovery, resulting in corrupted data lingering in the application, potentially causing more problems.

Another type of dependency is a foreign key constraint implemented at the application level. Most web applications handle these dependencies manually because they do not make any assumptions about the underlying storage engine, which may or may not implement foreign key constraints. For instance, Gallery2 uses a global table to keep track of individual items and another table for maintaining parent-child relations. A recovery action deleting an item from the global table, but not from the relations table, would cause an inconsistency, in which certain items would have a non-existent parent item.

Certain dependencies such as login sessions and user accounts can only be tracked

at the presentation tier and create client-level dependencies. Web applications usually store an identifier for each user in a database table and query it when a user logs into the application, assigning her a handle on user-specific state. This handle is used by later requests of this user as one of the parameters submitted to the application (e.g., cookies). Requests belonging to the same session or the user may not be considered as dependent at the application and the database layers.

Using dependencies for data recovery has been explored previously in the context of transactional databases [24, 29, 37] and file systems [32, 52]. However, these techniques are applied at a single layer, making them unsuitable for web applications. For example, transaction-level dependency analysis [24, 29, 37] cannot be directly applied to web applications for two reasons. First, web applications may not use transactions, so the recovery system cannot depend on their existence. Second, these applications operate at multiple layers. By ignoring the interactions across these layers, these approaches ignore related actions, potentially causing false negatives and inconsistencies during recovery. Our evaluation in Chapter 5 illustrates this problem.

Our recovery system correlates dependencies across different layers, namely at the presentation, application-logic and database tiers, thus helping diagnose data corruption more accurately. It does not rely on the web application for recovery and thus is resilient to failures and bugs in the applications. Our system monitors all the tiers of the web application to determine the effects of data corruption. This approach also allows us to provide last action undo functionality without making any changes to existing applications. Our recovery method takes advantage of several characteristics of web applications to track data corruption and provide a generic recovery method for web applications. These characteristics include a clean separation of the application into presentation, application-logic and database tiers. Since the application-logic is written in a high-level language and all persistent data is stored in the database-tier, it is easier to monitor the application as explained later, without modifying it.

Although our recovery system can correlate dependencies across all the tiers of the application, it can not recover from certain types of application bugs. For example, we do not handle bugs that occur at the presentation tier for two reasons. First, these 'views' of application states are temporary as they are generated every time a page is generated. Second, they are generated using the data that is stored persistently in the database, so focusing on the recovery of persistent data corruption makes a recovery system more usable. Also, if a bug in the application logic misses an update to the database, our recovery system can not handle it because the update never occurred.

We explore using several dependency policies at the different layers of the application and apply them to real bugs and misconfiguration in web applications and describe the benefits and drawbacks of these policies. A potential drawback of these policies is that they can create false positives, in which case dependent requests that are essentially independent are marked as dependent, leading to loss of good work. We identify and explain the causes of these false positives and suggest methods to reduce this problem. More importantly, we argue that the use of multiple policies helps the administrator determine the corrective recovery actions more rapidly.

Our recovery system uses two novel dependency analysis techniques. First, it combines application replay with offline taint analysis. Tainting has generally been used online for securing applications [40, 42, 51], but we are not aware of its use for data recovery. Second, we explore the benefits of finer-grained field-level dependencies at the database-tier than existing approaches that use row-level tainting [24, 29, 37]. These techniques help the administrator identify the data corruption more accurately.

We have implemented a prototype of our recovery system for the PHP interpreter and the MySQL database. PHP and MySQL are widely used by existing web applications for generating dynamic web pages and for storing application content and settings. We have tested our system on popular web applications including the Wordpress blogging application [16], Drupal content management system [4] and Gallery2 photo application

[39]. We evaluate the effectiveness of our approach for various data corruption scenarios that can be triggered by known bugs and misconfigurations in these applications.

1.1 Contributions

In this thesis, we make the following contributions.

- We provide a generic recovery system for web applications, that is resilient to bugs, failures and misconfigurations.
- We provide the simple undo functionality without any changes to the applications.
- We correlate dependencies acrosss layers and explore several dependency policies in real web applications. These policies employ application-level tainting and database field-level dependency analysis. We identify strengths and drawbacks of each policy and suggest methods to mitigate these drawbacks.

1.2 Outline

In the next chapter, we describe related work. Chapter 3 explains the details of our approach. Chapter 4 describes our prototype implementation. In Chapter 5, we evaluate the effectiveness of our approach for various data corruption scenarios in popular web applications. Chapter 6 presents our conclusions.

Chapter 2

Related Work

Liu et al. [37] initially proposed a method for recovering from malicious transactions based on tracking inter-transaction dependencies. These inter-transaction dependencies are created by examining the read and write sets of transactions. The attacking transaction and effected transactions are moved to the end of the transaction history to simplify recovery. Their follow-up works proposes a system in which normal operation is allowed while recovery is performed [24]. Similar recovery methods have been proposed in Fastrek [29, 43] and the Flashback Database[47]. These methods focus entirely on database-level recovery while ignoring application-level dependencies, which can cause inconsistent recovery at the application level. Our system tracks application-level dependencies during recovery by employing dynamic data-flow (i.e., tainting) within requests rather than just relying on the read and write sets of queries and requests, avoiding application-level inconsistencies and tracking data corruption more accurately with less false positives. The next chapter compares these approaches in more detail.

Compensating transactions have been used to recover from the effects of long-running or committed transactions [34, 36] and for recovery in multi-level systems that are designed to increase concurrency [49, 38]. We also use compensating transactions to perform recovery, but our focus is on recovering from application bugs or vulnerabilities that cause

data corruption, and we target web applications that may not use transactions.

File system backups and snapshot-based file systems are commonly employed for recovering from data corruption. However, these methods revert data based on time and can lose important or legitimate updates that have occurred since the backup was taken. Selective file-system recovery aims to solve this issue via a set of dependency rules [35] that taint certain file updates, and then the effects of only the tainted updates are reverted [32, 52]. This method is too coarse grained for database-based applications, because databases may save all information in a single file. The recovery operation would simply generate an older version of the database file, suffering from the same drawbacks as the file-system snapshot approach.

Self-securing storage devices [44] internally audit all requests and keep old versions of data for a window of time, regardless of the commands received from potentially compromised operating systems. This approach can be used to secure the logs in our system against OS exploits.

Operator Undo [26] is a powerful framework for recovering from application bugs. The authors use it to recover from e-mail server configuration failures, but the framework requires modifying applications to ensure that requests can be serialized and replayed, and persistent data is stored separately from applications. It also requires special recovery procedures for each type of application request. By focusing on web applications that have well-defined interfaces, we are able to provide similar functionality without modifying applications.

Causeway [27] provides operating system support for metadata (e.g., request id) transfer across the tiers of an application. Applications have to be modified to provide the metadata information. This information allows requests to be prioritized at each tier. Our work utilizes the well-defined interfaces in a web application's tiers and passes metadata across tiers to log and correlate them later without requiring any modifications to the applications. A similar idea was used by Magpie [25] where the request execution

paths were used to diagnose application failures.

There have been several proposals for using dynamic taint analysis for securing web applications. Nguyen-Tuong et al. [42] modify the PHP interpreter to dynamically track tainted data in PHP programs, and Haldar et al. [33] apply a similar approach to Java. WASC [40] is a compiler that adds taint checking in web applications to project against SQL and script injection attacks. An alternate approach is to use static taint propagation to detect vulnerabilities statically [51]. These approaches require setting initial sources of taint either statically or during execution, and typically all network and user inputs are marked tainted. We do not use tainting during normal operation. Instead, we follow the effects of a bug or an attack during analysis and recovery.

Replay has been used extensively for analysis of intrusions. ReVirt [31] performs replay at the machine level using a virtual machine monitor, allowing detailed analysis of vulnerabilities and intrusions, but it does not have any support for data recovery. The RolePlayer [30] replays the client and the server sides of a session for a variety of application protocols when given examples of previous sessions for network intrusion analysis.

Our work is also motivated by software configuration problems that cause data corruption. Several approaches have been proposed for dealing with configuration problems. PeerPressure [48] uses statistical analysis of multiple systems to find and suggest a working configuration. Chronus [50] pinpoints when a configuration problem occurred by using predicates that determine whether the system is working correctly. AutoBash [45] aims to detect configuration problems and suggest corrective actions based on causality analysis.

Chapter 3

Approach

This chapter describes the design of our data recovery system. Our aim is to help the administrator to identify the persistent data corrupted by a bug or a misconfiguration in a web application, and selectively recover this data without affecting the rest of the application. Our system consists of a monitoring component that operates during normal application operation (on-line phase), and two components that perform analysis and data recovery after corruption is detected (post-corruption phase). Below, we first present the application model assumed by our recovery system, then provide an overview of our system and describe each of the components of our system in more detail.

3.1 Application Model

A web application is typically designed using a three tier architecture, consisting of the presentation, application-logic and database tiers. A user or an administrator interacts with the web application by issuing *requests*, which are external actions at the presentation (or client) layer that invoke the application logic. The application logic executed by each request makes database queries or transactions to access application data and configuration information.

Our recovery system takes advantage of several features of web applications to track

bug-related activities and data corruption. First, most web applications store their persistent data in databases for concurrency control and easy search capabilities, which allows reusing the database logs for tracking the persistent modifications made by the application. Second, web applications are generally written in high-level or type-safe languages such as PHP or Java, allowing easier monitoring of the application. For example, an unmodified PHP application can be monitored by instrumenting the PHP interpreter, rather than requiring binary rewriting or source-code modifications for instrumentation.

Third, web servers treat each user request independently, often creating a separate process per request to ensure isolation, and any interaction between requests occurs through database queries. In contrast, full-blown OS processes have numerous IPC and shared memory mechanisms available for communication that not only make it hard to monitor the application [35], but these channels can also cause contamination to spread more easily [32]. Finally, web applications have a simple and well-defined interface that is mostly limited to requests and database operations. This interface makes it easier to replay requests to the application, since there are fewer sources of non-determinism. We use replay because it allows tracking data dependencies more accurately than previous methods.

Our recovery system assumes that the database and the application-logic engine (e.g., the PHP interpreter) are not buggy, and data is corrupted at the database layer due to bugs or misconfigurations in the application-logic or in the presentation layer. Our system also assumes that the underlying database supports transactions so that the database undo logs are generated and can be used for recovery. The web application may or may not explicitly use transactions, but if it does not, each query will be treated as a separate transaction at the database via the 'autocommit' feature. Our system does not immediately purge the undo log entries for a transaction after the transaction is committed. Instead, the log entries are purged after a user-configurable time. Transactions that occur before this time are considered stable and their effects cannot be reverted. Fi-

nally, we assume that the database uses a serializable isolation level so that the database transactions can be replayed correctly.

3.2 Overview

Our recovery system executes in three phases, an on-line monitoring phase, a post-corruption analysis phase and a recovery phase. The monitoring component is relatively lightweight, and broadly speaking, it tracks user (or administrator) requests across the three tiers of the application, namely at the presentation, application-logic and the database tiers, allowing data recovery at the request granularity. This method of tracking requests has similarities with the use of request paths for diagnosing application failures [25, 28]. Monitoring the application and tracking requests at all these tiers gives our system the ability to perform generic recovery.

The analysis and recovery components are used after corruption is detected, such as an administrator determining that a web page does not display as expected. These components use the data collected during the monitoring phase, including database logs, to help guide the administrator through the recovery process. The analysis component helps the administrator determine corruption related events, and is crucial for effective recovery.

Figure 3.1 shows the dependencies that are created at each tier of the application. Our analysis component makes use of three types of dependencies, at the database, the application and the presentation or the client side. First, it captures database dependencies at the row or field granularity based on the database rows or fields accessed by the application logic. These dependencies help correlate different requests based on the database operations performed by the requests, similar to existing approaches. These dependencies are coarse-grained because they are applied at a request granularity, but they help determine a maximal set of dependent requests.

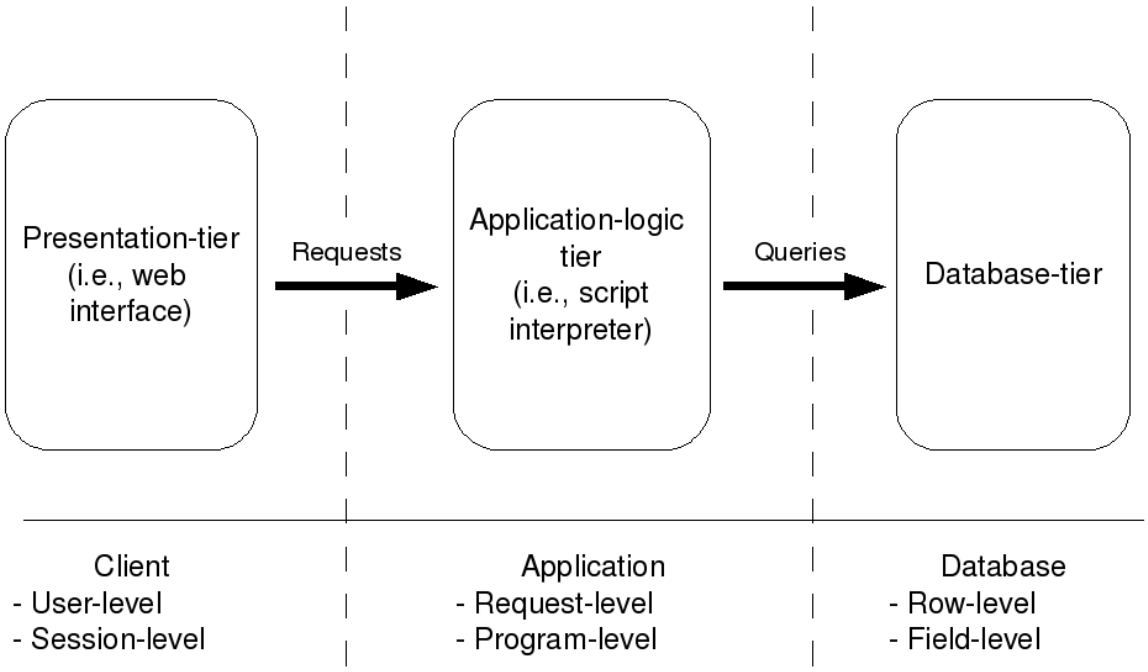

Figure 3.1: Dependencies across different layers of the application.

Second, the analysis component uses dynamic data-flow based tainting to generate application-logic data dependencies within a request (shown as program-level dependencies in Figure 3.1), which helps reduce false dependencies across requests that were generated previously. While tainting has been used extensively to detect whether untrusted data affects sensitive operations [33, 41], we are not aware of its use for data recovery. Furthermore, while tainting is normally used on-line to secure all code execution, the analysis component only uses tainting during the post-corruption phase on requests that are considered dependent on one or more initial tainted requests. It performs tainting by replaying the dependent requests, which requires capturing the request parameters during the monitoring phase.

Finally, the analysis component uses client-side dependencies across requests, such as login sessions and user accounts. For example, session cookies identify all requests associated with a login session. These types of dependencies provide a useful abstraction,

because they can help provide different starting points for the analysis. For instance, an administrator might know that the data corruption started with a specific user and start the dependency analysis by initially tainting all modifications by this user. This abstraction may also be useful for recovery. For instance, an administrator may wish to revert all the effects caused by a session, if she knows that the session is responsible for the data corruption and there are no other dependencies.

The recovery component provides various tools that simplify the recovery process. These tools provide information, such as the time line of requests and sessions that affected specific database tables or generated web pages, helping the user identify specific requests or sessions that are the root cause of the failure. These act as the starting point for our tainting analysis, where the analysis component described earlier determines the dependencies. Finally, the recovery component uses the information in the database log to generate compensating transactions that selectively revert the effects of the database operations issued by the tainted requests that caused the data corruption.

3.3 Monitoring

The monitors track and correlate requests across all the tiers of the application, allowing request-level data recovery. We chose requests as the minimal granularity for recovery, because they are the smallest logical unit of application interaction (i.e., applications execute code at the granularity of requests), and they are relatively independent. In essence, we convert a request into a transaction during recovery, thus avoiding application-level inconsistencies after the recovery operation, as we will explain later in Section 5.2.1. Requests serve two other purposes in our system. First, our system uses them to generate an initial dependency graph that provides a list of tainted requests. If our system used transactions to generate dependencies, it would miss dependencies across transactions within the same request. Second, our system replays requests to prune false dependencies in the

dependency graph, as described below.

The monitors log sufficient information to allow mapping each request to database transactions, and transactions to specific tables and rows that were modified. These *request* and *transaction-level* mappings, together with the database undo log, allow selectively reverting the effects of all persistent data modifications performed by a request, as described in the rest of this chapter.

The monitor at the database tracks the physical rows that were written by a transaction by instrumenting the database, since concurrent transactions can interleave and the precise execution order is only available within the database. Database modifications are stored in the undo log and our transaction-level mapping is an index into this log, with the key of the index being the commit log sequence number (LSN) of the transaction. This key value, called transaction ID, is ordered in transaction execution order, since we assume that the database uses a serializable isolation level. This ordering is important for replaying requests as described later.

The database, however, does not have application-specific information, such as the transactions generated by a request, or by a specific user, and it cannot distinguish between transactions issued by an explicit user request (e.g., form submission) and an automated system operation (e.g., maintenance scripts that are embedded in an image and executed when the browser loads the image). Fortunately, this information can be obtained by monitoring the application. We instrument the application-logic engine, such as the PHP interpreter, to log the transactions issued by a request. Specifically, the monitor at the application-logic engine generates a request-level mapping by assigning a unique identifier to each request, called request ID, and logging the transaction ID of all transactions issued by each request. This mapping also logs the queries issued by each transaction and some application-specific information described later in this section. None of this instrumentation requires any changes to the application code.

3.4 Analysis

The analysis component helps determine data corruption or loss related activities, and is crucial for effective recovery. It operates during the post-corruption phase. Before starting the analysis, the current state of the application (i.e., database tables) is saved. The analysis is performed in a sandbox environment, so that it will not affect the current state. After the analysis, the recovery actions can be performed on the previously saved state of the application. The analysis component uses the data collected during the monitoring phase to derive three types of data dependencies, at the database, program and the client level as shown in Figure 3.1. These dependencies help track contaminated data across the multiple tiers of the application.

The following sections describe each of these types of dependencies in more detail. Next, we present our replay-based tainting (i.e., dynamic data-flow) approach for identifying the effects of data corruption. Then, we present several dependency policies that we provide to the administrator during the analysis phase to help identify the effects of data corruption more rapidly. Finally, we describe our selective recovery component which is invoked after the data corruption has been identified accurately.

3.4.1 Database Dependencies

We generate data dependencies across requests based on database accesses. A query Q2 is dependent on another query Q1 when Q2 reads data written by Q1. Similarly, a request R2 is dependent on request R1, when R2 contains Q2 and R1 contains Q1. These dependencies help generate a dependency graph with requests as nodes and edges as data dependencies as shown in Figure 3.3.

The analysis component needs to know the read set and the write set of each query to generate a dependency. These sets can be at the granularity of a table, a column, a row or an individual field in a row, with finer granularity causing fewer false dependencies, but

requiring more bookkeeping. As described earlier, the monitor captures row-level write sets, because the database already maintains undo information at the row level. However, databases generally do not log read sets, because they do not need this information and this logging can impede performance. This problem has been addressed previously in a similar context in two ways, other than simply logging the read sets. The first is to create a read-set template for each query, and then materialize the rows read by the query based on the parameters passed to the query [24]. However, this method requires manual creation of a template for each query issued by the application. A second method consists of instrumenting the database to generate and store the dependencies during the on-line phase [29]. This approach generates dependencies more accurately, but affects performance during normal operation.

Our analysis component generates dependencies during the post-corruption phase, using a method similar to read-set templates, but without requiring manual creation of templates. It derives an approximate, but conservative estimate of the query read set by parsing the query and determining the tables accessed. This simple method for generating read sets results in a larger dependency graph compared to the previous approaches. However, we show in later sections that a larger dependency graph only affects the time to perform recovery, but not the overall accuracy of our solution.

The dependency graph is generated as the read and write sets of queries are determined. The queries are examined in transaction ID order, and the graph generation requires a single linear pass over all queries that have been issued since the time when the corrupting request was issued.

Blind-Writes

A blind-write is a database operation in which a query overwrites the value of a field without reading it. Previous approaches, focusing on security, convert blind-writes to regular writes, namely they assume that each write operation reads the value before

row-level tainting:

// taint the row

q1 = UPDATE table1 SET field1 = 5, taint = 1 WHERE id = 1

// reset the taint for the row (lose the taint)

q2 = UPDATE table1 SET field2 = 10, taint = 0 WHERE id = 1

field-level tainting:

// taint field1

q1 = UPDATE table1 SET field1 = 5, field1_taint = 1 WHERE id = 1

// reset the taint for field2; field1 is still tainted

q2 = UPDATE table1 SET field2 = 10, field2_taint = 0 WHERE id =1

Figure 3.2: Row-level and field-level tainting example. Row-level tainting cannot take advantage of blind-writes and has to be conservative. Field-level preserves taints for each field separately.

writing to it [24, 37]. These algorithms work by moving malicious transactions and their dependent transactions to the end of the transaction history and reverting their effects. A transaction G is assumed to be independent of a previous malicious transaction M if the intersection of the write set of M and the read set of G is empty. In this case, M can be undone without affecting the result of G. However, this result is not true if G performs a blind write, since undoing M may overwrite the value written by G. Managing blind writes increases complexity in these algorithms and so blind writes are converted to regular writes.

We believe that blind-writes can help break false dependencies and thus reduce the amount of work that is lost due to false positives. Our experience showed that many web applications use blind-writes. Since a blind-write overwrites a value without reading it, it allows resetting a row or field taint in our tainting algorithm. A row-level blind write

requires updating the entire row or else taint can be laundered. Consider the following example shown in Figure 3.2. If query q1 updates a field (`field1`) with a tainted value (5) for a row, the row will be marked as tainted. If another query q2 updates another field (`field2`) in the same row with an untainted value (10), then the taint information will be lost, even though the tainted value (5) is still stored in the database. Field-level tainting does not suffer from this issue as shown in the second part of Figure 3.2.

This example shows that row-level tainting provides fewer opportunities for breaking taint dependencies than field-level tainting. We provide both levels of tainting to compare field-level tainting with previous approaches using row-level tainting [29, 43] to propagate the taint in the database.

3.4.2 Application Dependencies

The dependencies described above apply to entire requests and they are tracked transitively. This approach is coarse-grained and can potentially generate many false dependencies. Such dependencies occur for two reasons. First, the analysis component tracks query read sets conservatively, as described above. Second, a request can issue multiple queries that may have no dependencies. For example, Figure 3.3 shows that request R2 depends on request R1, and R3 depends on R2, and thus R3 is also assumed to depend on R1. However, this dependency may not exist if, for example, R2 immediately discards the value that it read from R1 using Q2, while R3 only depends on Q4.

We propose using dynamic tainting to track program or application-logic dependencies within a request to prune *both* these types of false dependencies from the dependency graph. Program dependencies are caused by data flow in the application logic and can be captured by instrumenting the application-logic engine. This approach essentially validates a cross-request dependency.

Normally, tainting is used to detect whether untrusted data is used in sensitive operations without being properly sanitized, and it is performed during application execution.

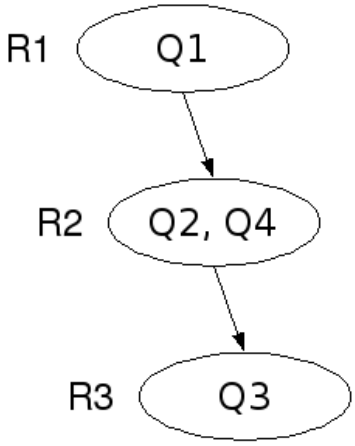

Figure 3.3: A request dependency graph.

For example, all variables assigned using untrusted network or user input can be marked with a tainted bit and this bit is propagated using data flow. An attack is detected when a sensitive operation, such as an SQL write query, uses tainted variables.

Our requirements are slightly different, which makes it hard to perform tainting during the on-line phase. Specifically, our aim is to determine given two requests R1 and R2, whether a write query Q2 issued by R2, depends on a write query Q1 issued by R1. If this dependency is established and R1 is tainted, then the effects of Q2 need to be reverted [1]. If we performed tainting on-line, then R1 and R2 could be arbitrary requests, and as a result, each write query issued by any request would need to be tracked independently. In other words, a taint bit would be needed for every write query, and these bits would have to be maintained and propagated for all program variables and database rows. Even with row-level tainting instead of field-level at the database-tier, this approach is clearly hard to implement and will not scale well. Instead, we avoid the problem described above by using replay-based tainting during the post-corruption phase, as described in Section 3.4.4.

[1] Note that this dependency can also occur via intervening read queries.

3.4.3 Client Dependencies

Besides database and application dependencies, our recovery system also allows tracking certain client dependencies across requests, such as login sessions and user accounts. These dependencies allow grouping requests, providing a useful abstraction for starting the analysis. An administrator can specify whether session or user dependencies should be used to generate the initial set of tainted requests. For example, an administrator can identify a user as the initial cause of the data corruption, so she would start the analysis by marking the requests belonging to that user. Another use of this type of dependency is for performing recovery. For instance, an administrator may determine that requests in a particular session caused the corruption and may wish to revert all effects caused by the session, assuming there were no future dependencies.

These client dependencies are generally not available at the application-logic or database level. For example, web applications generally store an identifier for each registered user in a database table. After a user logs into the application, requests read this table to obtain a handle on user-specific state. These read requests will be considered independent at the database and the application-logic tiers. Instead, we derive these dependencies by using application-specific code in our monitor component, but without requiring any changes to the applications themselves. Specifically, web applications use session and user information for reasons such as authentication or handling concurrent requests. This information is available in request parameters (e.g., URL has session id), or request headers (e.g., cookies), or database tables storing user or session information. The PHP monitor executes application-specific code to derive this information, and then logs this information with each request. As a result, the analysis component can derive the set of requests associated with each session or user.

3.4.4 Tainting

After corruption is detected, the administrator uses our recovery tools to identify one or more initial requests that trigger the bug or vulnerability in the application. With these initial requests, our system generates a coarse-grained dependency graph rooted at this request using database dependencies, as described earlier. This graph provides a maximal set of dependent requests. However, edges in this graph can be false dependencies, which we then prune using tainting.

Our system performs tainting by replaying requests in the dependency graph to a taint-based PHP interpreter. We have modified the interpreter to taint an application variable that reads a tainted database row or field, and taint database rows or fields that are modified by queries using tainted application variables. We store the taint information persistently at the database via helper fields added to the tables. For row-level tainting granularity, a taint field is associated with each row. Similarly, for field-level tainting, each field has its own taint field. Specifically, all the rows or fields modified by a query are marked tainted via their associated taint fields, if the query uses a tainted variable. We provide more details of our implementation in Chapter 4.

The taint analysis component starts by tainting the initial requests and replaying them. It then replays requests that have incoming edges in the dependency graph and uses tainting to prune outgoing edges that are false dependencies. Recall that a request in the dependency graph has an incoming edge when it reads database rows that were modified by the request located at the source of the edge. When using tainting, at least one of the rows associated with an incoming edge will be tainted. After a request is replayed, all the outgoing edges of the request are pruned, if none of the write queries issued by the request were tainted. If even a single write query was tainted, then all the write queries are marked tainted, because our system performs recovery at a request granularity.

This post-corruption analysis associates a single taint bit with each application vari-

able and database row or field, because it has a well-known starting point, i.e., the initial requests, and its goal is to identify other requests that depend on these requests.

3.4.5 Replaying Requests

Our system performs tainting while replaying requests, which requires capturing all request parameters. These parameters are logged by our application-logic monitor, which captures all HTTP request parameters, including the parameters of POST and GET requests. POST requests are used to submit data from clients to the server and typically access and modify database state. GET requests are normally used to obtain data from the server. Our system tracks GET requests, because some web applications use GET requests for modifying state and also because we allow recovery at higher application-specific granularities, such as login sessions, that can span multiple dependent requests, as described in Section 3.4.3.

The replay process needs to use a database snapshot that captures the database state just before the earliest initial tainted request was executed. It recreates this snapshot by rolling back the database state using the recovery tools described later in Section 4.3.

The analysis component replays requests by issuing the queries or transactions in the request issued in the correct serialization or transaction ID order. A scheduler orders all the requests in the dependency graph in request ID order. It replays requests in this order, but ensures that a request is delayed when it issues a transaction with an ID that is not in the correct serialization order. The request ID and the transaction ID information is available in the request-level mapping captured by the monitor (see Section 3.3). During replay, when the pruning of edges causes one ore more nodes to be disconnected from the graph, then those requests are removed from the scheduler queue. This approach of initially generating a coarse-grained dependency graph and then successively pruning the graph allows replaying the minimal number of requests. Note that requests that are not in the dependency graph do not need to be replayed because, by definition, they do not

affect and are not affected by the requests in the graph.

A web application is easier to replay deterministically than arbitrary OS processes, because application requests are relatively independent. Each request is handled by a separate process that is either discarded or re-initialized when handling another request, and requests interact with each other mainly through the database, so rolling back the database recreates the original state for replay. Even so, it is possible that our recovery system does not capture all non-deterministic application behavior. Our system detects any non-determinism while replaying the requests by comparing the queries generated during replay with the queries logged by the monitor. If an inconsistency is detected, we currently ensure safety by aborting the entire analysis process.

3.4.6 Dependency Policies

The aim of our analysis tools is to provide sufficient information to the administrator to identify data corruption. To this end, our tools provide support for the different dependency policies described below. We believe that the outputs of these different policies will help the administrator determine the effects of the corruption and choose the correct recovery actions.

1. **Request-level dependency with row-level tainting granularity (request-row):** This policy is the most conservative dependency policy. It assumes that a request is tainted, if it reads a tainted database row. All further database updates by the request are marked as tainted regardless of whether tainted information is used to update the database.

2. **Program-level dependency with row-level tainting granularity (program-row):** This policy takes application-level data flow into consideration when generating the dependency graph. During a request, all variables that are initialized using a tainted row are marked tainted and the taint is propagated throughout the

request. Whenever a query with the tainted values is executed, the taint information is saved in the database at a row granularity, which preserves taints across requests.

3. **Database-level dependency with row-level tainting granularity (database-row):** This policy propagates taints when queries read tainted rows and update other rows. Since this policy does not regard application-level dependencies, such as the dependencies between the queries of a request, it may fail to identify all the effects of the corruption. Also, the recovery process reverts operations at the query level rather than the request level, which can result in application-level inconsistencies.

4. **Program-level dependency with field-level tainting granularity (program-field):** This policy is similar to 2), the only difference being that the taint information is stored in the database at a field granularity. This policy allows us to take advantage of blind-writes, in which the value of a field is updated without reading its value. If the updated value is untainted, the taint value of the field can be reset even if it was originally set, which helps to reduce false positives. This policy also helps in generating finer-grained whitelisting policies.

5. **Database-level dependency with field-level tainting granularity (database-field):** This policy is similar to 3), the only difference being that the taints are stored in the database at a field-level granularity.

3.4.7 Whitelisting

Our analysis tools can also incorporate administrator knowledge about the application and help her determine the effects of data corruption. An administrator can whitelist tables, columns, rows or even fields, to stop taint propagation at the database tier. There are two possible ways to generate these whitelists. First, an administrator with sufficient

knowledge of the application may be able to analyze the causes of false dependencies generated by the dependency policies and generate the whitelist appropriately. When an administrator starts the replay-based analysis, our replay trace files collect sufficient information about taint propagation, at the database and the application-logic layers. This information is very useful for generating whitelists. We have used this approach to generate the whitelists used in our evaluation in Chapter 5.

Second, our system can analyze the monitoring logs to determine database entities (i.e., tables, columns etc.) that are heavily shared across all requests to generate a list of data items suitable for whitelisting.

These heavily shared data items can also be generated based on different *types* of requests. The *type* of a request can be defined as an application-level operation. For example, adding a comment, editing a post or updating permissions of a user can be considered three different types of requests. If a column is heavily accessed by one type of request, but not the others, then an administrator may choose to avoid propagating taint when the first type of request accesses this column.

There are several ways to categorize requests by type. First, one can identify different page names that the application is executing to handle a certain type of request. For example, one application may be using a script file to add a comment (i.e., `add-comment.php`) and a different file to update a comment (i.e., `update-comment.php`). Second, one can classify requests by looking at the queries that have been executed, their ordering, and the database entities they have accessed. For example, an addition of a comment may access the comments table via an INSERT operation, while an update uses an UPDATE operation.

With this request classification, an administrator may choose to avoid propagating taint based on certain sequences of queries, namely requests. In our comment example, requests that update a comment affecting the comments table may be whitelisted, without necessarily whitelisting the requests that insert a comment.

Although whitelisting can potentially miss certain dependencies and lead to incorrect recovery actions, we believe that whitelisting at a fine granularity, such as at the field level, as can be implemented with our field-level tainting, will reduce this problem.

3.5 Recovery

The recovery component provides administrative tools for performing recovery. These tools provide information about requests and sessions, such as their timeline, requests associated with a session or user, requests or sessions that affected specific database tables or generated a corrupted web page, helping the user identify and investigate specific requests or sessions that are the root cause of the problem. Also, the root cause can be determined by undoing each request, either one-by-one or by using a binary-search [50], until the administrator determines which request caused the corruption.

After the root cause is detected, the administrator provides these initial requests or sessions to the analysis component, which then generates a list of tainted requests via replay. Then the recovery component uses the queries that belong to these requests to generate the compensating transactions to revert their effects. The end result is that only the effects of the initial requests are reverted. Next, we describe this selective recovery process.

3.6 Selective Recovery

The database stores an undo log, but it does not provide a method for undoing committed transactions because it guarantees that committed transactions are stored durably. Our recovery process uses the request and transaction-level mappings and the undo log to generate compensating transactions for the tainted transactions. Note that many web applications do not use transactions. In that case, we treat each query as a separate transaction.

The compensating transactions for the tainted transactions are applied in reverse serialization order on the current state of the database. While the compensation process is relatively simple, it can cause a conflict when a tainted query deletes a row because taint analysis does not create a dependency for deleted rows. Specifically, if the tainted query had not executed, then a future query could have read the deleted row. While it is possible to taint the results of such a query during replay, currently we have chosen to ignore such deleted row dependencies for simplicity. Instead, when a compensating transaction generates a conflict, the conflicting transaction is aborted, and the administrator is provided with a list of all such aborted transactions. A conflict occurs only when the compensating transaction attempts to reinsert a deleted row, but an untainted transaction had inserted a row after it was deleted with the same primary key. In this case the INSERT fails during compensation. Compensating transactions are described in more detail in Section 4.3.

Selective recovery is first performed in the same sandbox environment where the analysis was performed. If the recovery actions solve the problem, they are applied to the original application state that was stored before the analysis started. If a problem occurs during the selective recovery process and the recovery action (i.e., the compensating transactions for a request) does not remove the corruption, the administrator has two ways to solve this issue. First, she can discard the sandbox environment and start the analysis again, this time selecting different recovery actions. Second, she can revert the effects of the compensating transactions (i.e., 'undoing the undo') because the database undo log will have the previous values. After that, she can select different recovery actions to be performed to fix the problem.

Conceptually, our selective recovery method should yield the same results as redo recovery that starts with a database snapshot taken just before the earliest initial tainted request and replays all the untainted requests. We do not use redo recovery for two reasons. First, we assume that the damage caused by the tainted requests is small

compared to the work done by the untainted requests, and hence the compensation process based on undoing the effects of the tainted requests will be more efficient than replaying all the untainted requests. Second, redo recovery will not work correctly in the presence of conflicts. For example, suppose a tainted request that deletes a row is not replayed. A request that originally did not read this row, may read it during replay, causing non-deterministic and potentially failed replay. Resolving these issues requires a great deal of application-specific knowledge [26]. Our compensation based approach does not replay requests and thus does not suffer from this problem.

Chapter 4

Implementation

We have implemented a prototype of our recovery system for the PHP scripting engine and the MySQL database. This section describes the main aspects of our implementation, consisting of the monitoring, analysis and recovery components. Table 4.1 shows the number of lines of code that we have changed to implement our recovery system's functionality in existing software code. Note the majority of the code lies in the recovery component, and our changes to the PHP scripting engine and the MySQL database are relatively small. Thus, we believe that it should be be relatively easy to port our system to other languages and databases.

4.1 Monitor

The monitor is implemented by making a small number of modifications to the MySQL database and instrumenting the PHP engine with Xdebug [23], a popular PHP debugging tool. PHP is widely employed by web applications, and the monitor can be used by all web applications that use PHP. The PHP module is used with Apache in pre-fork mode, so that each request is handled by a separate process, and the database provides the primary means for sharing information between these processes.

We made three main modifications to the MySQL database. These modifications are

Component	Existing Software	Changed Lines of Code
Database Monitor	MySQL	287
Application-logic Monitor	PHP interpreter	219
Application-logic Analysis	PHP interpreter with taint support	519
Query Rewriter	JSQLParser	1850
Recovery Component	-	4757

Table 4.1: Modifications to existing software. The numbers indicate physical lines of code that were added, modified or deleted. All numbers are approximate and may contain comments.

necessary to capture the necessary information for generating the compensating transactions during the recovery process. First, we modified the database to generate the transaction-level mapping that maps transaction ID's to row-level undo information. This undo information is already generated and stored in the database undo log by the InnoDB transactional storage backend of MySQL. Second, we modified the undo log purge operation so that it purges the log only after a user-configurable time. Third, we modified the MySQL protocol, so that each query returns its unique transaction ID to the MySQL client used by the PHP interpreter. This unique identifier is important for our recovery system because it is used by the PHP monitor to create the request-level mapping as described next.

The PHP instrumentation consists of generating the request-level mapping that maps a request ID associated with a request to the queries and their transaction ID's executed by the request. This mapping also stores the parameters of the request (e.g., the GET and POST method parameters) for replaying the requests. In addition, application-specific PHP code derives session and user information from parameters or cookie values available in requests and logs them with each request. This is the only application-specific code in our system, and it is used for session or user-level dependencies in our analysis. This code is implemented in the PHP monitor, without needing any modifications to the

Original Query	Rewritten Query
INSERT INTO table1 (id, field1, field2) VALUES (2, 5, 10)	INSERT INTO table1_extended (id, field1, field2, taint) VALUES (2, 5, 10, 1)
UPDATE table1 SET field1 = 5 WHERE id = 2	UPDATE table1_extended SET field1 = 5, taint = 1 WHERE id = 2
DELETE FROM table1 WHERE id = 2	DELETE FROM table1_extended WHERE id = 2
SELECT field1, field2 FROM table1 WHERE id = 2	SELECT taint FROM table1_extended WHERE id = 2

Table 4.2: Query rewriting examples with row-level tainting. All write queries are assumed to have a tainted value.

applications themselves.

4.2 Analysis

The analysis component initializes its state by reading the metadata associated with the database tables used by the application, including the names of the fields, primary keys, etc. Then, given an initial set of requests, it uses the query information available in the request-level mapping to generate the coarse-grained dependency graph at the request-level. The dependency graph is used to generate a replay script for all requests in the dependency graph. This script uses the request parameters stored in the request-level mapping.

Original Query	Rewritten Query
INSERT INTO table1 (id, field1, field2) VALUES (2, 5, 10)	INSERT INTO table1_extended (id, field1, field2, id_taint, field1_taint, field2_taint) VALUES (2, 5, 10, 1, 1, 1)
UPDATE table1 SET field1 = 5 WHERE id = 2	UPDATE table1_extended SET field1 = 5, field1_taint = 1 WHERE id = 2
DELETE FROM table1 WHERE id = 2	DELETE FROM table1_extended WHERE id = 2
SELECT field1, field2 FROM table1 WHERE id = 2	SELECT field1_taint, field2_taint FROM table1_extended WHERE id = 2

Table 4.3: Query rewriting examples with field-level tainting. All write queries are assumed to have a tainted value.

4.2.1 Query Rewriting

Conceptually, the taint information of the rows or fields can be stored in the database. This approach requires the interface between MySQL and the PHP interpreter to be enhanced, so that MySQL knows when to set the taint information during an INSERT, UPDATE or DELETE operation. Also, for SELECT operations, the taint information needs to be returned. However, this approach requires drastic changes to MySQL code to support all SQL functionality. For example, the more complex JOIN, GROUP BY, ORDER BY and DISTINCT SQL operations require significant database modifications to support tainting. Also, unlike row-level tainting, where an unused bit in the row structure within the MySQL database can be utilized, field-level tainting would have required incompatible changes to the database format.

We have implemented tainting using an alternative query rewriting approach, because it does not require any modifications to the database code for the tainting functionality. Our query rewriter employs a slightly modified version of JSQLParser [14]. We store the taint information by modifying the database tables used by the web application to store a per-row or per-field taint bit during replay. The PHP interpreter calls the query rewriter to access or update the taint bits for each query. For example, for field-level tainting, when an UPDATE query with a tainted value is issued, the rewriter adds a new field (i.e., taint field) to the SET clause to set the taint values for each of the fields. Similarly, a SELECT operation returns taint information of the rows in the result set to the PHP interpreter. The query rewriter retrieves the taint information for the rows and passes it to the PHP interpreter. The PHP interpreter issues the original SELECT query and associates the result with the taint information returned by the query rewriter. Compared to modifying the database, our rewriter implementation provides much greater flexibility for implementing different dependency policies. Tables 4.2 and 4.3 show other query rewriting examples with row-level or field-level tainting, respectively.

4.2.2 Handling of Blind-Writes

Our current implementation conservatively converts blind writes to regular writes and retains taints for row-level tainting. However, our field-level tainting solution resets taints for blind writes if the value being written is not tainted.

4.2.3 Propagation of Taint

On the PHP side, we use a PHP interpreter that supports tainting [46] during replay. We modified the interpreter to initialize taint propagation by tainting queries issued by the initial set of requests, and taint variables that are assigned via reading tainted database rows or fields. The taint propagation mechanism taints a query when it is executed with a tainted variable in its argument. When a query is marked tainted, it is added to a

tainted query log that is used by the recovery component. We also added some taint propagation support for conversions from one type to another (i.e., type casting) and some string operations (i.e., formatted strings).

4.2.4 Whitelisting

Currently, we manually inspect our replay trace files to determine which tables spread the taint. Our whitelisting granularity is at the table level and the administrator can specify which tables should not propagate the taint information at the database. We have also implemented a statistical analysis tool to determine heavily shared database items, such as tables and columns. Our implementation was able to determine most of the tables that we manually extracted by examining the replay trace logs. We plan to extend our query rewriter to accept {table, column} pairs for finer-grained whitelisting. Such whitelisting is enabled by field-level tainting in the database.

We have also implemented a request profiling that identifies the type of a request based on the queries executed, their order and types (i.e., INSERT, UPDATE, DELETE or SELECT), and the database items (i.e., tables and columns) they accessed. We confirmed that the types of requests that were identified were indeed different types of application operations. We plan to incorporate request profiles into our whitelisting scheme, so that the administrator can use types of requests along with the heavily shared database items to determine when to propagate the taint.

4.3 Recovery

The recovery component provides administrative tools for performing recovery and performs recovery by using compensating transactions that selectively revert the effects of tainted requests, as described in Section 3.5. Next, we describe our method for generating compensating transactions.

The compensating transaction for a committed transaction consists of write operations. For each update operation in the committed transaction, an operation that writes the previous value of the updated row(s) is appended in reverse order to the program of the compensating transaction [24]. The previous values are obtained from the undo log. For example, row-level write operations include INSERT, DELETE and UPDATE. An INSERT operation can be reverted by simply issuing a DELETE operation for that row, since there is no previous value for the row. Similarly, a DELETE can be compensated by issuing an INSERT operation with the previous row value. An UPDATE is compensated by another UPDATE operation that restores the previous row value.

The monitor collects sufficient information needed for each compensation. For example, the compensation for an INSERT operation requires the primary key of the row so that it can be deleted, while DELETE and UPDATE operations need the location of the undo information in the undo log so that the compensating queries can restore the contents of the row.

The recovery process can also be used to create a database snapshot at any time T. In this case, compensating transactions are issued for all transactions that occurred after time T. If a database provides support for fast, say periodic, snapshots, the first snapshot taken after time T can be used to generate the snapshot at time T.

4.4 Limitations

Our system has some limitations that stem from an incomplete implementation. For example, our PHP monitor does not record all non-deterministic functions, such as `time()` and `rand()`. This may cause verification of nonces for certain requests and session expiry checks to fail, resulting in a different control flow in the application execution, causing our replay trace of executed queries to diverge from the actual trace. Currently, we have manually disabled these checks during replay. In a complete implementation, the

recorded values would be used, allowing deterministic replay.

Another limitation of our system is that the JSQL parser does not handle certain MySQL specific operations. Using a full MySQL parser would avoid this limitation. Currently, we have manually generated helper queries for the MySQL extensions for our experiments.

Also, our recovery system does not handle changes made to the file system. For example, a request may be updating the dimension information about an image file in the database and resizing the image to the new dimensions. Our recovery system can revert the updates to the database, but will not be able to resize the image back to its old dimensions. In that case, our work is complementary to file system recovery systems, such as Taser [32], where older versions of files can be recovered selectively.

Chapter 5

Evaluation

In this chapter, we evaluate the accuracy and the overheads of our recovery system. We evaluate our dependency policies in terms of how accurately they determine and help with recovery from the data corruption caused by real bugs and vulnerabilities found in popular web applications. Then, we measure the performance and space overheads of monitoring at the application-logic and the database tiers.

5.1 Experimental Setup

We use the MySQL database for our experiments. The default storage engine in MySQL is called MyISAM, and it is non-transactional. Although some web applications explicitly specify the storage engine during the installation process, others do not do so. These applications do not make any assumptions about the underlying storage engine in the database, which allows us to use the transactional engine in MySQL, called InnoDB, as the default storage engine and utilize its undo log for recovery. For example, during the installation of Gallery2, the administrator specifies the database and the tables are created using InnoDB. On the other hand, Wordpress does not specify any information about the storage engine during the installation process. Although Wordpress does not use any transactions, we can run it with InnoDB to utilize InnoDB's undo log to provide

recovery. In this case, all queries are transformed implicitly into single query transactions via InnoDB's autocommit property.

5.2 Recovery Accuracy

We evaluate the accuracy of our dependency policies by triggering five real bugs in popular web applications, including Wordpress, Drupal and Gallery2. In real life, some of these bugs might manifest themselves so that the administrator recognizes that there was data corruption right after the corrupting request was issued and no other requests were served afterwards. In these cases, a single undo of that request would be sufficient to recover from the corruption. Our evaluation focuses on how various dependency policies are performing on recognizing the effects of data corruption, and therefore we use scenarios where the administrator does not recognize the corruption right away and continues to use the application, possibly generating dependencies.

We describe these bugs, failure scenarios and the correct recovery actions and report how our recovery system works under these real-life scenarios. We assume that the administrator already identified the root cause of the data corruption as described in Section 3.5 and thus, the initially tainted request in our analysis is known. We focus our evaluation on how different dependency policies perform and identify the dependencies that might have been created after the initial corruption.

5.2.1 Correct Recovery Actions

For our evaluation, we define the *correct recovery actions* to be the actions that will remove data corruption and its effects, bring the application into a consistent state and minimize the amount of data that is lost. In order to measure the effectiveness of each policy, we define three metrics. Our first metric is application-level inconsistencies that may be introduced after the recovery. An *inconsistency* in the application is created when

the recovery actions will create an inconsistent state in the application which will cause problems in application functionality. It is important that the recovery actions leave the application in a consistent state. This can be achieved by employing request-level recovery actions because web applications generally expect that the requests execute atomically. Database-level dependency policies operate at query granularity, and the corresponding recovery actions may revert only certain queries in a request. This will contradict with the assumption about requests executing atomically and thus potentially create problems in the application. Therefore, one needs to consider these inconsistencies as an important factor in deciding which dependency policy to use for the analysis.

Our second metric is the number of false positives. A *false positive* is a request or query that is marked as dependent on the data corruption during the analysis, even though it is not truly dependent. It is important to minimize the number of false positives because false positives during recovery will cause loss of data that was unrelated to the corruption. Our third metric is the number of false negatives. A *false negative* is a request or query that is dependent on the data corruption, but is not marked as dependent. False negatives will cause corruption to linger in the application even after recovery, possibly causing more problems in the future.

In a recovery system, it is difficult to know which of these last two metrics is more important. Some bugs do not have any dependencies and thus the correct recovery action is to revert the initial request that corrupted data. On the other hand, some bugs generate obscure dependencies. We show that these bugs can be complex and are very specific to the application. As a result, the administrator needs to intervene with the recovery process. We provide a discussion of alternative recovery actions another administrator could have preferred at the end of each analysis. Our goal is to help the administrator identify the dependencies and choose the correct recovery actions rapidly.

We believe that our dependency analysis approach is necessary and beneficial. First of all, it is hard to know beforehand whether there were any dependencies generated after the

corruption. These bugs can be complex and application-specific, so making an informed choice on how to perform the recovery is important. We provide the administrator with the necessary tools and detailed results of various dependency policies, so that she can decide on the correct recovery actions quickly. Second, our evaluation may be underestimating the benefits of the dependency analysis as the bugs we picked do not generate many dependencies. However, we still believe that even with these bugs, we are able to show the strengths and drawbacks of different dependency policies.

We summarize the results of various dependency analysis policies in Table 5.1 and Table 5.2. In Table 5.1, the second column shows the total number of requests we had to replay for the dependency analysis. The third column indicates the number of requests that the administrator needs to undo to correctly recover from the data loss or corruption. The next column shows the dependency policy that was used; 'none' indicating that no dependency information is taken into account for undo. The last two columns present the accuracy of the policies in terms of false positives and negatives. These numbers are in terms of requests. In Table 5.2, we present the results of database-level dependency policies. Since database-level policies only create dependencies across queries, all the numbers shown are in terms of queries. The last column shows the inconsistencies that are encountered after undoing these queries. For each of these cases, we first give an overview of the application and provide background information for the corresponding bug, which helps explain the results of each dependency policy.

5.2.2 Wordpress Functionality Bug: Renamed Link Categories

Wordpress is a popular blogging application that allows users to create content (e.g., posts, pages, links) and associate them with tags and categories. Content can be grouped

Case	Total Number of Requests	Requests to Undo	Dependency Policy	False Positives	False Negatives
Wordpress - link category rename	109	1	none	0	0
			request-row	60	0
			program-row	8	0
			program-field	6	0
Drupal - lost comments	117	1	none	0	0
			request-row	116/102	0
			program-row	100/93	0
			program-field	95/0	0
Drupal - lost voting information	118	7	none	0	6
			request-row	111/100	0
			program-row	95/89	0
			program-field	89/0	0
Gallery2 - permission	91	1	none	0	0
			request-row	90/13	0
			program-row	88/11	0
			program-field	82/10	0
Gallery2 - resizing images	151	1	none	0	0
			request-row	148/0	0
			program-row	139/0	0
			program-field	119/0	0

Table 5.1: Recovery accuracy for request-level and program-level dependency policies. Each replay starts with one initially tainted request. The false positives column show numbers without and with table whitelisting, respectively.

Case	Queries to Undo	Dependency Policy	False Positives	False Negatives	Inconsistencies after Undo
Wordpress - link category rename	23	database-row	0	15	The count value does not match to the actual number of links.
		database-field	0	21	
Drupal - lost comments	24	database-row	116	0	-
		database-field	0	0	
Drupal - lost voting information	38	database-row	86	16	The poll_votes table has duplicate entries.
		database-field	0	18	
Gallery2 - permission	9	database-row	97	0	The global sequence id has an old value breaking future inserts requiring a new id.
		database-field	9	0	
Gallery2 - resizing images	17	database-row	110	0	
		database-field	20	0	

Table 5.2: Recovery accuracy of database-level dependency policies. Each replay starts with one initially tainted request. All numbers indicate queries.

into categories, which is used to summarize and present it in a more organized way.

Scenario: An administrator already has a number of links posted and associated with a certain category, `category_one`. The Wordpress interface requires the administrator to click on the name of the category to edit it. The Wordpress bug [21] allows the administrator to rename the category name to an empty string. When she accidentally renames `category_one` into an empty string, she cannot edit it back and has to use the application with the empty category name. However, the administrator can still associate links with this category, because the application interface allows the administrator to select its checkbox. In our scenario, she continues to add new links, associated with the renamed category, as well as with other categories (e.g., {`category_two`}, {`old_category_one`,

category_two}, {category_two, category_three}). She also continues to create new content and modify other settings.

Correct recovery actions: Undo the rename operation.

Background: Wordpress maintains link entities, terms (i.e., tags and categories) and their types (i.e., whether it belongs to posts, pages or links) in separate tables (i.e., wp_links, wp_terms, wp_term_taxonomy). There is also another table storing the relationships between the actual content and the terms (i.e., wp_term_relationships). The count column in the wp_term_taxonomy table is associated with the number of links belonging to a certain category as determined by querying the wp_term_relationships table. This is used for easy access when generating a page displaying how many links belong to a category.

Results: The request-row policy marks many requests (60) as falsely dependent. The false positives in this case arise from an actively shared table (e.g., wp_options). When a request updating this table reads tainted information from the above mentioned tables, it taints a row in the wp_options table. As the request-row policy does not consider whether this information has been used when updating the database, it conservatively taints the request, thus, spreading the taint to the wp_options table. Since all requests query this table, they get tainted creating many false positives.

The program-row policy reduces the number of false positives from 60 to 8, because data flow at the program-level prevents taint spreading to the wp_options table. However, there are still eight false positives because of the row-level tainting granularity. When the administrator adds links with other categories than the renamed one (i.e., {category_two, category_three}), these operations get tainted, because each of these categories were previously used with the renamed category, such as {old_category_one, category_two} and {old_category_one, category_three}. However, the addition of the links associated with {category_two, category_three} are not related to the renamed category and therefore, these requests are marked as falsely dependent.

The finer-grained program-field policy has six false positives and no false negatives. Field-level tainting improves accuracy, because the addition of links associated with {`category_two, category_three`} are recognized as independent of `old_category_one`. The six false positives are caused because these requests added new links associated with `old_category_one` and updated one link that belonged to `old_category_one`.

Associating a new link with a category in the Wordpress application consists of three steps. First, the relation between the link and the category is inserted into the `wp_term_relationships` table. Second, this table is queried for the number of links associated with the category. Third, this number is used to update the `count` field of the category in the `wp_term_taxonomy` table. The database-row policy only marks the third step (i.e., the update) as tainted, because it reads a tainted row (i.e., the category's row) that was tainted previously when the administrator added another link associated with the renamed category. However, the insert operation in the first step was not marked as tainted, because it did not read any tainted rows at the database. Therefore, reverting only the update operation will cause an inconsistency in the application, because the actual number of links belonging to a category in the `wp_term_relationships` table will not match the `count` value in `wp_term_taxonomy` table. On the other hand, the database-field policy misses all other related operations, because the `count` value is blindly overwritten, causing its taint value to be reset. Thus, related operations, such as creating a relationship with that category, are missed. Although exploiting blind-writes via field-level tainting to break dependencies can be desirable, employing a database-level dependency policy can have false negatives.

Discussion: One can argue that the addition of the links associated with `old_category_one` and updating an already existing link that belonged to this category should be considered as dependent on the initial corrupting request, because there is an explicit data dependency between the requests (i.e., `old_category_one`'s id is used to create the relationship). However, this recovery action based on this dependency would lose the

new links and the update. In this case, choosing the correct recovery action is non-trivial because this problem is application specific. Instead, we provide detailed results for the different policies and thus help the administrator make an informed decision about the correct recovery action. We did not have to use whitelisting for this case, because the number of tainted requests was small, allowing us to manually determine whether a request is really dependent or not.

5.2.3 Drupal Data Loss Bug: Lost Comments

Drupal allows fine-grained access control for users via roles. A role has capabilities, consisting of combinations of individual actions. Some of these actions are posting new content, moderating comments and adding new users.

Scenario: When two users with the capability of moderating comments are visiting the same page, the application shows a list of comments. When the first user selects and deletes `comment1`, the application works as expected: `comment1` is deleted and the view is updated with the new list of comments. However, the list of comments is not refreshed for the other user moderating the comments. If the second user selects and tries to delete `comment1`, a Drupal bug [6] is triggered so that the application ends up deleting all comments from the database. In our scenario, the first user generates new content after deleting `comment1`, such as adding new stories and polls, and makes some configuration changes and activates a new module. Meanwhile, other users comment on both the old and new content.

Correct recovery actions: Recover the comments that were deleted because of the bug. These will be in the persistent storage (i.e., the database) once the recovery is complete and therefore, the temporary views of moderators are not considered important.

Background: Drupal maintains the session information in the `sessions` table. The session information, which is retrieved at the beginning of each request, contains the associated user's id, `uid`. This identifier is later used to make the appropriate permission

checks. At the end of each request, the `sessions` table is updated with a timestamp and other session related information. The content (e.g., posts, pages, polls) is stored in the `node` table with a unique identifier, `nid`, and user generated content (e.g., comments) is stored in the `comments` table. There is a `users` table for user information, which also has an `access` timetamp updated at the end of each request.

Results: Once the session information is tainted after the initial request, the request-row policy marks every subsequent request that reads the tainted session information as tainted. Note that the request-row policy does not consider whether this tainted information is used to update the database to mark a request as tainted. This taint spreads to other users' requests via the `node` table when the front page of the application is accessed, resulting in many false positives.

The program-row policy still has many false positives. The initial request taints the session information, which also includes the associated user id, `uid`. This tainted user id is used whenever new content is inserted into the `node` table, causing the new nid to be dependent. This node id is then used if another user uses it when generating a content, such as adding a new comment, which again causes the session information for that user to be tainted, thus increasing the number of false positives. For this policy, users logging in, viewing the content, reading the tainted information and logging out are not marked as tainted, because these tainted values are not used to update the database. For the requests that belong to a different session for the same user, the taint is spread, because the initial request has tainted the user's `access` timestamp. With row-level tainting, this causes that any subsequent session for that user to be marked as tainted.

To our surprise, the finer-grained program-field policy did not reduce the number of false positives much, even though the session information was being updated with a blind-write at the end of each request, where the taint information could have been reset. Our investigation revealed that this update operation was using a tainted value, namely the user id, that got tainted when the initial request updated the session information. This

taint then spreads resulting in false positives. The number of false positives is lower than the program-row policy, because program-field policy takes advantage of blind-writes to contain the taints in a specific session. For example, although the access timestamp for a user may get tainted because of a tainted request, the blind-write with a clean value in another request resets its taint, thus reducing the number of false positives.

The database-row policy marks all updates to the sessions table for that user as tainted, since the session information was tainted in the initial request. However, it is contained for this user, because each other user has her own separate session information. Additionaly, queries updating the `users` table in subsequent sessions of the same user get tainted, because of the update to the `access` time. On the other hand, with field-level tainting, the update resets the taint, because it is a blind-write with a clean value.

After we ran our analysis with these different dependency policies, we decided to whitelist the `sessions`, `history` and `watchdog` tables, because our replay log files showed that they were spreading the taint. These log files contain sufficient information to identify why a request was marked as tainted during replay. For example, the administrator would know whether a query was executing with a tainted value or read a tainted row at the database. Using these logs, we determined that the tainted user id was used in updating the session information, described above.

We ran our analysis again and the request-row and program-row policies with whitelisting still produced many false positives. We realized that these false positives were generated by the `users` table. Our initial replay logs did not indicate that this table was a source for false positives. On the other hand, the program-field policy did not produce any false positives with the same tables being whitelisted.

Discussion: Although whitelisting tables is a coarse-grained method of reducing false positives, it still shows potential. One can expect an administrator with sufficient knowledge of the application to determine these whitelists. Furthermore, our replay log files help the administrators to determine how the taint is spreading. Although we determined

these tables manually by inspecting the log files, one can generate these statistically via investigating traces and determining which tables are being shared extensively (i.e., being written to and read from by many requests/queries). Also, one can create these whitelists at the row or at the column granularity, where administrator can specify a set of rows or columns to be ignored while propagating the taint. Finer-grained tainting at the database (i.e., row vs. field-level) will allow finer-grained whitelists. For example, one could have whitelisted the `uid` column in the `sessions` table. Finer-grained tainting is also more resilient to incomplete whitelists. In our case, we did not realize that the `users` table was a source of taint spreading, but the program-field policy with an incomplete whitelist did not generate any false positives.

5.2.4 Drupal Data Loss Bug: Lost Voting Information

Drupal allows an administrator to create a poll with multiple choice options via the poll module. One can set access controls on who can vote, such as only registered users or any other role the administrator has created and assigned voting capability. Also, a user can only vote once and the application asserts this by keeping track of which users have voted.

Scenario: An administrator creates a poll for the registered users. After some users have voted, the administrator modifies the poll contents (e.g., to fix a typo, clarify the question, etc.). The bug [7] causes the information about which users have voted to be lost, allowing the users to vote again. However, the vote counts are still preserved creating an inconsistency in the application. For example, if there are only 5 registered users, the sum of the votes should not be greater than 5.

Correct recovery actions: Determine the 'second' votes and restore information about who has voted.

Background: The poll information is saved in three different tables. The content is saved in the `poll_choices` table, such as the content of the choices, the number of votes,

etc. The `poll_votes` table keeps track of who has voted and uses this information to prevent users from voting twice. The `poll` table keeps the association between the `node` table and the poll-type content. For the rest of relevant information, see Section 5.2.3.

Results: The request-row policy marked all requests as tainted, because of the shared session information, similar to the case described in Section 5.2.3. The application puts the updated poll to the front page of the site. The taint is propagated between different users, because every session starts requesting the front page.

The program-row policy would taint requests that are reading and using the poll information during the voting process. When different users vote on the same poll, their sessions also get tainted, resulting in many false positives. The program-field policy also has many the false positives as the tainted user id retrieved from the session information is used to update the session information at the end of the request.

The database-row policy marked queries in many distinct requests to be tainted. Similar to the previous case, all false positives were related to the `sessions` and `users` table. On the other hand, the database-field policy only marked the queries that updated the number of votes in the `poll_choices` table. However, reverting the effects of only these queries would create an inconsistency in the application, because the voting request, not only increments the number of votes, but also inserts the information about who has voted into the `poll_votes` table. A, this table would have duplicate entries.

Discussion: Again, we chose the same tables to be whitelisted, namely `sessions`, `history` and `watchdog`. The request-row policy still produced many false positives via the `users` table, while updating the access timestamp field. On the other hand, the field-level policy resets the taint for the `access` timestamp field, because it is updated with a blind-write. Therefore, it does not produce any false positives.

As we can see from this case and the case described in the previous section, the same dependency policies produce different results in terms of false positives, even though the scenarios involve the same application. This shows that the nature of the bug the

administrator is trying to recover from plays an important role on determining what kind of dependencies really exist and thus effects the correct recovery actions. We can help the administrator in this process, by providing detailed results and analysis logs for each policy and how they generate the dependencies.

5.2.5 Gallery2 Functionality Bug: Removing Permissions Breaks the Application

Gallery2 has a fine-grained access control mechanism. An administrator can give various capabilities to specific users and/or groups. Examples of these capabilities include viewing an item, commenting, moderation of comments and rating. These capabilities apply to specific pictures or whole albums.

Scenario: An administrator wishes to make several changes to the photo gallery settings. She temporarily removes other users' permissions to view the entire gallery, makes her changes, such as creating sub-albums under the main gallery, adding users and groups and then logs out. The Gallery2 bug [12] causes the application to show an error message after the administrator logs out and to stop working completely, making the web interface to the application no longer available.

Correct recovery actions: Restore the permission to view the gallery. This administrator prefers to be less conservative and therefore, does not consider the addition of the sub-albums relevant to the corruption.

Background: Gallery2 uses a global sequence id for every item (e.g., picture, album, users, groups) that is inserted into the database making this id the primary key for these items in their respective tables. It keeps track of the last used value with the `g2_SequenceId` table. There is also a global `g2_Entity` table that keeps track of each item and its associated information. For entities, such as sub-albums and comments belonging to an album, a `g2_ChildEntity` table stores the relationships. The table `g2_SessionMap` tracks open sessions, associating each session with a specific user id, `g_id`, which is then

used throughout the request.

Results: The request-row policy marks almost every request as tainted. The reason is that the initial request taints the session information. The taint is spread to other requests, because the session information is read at the beginning of each request to fill in the current user information.

The program-row and the program-field policies also have large numbers of false positives. The main reason, similar to the Drupal bugs, is that the user id retrieved from the tainted session information is propagated throughout the request and is used to update the session information at the end of the request. The user id value is tainted and, thus this update taints the request.

The database-row policy marks the update queries to the `g2_SessionMap` and `g2_SequenceId` tables as tainted. One can argue that the session information is temporary and can be ignored during the recovery process. However, this is not true for the sequence update queries, because reverting only these will create an inconsistency in the application. The reason is that every insertion of an item (e.g., album, comment, picture, etc.) will increment the sequence id. If this value is reverted back to its original state before the corruption, a new item being inserted will get an id that has already been assigned to another item. This will certainly cause the application to behave in an undesired fashion and fail, because another item uses the same id in the `g2_Entity` table.

On the other hand, the database-field policy only marks the sequence id updates, because these queries read the old value of the field during the update. The policy resets the taint for the session information, because the update is a blind-write and the value used in the update cannot be tainted, because the database-level policy does not propagate the taint throughout the program execution.

Discussion: Based on our logs, we whitelisted the `g2_SessionMap` and `g2_SequenceId` tables which significantly reduces the number of false positives. The false positives are caused by the parent-child relationship between the main gallery and the sub-albums

the administrator added. During the addition of the sub-albums, the id of the main gallery, which was tainted by the initial request, is used to insert new entries to the `g2_ChildEntity` table, resulting in false positives. These false positives could be prevented via whitelisting by the administrator.

We believe that this situation is application-specific in terms of choosing the correct recovery actions. Another administrator might think that these requests are really dependent because viewing the sub-albums is also prevented when the parent album is not viewable. Therefore, the addition of the sub-albums are dependent on the first request that removed the permissions from the parent (i.e., main) album. However, the number of false positives is small and manageable. Furthermore, our replay log files give enough information about how taint spreads to the administrator, so that she can decide for the correct recovery actions.

We should note that the database-row and database-field policies not only had false positives, but undoing these false positives would create the inconsistencies described in Table 5.2.

5.2.6 Gallery2 Functionality Bug: Resizing Images Breaks Existing Links to Images

Gallery2 works as a web application where users can organize their pictures into albums and custom web pages. One useful functionality is to provide different sized versions of images. These images can be linked from other web sites, where the application would work as a storage system. Gallery2 is using image id's in the URLs of resized images. Because of a bug in the resizing procedure of the application, the resized images receive new id's, causing their their URLs to change. This breaks the existing links to old versions of resized images.

Scenario: A Gallery2 administrator has multiple albums and multiple images in her albums. She has used the links to resized versions of these images in other web sites. She

wishes to create a new resized version of her pictures and recreates the resized versions for one of her albums. However, the bug in Gallery2 resizing procedure [11, 13] causes the existing links to be broken. She continues to interact with her gallery installation regularly by adding users, groups and albums, while other users view and post comments to these. After a while, she realizes that all the external links to the images she had posted are broken.

Correct recovery actions: Revert back the newly created versions.

Background: Additional to the information in Section 5.2.5, there are two tables that are shared among all users. These are the `g2_AccessMap` and `g2_AccessSubscriber-Map` tables, where the permissions of the users are kept.

Results: The request-row policy marks almost all requests to be falsely tainted, mostly because the session information is shared by all requests. The taint spreads to other users' sessions, when they read the `g2_AccessMap` and `g2_AccessSubscriberMap` tables at the beginning of their sessions. The program-row and the program-field policies contain the taint in one session of the administrator, because the tainted information obtained via reading the shared tables (i.e., `g2_AccessMap` and `g2_AccessSubscriberMap`) is not used to update the database. However, all requests in that session is tainted because of the tainted user id.

Similar to the previous case, the database-row policy marked the queries that updated the session information and the global sequence id. The database-field policy marked updates to the global sequence id. However, we should note once again that reverting the effects of the global sequence id would create an inconsistency in the application by breaking future insertion operations.

Discussion: In this particular case, whielisting the `g2_SessionMap` and `g2_Se-quenceId` tables reduces the number of false positives drastically to 0. Although this may make the dependency analysis seem like redundant, our replay trace logs indicated that these tables should be whitelisted.

Both Gallery2 cases once again show that the nature of the bug plays an important role in the correct recovery action. Although in some of the cases that we have described the correct recovery action is to only revert the initial request, doing so without thoroughly analyzing the dependencies might generate other problems. With our different dependency policies, we can help the administrator gather the necessary information to choose the correct recovery action.

5.2.7 Remarks

We have applied the different dependency policies and to several bug and misconfiguration scenarios across several applications. We have shown that the request-level dependency policies suffer from high false positive rates. On the other hand, the database-level policies can have many false negatives. Furthermore, web applications generally expect that requests execute atomically and thus recovery should be performed at a request granularity to minimize inconsistency in the application after recovery. We have also shown that the program-field dependency policy has the best results. It performs recovery at the request granularity with the least number of false positives.

Although database-level policies can cause application-level inconsistency, they tend to have fewer false positives than the request- and program-level policies. Thus, an administrator can compare the outputs of the database- and the program-level policies to derive the correct recovery actions more accurately and rapidly. On the other hand, the request-level policies can be useful if the replay needed for the program policy cannot be performed successfully (e.g., incomplete implementation).

5.3 Performance

In this section, we measure the performance overhead and the disk space requirements of our monitoring tools. Our experiments use the TPC-W industry benchmark. We

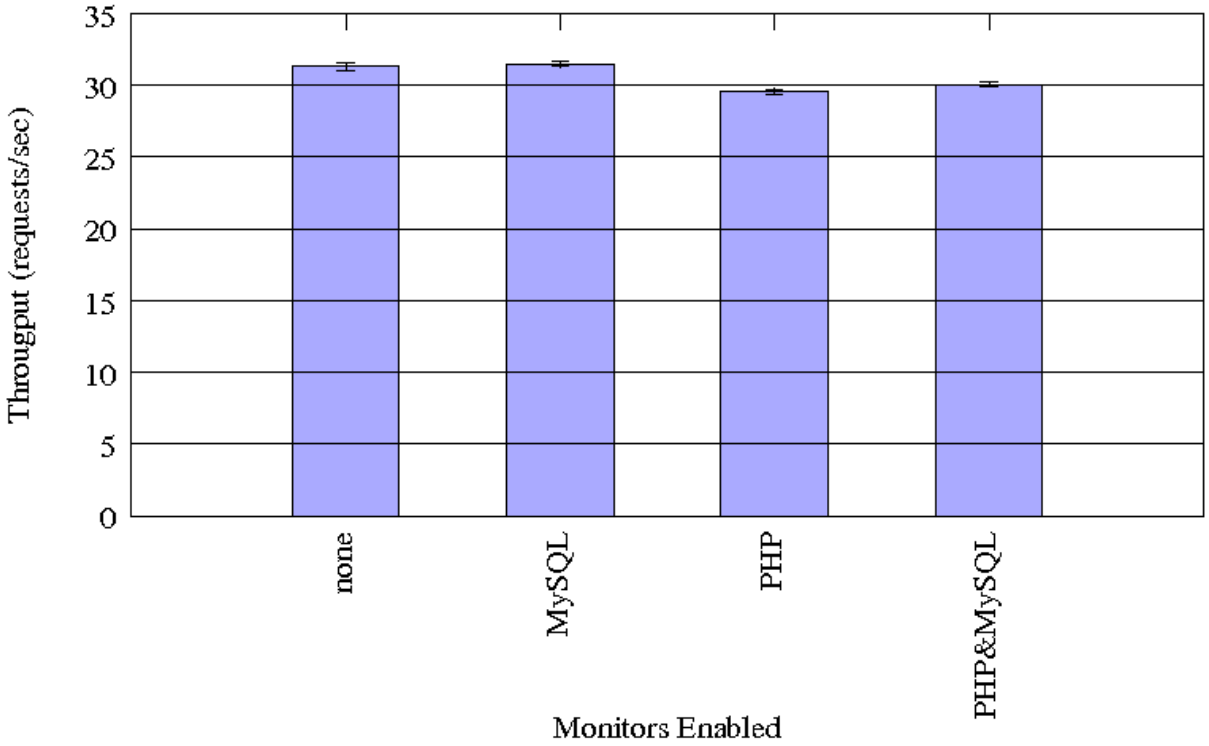

Figure 5.1: Throughput results.

measure the throughput overhead of our MySQL and PHP monitors. We also measure the logging overhead caused by our instrumentation of the MySQL and PHP interpreter. The log at the PHP interpreter tracks requests, sessions and users, while the MySQL logs track update queries and the rows that are modified. We also measure the logging overhead that results from disabling of the undo log purge operation in the database.

All tests were conducted on a server with Intel Pentium 4 2.80 GHz with dual CPU and 1.5 GB of RAM on Ubuntu Linux 8.04 with Apache server 2.2.8 running in pre-fork mode. Both CPU's were saturated using 100 emulated clients running on an Intel Pentium 4 3.0 GHz with 4 CPU's and 2 GB of RAM. The client machine and the server were connected via a 1000Mb LAN connection. All results reported are the average of at least 15 trials that run for 30 minutes.

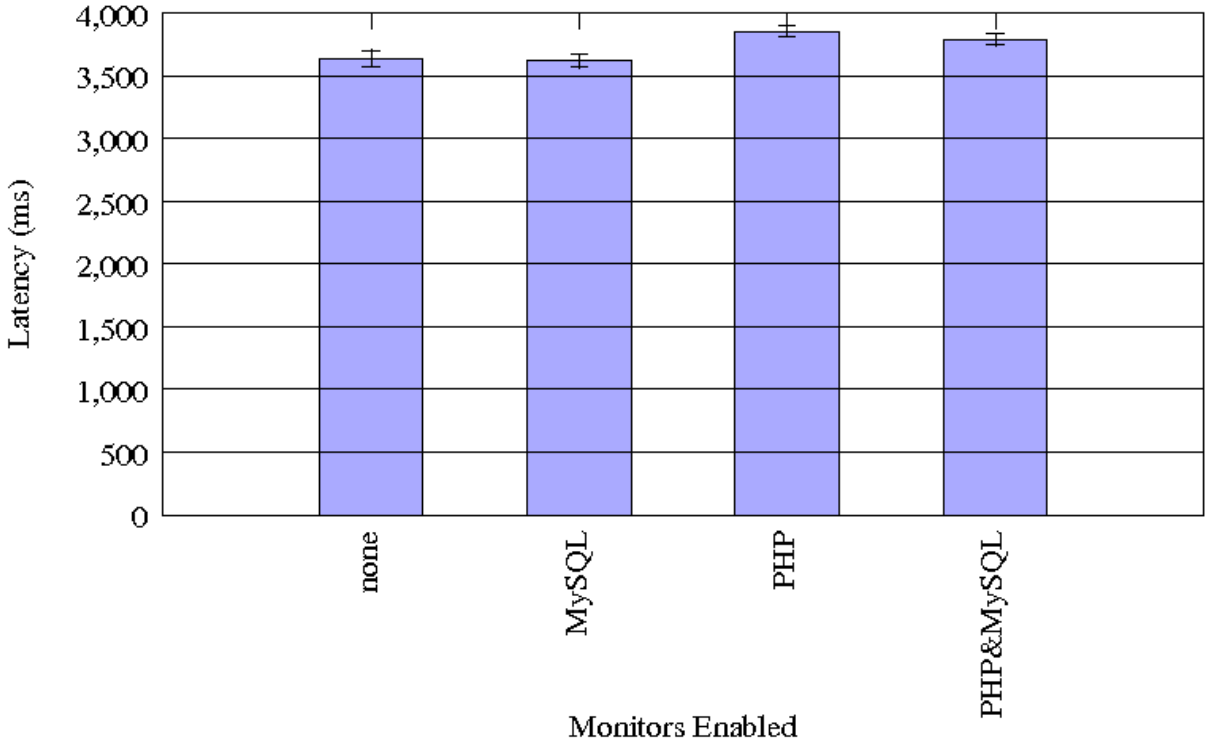

Figure 5.2: Latency results.

5.3.1 Throughput Overhead

To measure the performance overhead of each monitoring component at the application and the database tiers, we ran tests by enabling each of the monitors separately and then we enabled both of them together. The throughput and latency results can be found in Figures 5.1 and 5.2, respectively. Our results show that our instrumentation incurs a maximum of 4% overhead in throughput and latency, when both of our monitors are enabled. The largest portion of the overhead is caused by our PHP instrumentation. We believe that this instrumentation can be optimized further.

The reason that our database instrumentation improves performance slightly (compare the first and second bars and the third and the fourth bars in Figures 5.1) is that our monitor disabled the periodic purge operation that would remove the undo information of committed transactions. To verify this, we conducted another set of experiments in which we disabled the purge operation for the standard MySQL database. As expected,

Monitors Enabled	Increase in Undo Log Size (MB)	Undo Index Log Size (MB)	PHP Log Size (MB)	Number of Requests	Total Space Overhead per Request (KB)
None	0.94	0	0	56352	0
MySQL	32.0	2.67	0	56708	0.63
PHP	0.53	0	159.41	53184	3.08
PHP & MySQL	29.87	2.53	163.55	54114	3.71

Table 5.3: Disk space overhead

the throughput result was better with the purge operation disabled, with the throughput increasing by about 1%. We also enabled all monitors (i.e., MySQL & PHP), but left the purge operation enabled. In this case, there was a slight increase in throughput overhead from 4% to 5%.

5.3.2 Disk Space Overhead

Table 5.3 shows the disk space overhead of the log files generated by our monitors. The disk overhead arises from disabling the undo log purge and keeping the mapping between transactions and modified rows in the database and the PHP log. The database logs account for roughly 0.63 Kbytes per request (i.e., (32 MB + 2.67 MB)/56708), while the PHP log accounts for 3.08 bytes per request (i.e., (0.53 MB + 159.41 MB)/53184) for the TPC-W benchmark. Our PHP log uses textual data and the logging operation can be significantly optimized.

For the 30 minute experiment, our total log size is around 196 MB (i.e., 29.87 MB + 2.53 MB + 163.55 MB). This number extrapolates to 9.19 GB per day. When the PHP log file is compressed, our total log size decreases to around 48 MB (i.e., 29.87 MB + 2.53 MB + 15.12 MB) for the experiment and 2.23 GB per day. Given current disk

capacities, these logs can be saved on a 250 GB disk for about 104 days. We believe that this overhead is acceptable for providing a generic recovery system for web applications.

Chapter 6

Conclusion

A misconfiguration or a bug causing data loss or corruption in web-based applications can affect a large number of users, because these applications store data at the server side. Although useful for recovery, data backup solutions have shortcomings when it comes to diagnosing the actions and identifying the specific changes caused by them.

We have described the design of a generic recovery system for web applications that helps administrators to selectively recover corrupted data caused by a bug or a misconfiguration. Our system tracks and correlates requests across multiple tiers of the application with modest changes to existing software to help the administrator to determine correct recovery actions. Our recovery system reuses undo logs of the database to recover from data corruption. Our analysis tools assist the administrator in identifying requests that led to data corruption and provide multiple application and database dependencies to identify the effects of data corruption more effectively and rapidly. Our prototype implementation with MySQL and PHP interpreter shows that this generic recovery functionality can be obtained with little overhead and no modifications to the web applications themselves. Our evaluation showed that our system can help diagnose and recover from various corruption scenarios and real bugs.

Bibliography

[1] Amazon hit by pricing error. http://news.zdnet.co.uk/internet/0,1000000097,39226977,00.htm, last access August 23, 2009.

[2] Amazon shuts after price error. http://news.bbc.co.uk/2/hi/business/2864461.stm, last access August 23, 2009.

[3] Best Buy will not honor $9.99 big-screen TV deal. http://edition.cnn.com/2009/US/08/13/bestbuy.mistake/, last access August 23, 2009.

[4] Community plumbing. http://drupal.org/.

[5] Dell customers get snappy at pricing error. http://news.zdnet.co.uk/internet/0,1000000097,39181032,00.htm, last access August 23, 2009.

[6] Drupal Bug Report: Big bug in management comments. http://drupal.org/node/67745, last access August 23, 2009.

[7] Drupal Bug Report: Editing a poll clears all old votes. http://drupal.org/node/67895, last access August 23, 2009.

[8] Drupal Group: Remove warning modal dialogs and replace them with undo. http://groups.drupal.org/node/21913, last access August 23, 2009.

[9] Drupal Project: Deletion API for core. http://drupal.org/node/147723, last access August 23, 2009.

[10] Drupal Project: Drupal gets a trashbin, and other delete goodies! http://drupal.org/node/35422, last access August 23, 2009.

[11] Gallery2 Bug Report 1574209: Rebuilding thumbs/resized changes path to resized image. http://sourceforge.net/tracker/index.php?func=detail&aid=1574209&group_id=7130&atid=107130, last access August 23, 2009.

[12] Gallery2 Bug Report 2016834: One easy step to break G2 with album permissions. http://sourceforge.net/tracker/index.php?func=detail&aid=2016834&group_id=7130&atid=107130, last access August 23, 2009.

[13] Gallery2 Support Node 53486: Rebuilding Thumbs/Resized causes new ItemIDs to be created which breaks old links and images. http://gallery.menalto.com/node/53486, last access August 23, 2009.

[14] Jsqlparser project. http://jsqlparser.sourceforge.net/.

[15] MySQL 5.0 Reference Manual :: 13.2.4.4 FOREIGN KEY Constraints. http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html, last access August 23, 2009.

[16] Wordpress - blog tool and weblog platform. http://wordpress.org.

[17] Wordpress Codex - Managing Plugins. http://codex.wordpress.org/Managing_Plugins, last access August 23, 2009.

[18] Wordpress Codex: IRC Meetup. http://codex.wordpress.org/IRC_Meetups/2007/September/September26RawLog, last access August 23, 2009.

[19] Wordpress Ticket 4720: Users without unfiltered_html capability can post arbitrary html. http://trac.wordpress.org/ticket/4720, last access August 23, 2009.

[20] Wordpress Ticket 4748: Unprivileged users can perform some actions on pages they aren't allowed to access. `http://trac.wordpress.org/ticket/4748`, last access August 23, 2009.

[21] Wordpress Ticket 5809: Categories affect tags of the same name. `http://trac.wordpress.org/ticket/5809`, last access August 23, 2009.

[22] Wordpress Ticket 6662: Users without capability "create_users" can add new users. `http://trac.wordpress.org/ticket/6662`, last access August 23, 2009.

[23] Xdebug - Debugger and Profiler Tool for PHP. `http://www.xdebug.org`.

[24] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[25] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 259–272, 2004.

[26] Aaron B. Brown and David A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX Technical Conference*, pages 1–14, June 2003.

[27] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: Support for Controlling and Analyzing the Execution of Web-Accessible Applications. In *Middleware 2005*, 2005.

[28] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the Networked Systems Design and Implementation (NSDI)*, 2004.

[29] Tzi cker Chiueh and Dhruv Pilania. Design, implementation, and evaluation of a repairable database management system. In *Proceedings of the Annual Computer Security Applications Conference*, pages 179–188, 2004.

[30] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the Network and Distributed System Security Symposium*, February 2006.

[31] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2002.

[32] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 163–176, October 2005.

[33] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the Annual Computer Security Applications Conference*, pages 303–311, 2005.

[34] K. Salem Hector Garcia-Molina. Sagas. In *Proceedings of the ACM SIGMOD International Conference of Data*, pages 249–259, 1987.

[35] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.

[36] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

[37] Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[38] David B. Lomet. MLR: a recovery method for multi-level systems. *SIGMOD Rec.*, 21(2):185–194, 1992.

[39] Bharat Mediratta. Gallery photo album organizer. `http://gallery.menalto.com/`, 2004.

[40] Susanta Nanda, Lap-Chung Lam, and Tzi cker Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the ACM/IFIP/USENIX international conference on Middleware*, pages 1–20, 2007.

[41] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[42] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[43] A. Smirnov and T. Chiueh. A portable implementation framework for intrusion-resilient database management systems. *Proceedings of the IEEE Dependable Systems and Networks*, pages 443–452, 2004.

[44] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 165–180, 2000.

[45] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 237–250, 2007.

[46] Wietse Venema. Taint support for PHP. `ftp://ftp.porcupine.org/pub/php/index.html`, last access August 23, 2009.

[47] W. J. Lee, J. Loaiza, M. J. Stewart, W. Hu, W. H. Bridge, Jr. Flashback Database - US Patent 7181476, 2007.

[48] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 245–258, December 2004.

[49] Gerhard Weikum, Christof Hasse, Peter Broessler, and Peter Muth. Multi-level recovery. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 109–123, 1990.

[50] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[51] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, 2006.

[52] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.