Data Recovery for Web Applications

İstemi Ekin Akkuş, Ashvin Goel
University of Toronto
{ekin, ashvin}@eecg.toronto.edu

Abstract

Web-based applications store their data at the server side. This design has several benefits, but it can also cause a serious problem because a misconfiguration, bug or vulnerability leading to data loss or corruption can affect many users. While data backup solutions can help resolve some of these issues, they do not help diagnose the events that led to the corruption or the precise set of changes caused by these events.

In this paper, we describe the design of a recovery system that helps administrators recover from data corruption caused by bugs in web applications. Our system tracks application requests, helping identify requests that cause data corruption, and reuses undo logs already kept by databases to selectively recover from the effects of these requests. 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 (database, application, and client) to help identify data corruption accurately. We evaluate our system using known bugs 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 and incurs small runtime overhead.

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 [11] are generally advised to back up all data before installing any new plugins or new versions of the application [12].

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 [29] photo application that allows storing and sharing personal photos with specific users. However, Wordpress has had several vulnerabilities [15, 16, 17] 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 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 an undo functionality, available in web applications such as Google Mail. This feature allows the user to undo her last action, enabling recovery from simple misconfiguration problems or accidental clicking. While this feature is useful, it has several limitations. First, it needs careful design and significant modifications to applications, especially if the application is extensible [8, 13]. Second, it may require several, potentially complex, manual operations if the corruption is detected much after it occurred. Finally, most importantly, application-specific recovery depends on the correctness of the application. For example, Gallery2 bug number 2016834 [9] prevents all users, including the administrator, from accessing the application interface and thus the last action cannot be reverted, even if the undo feature was available. Similarly, 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 paper, we describe the design of a generic data recovery system for web applications that store their persistent data in a database tier. Our system does not rely on the web application for recovery and thus, is resilient to failures and bugs in the applications. Our system has two main goals: 1) allow web application administrators to diagnose application failures that corrupt persistent data 2) enable selective recovery of this data, without affecting the rest of the application.

A significant challenge in achieving our goals lies in identifying data corruption and its dependent effects accurately. 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 its dependent effects.

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, in which a row update or delete in a table causes other referencing tables to be updated, also cause dependencies.

Using dependencies for data recovery has been explored previously in the context of transactional databases [19, 23, 27] and file systems [24, 38]. However, these techniques are applied at a single layer, making them unsuitable for web applications. For example, transaction-level dependency analysis cannot be directly applied to web applications as they may not use transactions, so the recovery system cannot depend on their existence. Also, web applications operate at multiple tiers. Ignoring the interactions across tiers, these approaches can cause inconsistency after recovery, as shown in Section 4.2.

Our recovery system correlates dependencies across different layers, namely at the presentation, application and database tiers, thus helping diagnose data corruption more accurately. A potential drawback of our approach is that it can have false dependencies between requests that are essentially independent, leading to data loss. We explore using several dependency policies to resolve this problem. Furthermore, we show that the use of multiple policies helps the administrator determine the correct recovery actions more rapidly.

Our system uses two novel methods for dependency analysis. First, it combines application replay with offline taint analysis for deriving application-level dependencies. Tainting has generally been used online for securing applications [30, 31, 37], but we are not aware of its use for data recovery. Second, we explore the benefits of using finer-grained field-level dependencies at the database than existing approaches that use row-level dependencies [19]. These techniques help the administrator identify data corruption more accurately.

Our main contribution is a dependency-based recovery system for web applications, that is resilient to bugs and application misconfigurations. We have implemented a prototype of our system for the widely used PHP interpreter and the MySQL database, and tested our system on popular web applications including the Wordpress [11], Drupal [4] and Gallery2 [29]. We evaluate the effectiveness of our approach for various data corruption scenarios that can be triggered by known bugs and misconfigurations in these applications. Our second contribution is that we compare our results with previous approaches [19, 23, 27]. These approaches conducted performance evaluation, but did not evaluate whether their approach caused false positives/negatives during recovery. We have used real web applications to evaluate the accuracy of our dependency analysis. Also, our database-row dependency scheme was designed to mimic the previous approaches.

Next section discusses related work in the area. Section 3 describes our approach and prototype implementation. In Section 4, we evaluate our system for various data corruption scenarios in popular web applications. We conclude in Section 5.

2 Related Work

Liu et al. [27] 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-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 [19]. Similar recovery methods have been proposed in Fastrek [23] and the Flashback Database [34]. 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-write sets of queries and requests, avoiding application-level inconsistencies and tracking corruption more accurately.

Compensating transactions have been used to recover from the effects of long-running or committed transactions [26] and for recovery in multi-level systems designed to increase concurrency [28]. 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 are commonly employed to recover from data corruption. However, backups revert data based on time and can lose 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 [25] that taint certain file updates, and reverting the effects of only the tainted updates [24, 38]. This method is too coarse-grained for database 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 a backup approach.

Operator Undo [21] is a powerful framework for application recovery. The authors use it to recover from e-mail configuration bugs, but the framework requires modifying applications to serialize requests for replayed. It also requires separating persistent data and special recovery procedures for each type of application request. By focusing on web applications with well-defined interfaces, we can provide similar functionality without modifying applications.

Causeway [22] provides operating system support for metadata (e.g., request id) transfer across the tiers of an application. A similar idea was used by Magpie [20] in which the request execution paths were used to diagnose application failures. Unlike these systems, our work utilizes the well-defined interfaces in a web application's tiers and passes metadata across tiers to log and correlate them without requiring any modifications to the applications.

There have been several proposals for using online taint analysis for securing web applications [30, 31]. Unlike these approaches, we use tainting only after a failure occurs, to follow the effects of a bug.

Our work is also motivated by various approaches for dealing with software configuration problems. PeerPressure [35] uses statistical analysis of multiple systems to find and suggest a working configuration. Chronus [36] pinpoints a configuration problem by using predicates that determine whether the system is working correctly. AutoBash [32] aims to detect configuration problems and suggest corrective actions based on causality analysis.

3 Our Approach

This section describes the design of our recovery system. Our aim is to help the administrator 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. First, we present the application model assumed by our recovery system. Section 3.2 provides an overview and the rest describes 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 using 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 [25], but these channels can also cause contamination to spread more easily [24]. 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. Using replay, we track 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 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. If the web application does not use transactions, each query is treated as a separate transaction via the database 'autocommit' feature. Our system does not purge the undo log entries for a transaction immediately after the transaction is committed, but after a user-configurable time. Transactions occurring before this time are considered stable and their effects cannot be reverted. Finally, we assume that the database uses a serializable isolation level so that the database transactions can be replayed correctly.

3.2 System Overview

Our system consists of a monitoring component operating during run-time (on-line phase), and two components that perform analysis and data recovery after corruption is detected (post-corruption 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 request granularity. 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 guide the administrator through the recovery process. The analysis component tracks dependencies across the application tiers, helping the administrator determine corruption related events, and is crucial for effective recovery. The recovery component generates compensating transactions to selectively revert the effects of database operations that caused 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 in-

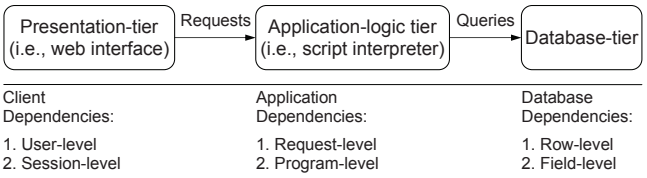

Figure 1. Dependencies across layers.

teraction (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 reducing application-level inconsistencies after the recovery operation as shown later in Section 4.2.

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* mappings, together with the database undo log, allow selectively reverting the effects of all persistent data modifications performed by a request.

The transaction mapping is an index into the database undo log. The key of the index, which we call a transaction ID, is the commit log sequence number (LSN) of the transaction. The transaction ID is ordered in transaction execution order, since we assume that the database uses serializable isolation. This ordering is important for replaying requests, as described later. The request mapping logs the transaction ID of all the transactions issued by each request. It also logs the queries issued by each transaction and some application-specific information described later. This instrumentation does not require any changes to the application code, and it does not depend on application correctness.

3.4 Analysis

The analysis component helps determine data corruption or loss related activities, and is crucial for effective recovery. Before the analysis, the current state of the application (i.e., database tables) is saved. The analysis is performed in a sandbox environment. 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 1. These dependencies help track contaminated data across the multiple tiers of the application.

3.4.1 Database Dependencies

Database dependencies are generated 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 [19, 27]. As shown in Figure 2, 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.

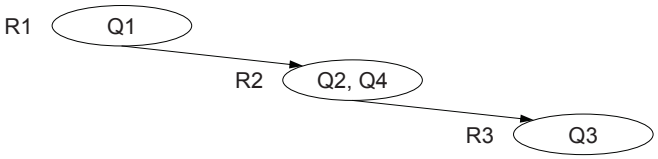

Figure 2. A request dependency graph.

The analysis component needs to know the read and write set of each query to generate a dependency. The monitor captures row-level write sets, because the database already maintains undo information at the row level. However, databases do not log read set information, because they do not need it and this logging imposes significant overheads. This problem has been addressed previously 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 [19]. However, this method requires manual creation of a template for each query issued by the application. The second method is instrumenting the database to generate and store the dependencies during the on-line phase [23]. This approach generates dependencies more accurately, but affects performance during normal operation.

After corruption is detected, the administrator uses our tools to identify one or more initial requests that trigger the bug or vulnerability in the application. Then the analysis component generates dependent requests 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, this larger dependency graph only affects the time to perform recovery, but not the overall accuracy of our solution, because we use application-level tainting as described below.

3.4.2 Application Dependencies

The dependencies described above apply to entire requests and are tracked transitively. This coarse-grained approach 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 2 shows that request R2 depends on 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 it read from R1 using Q2, while R3 only depends on Q4.

We use dynamic tainting to track application-logic dependencies within a request (shown as program-level dependencies in Figure 1) to prune *both* these types of false dependencies from the dependency graph. This approach essentially validates a cross-request dependency. The analysis component starts by tainting the initial request(s) in the dependency

graph and replaying them. It then replays requests that have incoming edges in the dependency graph and uses tainting to prune outgoing edges that are created by untainted queries. A scheduler orders all the requests in the dependency graph to replay them in the transaction ID or serialization order. We replay requests by capturing all HTTP request parameters during the monitoring phase. While web requests are mostly deterministic, our system detects any non-determinism 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 analysis process.

Our implementation uses a taint-based PHP interpreter [33]. 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. Unlike previous work that implemented row-level database dependencies, tainting allows us to implement more accurate field-level dependencies. Furthermore, these dependencies allow us to take advantage of *blind writes* (a query overwrites a field without reading it) for breaking dependencies. With row-level dependency, a blind write requires the entire row to be overwritten.

3.4.3 Client Dependencies

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: an administrator might know that the data corruption started with a specific user and start the dependency analysis by 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 session is responsible for the data corruption and there are no other dependencies.

Client dependencies are not directly available at the application-logic or database level. We derive these dependencies by using application-specific code in our monitor component, but without requiring any changes to the applications themselves. For example, session information is typically available in request parameters.

3.5 Recovery

The recovery component provides tools that simplify the recovery process. These tools provide information (e.g., time line of requests that affected specific tables, generated pages), helping the user identify requests that are the root cause of the failure, and serving as the starting point for the analysis component. For example, the root cause can be determined by reverting requests using binary-search [36], until the administrator determines which request caused the corruption.

After the analysis component generates the set of tainted requests, the recovery component uses the information in the database log and our request and transaction mappings to generate compensating transactions. For each update operation in a 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 [19]. These transactions are applied in reverse serialization order on the current state of the database and they selectively revert the effects of the database operations issued by the tainted requests. Unlike redo recovery [21], our recovery component does not replay application requests, and thus does not require any application-specific information. More details of our approach are available elsewhere [18].

Table 1. Modifications to existing software.

Component	Existing Software	Changed Lines
DB 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

3.6 Implementation

We have implemented a prototype of our recovery system for the PHP scripting engine and the MySQL database. Table 1 shows the number of lines of code we added or changed to implement our system. Note the majority of the code lies in the recovery component. Our changes to the PHP engine and MySQL are relatively small, and thus, it should be relatively easy to port our system to other languages and databases.

Implementing database tainting by modifying the database would have required significant changes to MySQL to support all SQL functionality. Instead, we implemented tainting with query rewriting by slightly modifying JSQLParser [10]. We modify the database tables to store a per-row or per-field taint bit and rewrite queries during replay. This approach is simpler to implement and provides significant flexibility to implement different dependency policies described later.

4 Evaluation

We evaluate our system in terms of how well it helps recover from data corruption caused by bugs found in popular web applications. We also measure our system's performance and space overheads. For our experiments, we used MySQL configured with its transactional storage engine (InnoDB).

4.1 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 different dependency policies described below. Our evaluation compares the recovery accuracy of these policies.

1) Request-level dependency with row-level tainting (request-row): This policy is the most conservative depen-

dency 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 (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. The taint is propagated throughout the request. When a query with tainted values is executed, the taint information is saved in the database at a row granularity preserving taints across requests.

3) Database-level dependency with row-level tainting (database-row): This policy implements previously proposed recovery methods [19]. It propagates taints when queries read tainted rows and update other rows. Since this policy does not consider application-level dependencies, such as dependencies between the queries of a request, it may fail to identify all the effects of the corruption. Also, the recovery reverts operations at the query level rather than the request level, probably resulting in application-level inconsistencies.

4) Program-level dependency with field-level tainting (program-field): This policy is similar to 2), except that taints are stored in the database at a field granularity, which also allows us to take advantage of blind writes.

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

Our analysis tools can also incorporate administrator knowledge about the application and help her determine the effects of data corruption more accurately. An administrator can whitelist tables, columns, rows or even fields, to stop taint propagation at the database tier. Our replay logs collect sufficient information about taint propagation, at the database and application layers, which is useful for generating whitelists. We used this approach to create whitelists for our evaluation.

We have also implemented a request profiler detecting requests with different application-level semantics, such as adding a comment, editing a post or updating a user. The profiler identifies request types based on the queries issued (e.g., INSERT), their order, and the database items (i.e., tables, columns) they accessed. The profiler generates a list of request types together with heavily accessed data items suitable for whitelisting. The intuition is that these items may cause significant taint propagation, but the frequent requests and accesses are unlikely to be a cause of corruption. Once the administrator whitelists these items, the tainting engine ignores them during replay, generating a new set of dependencies, thus giving the administrator a better understanding of how the corruption may have propagated.

4.2 Recovery Accuracy

We evaluate the accuracy of our dependency policies by triggering five real bugs in popular web applications, including Wordpress, Drupal and Gallery2. We investigated bug repositories and selected these bugs as follows: 1) data was corrupted/lost and there was no easy way to restore it (except using backups which may lose valid data), 2) the bugs were related to the application-logic and not the underlying software (e.g., PHP, MySQL). We describe these bugs, failure scenarios, the correct recovery actions and report how our recovery system performed. We assume that the administrator has identified the root cause of the corruption as explained in Section 3.5, and thus, the initially tainted request is known.

For our evaluation, we define *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 lost. We use three metrics to measure recovery accuracy. First, we determine whether recovery operations cause application-level inconsistencies that break application functionality. Second, we measure false positives, which are requests that are marked tainted even though they are unrelated to the corruption and will cause data loss during recovery. Third, we measure false negatives, which are requests that are not marked tainted, but whose effects should be reverted. These will cause corruption to linger in the application after recovery, possibly causing problems in the future.

4.2.1 Results

We summarize the results of various dependency analysis policies in Table 2 and Table 3. In Table 2, the second column shows the total number of requests we had to replay for the dependency analysis. All bugs had one initial request that corrupted data and each replay starts with one initially tainted request. The ”requests to undo” column shows the number of requests the administrator needs to undo to correctly recover from data corruption. The next column shows the dependency policy used; ’none’ indicating that no dependency information is considered for undo. The last two columns present the accuracy of the policies in terms of false positive and negative requests. The false positive numbers are without and with whitelisting. In Table 3, we present the results of database-level dependency policies. Since database-level policies only create dependencies across queries, all the numbers are in terms of queries. The last column shows the inconsistencies that are encountered after undoing these queries. The policies in Table 2 did not have any inconsistencies.

Many bugs we investigated did not corrupt data after the initial request and thus the ‘none’ policy (no dependency analysis) works well. One bug in Drupal created dependencies, and hence false negatives when dependencies are not considered. Note that we do not know beforehand whether a corruption will create dependencies, and thus dependency analysis provides useful information during recovery.

Table 2 shows that the request-level dependency policies suffer from high false positive rates, while Table 3 shows that the database-level policies can have many false negatives and inconsistencies. These results show that web applications

Table 2. Recovery accuracy for request-level and program-level dependency policies. The false positives column shows numbers without and with table whitelisting, respectively.

Case	Total Number of Requests	Requests to Undo	Dep. 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 voting information	118	7	none	0	6
			request-row	111/100	0
			program-row	95/89	0
			program-field	89/0	0
Drupal - lost comments	117	1	none	0	0
			request-row	116/102	0
			program-row	100/93	0
			program-field	95/0	0
Gallery2 - removing permissions	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 3. Recovery accuracy of database-level dependency policies. All numbers indicate queries.

Case	Queries to Undo	Dep. Policy	False Positives	False Negatives	Inconsistencies after Undo
Wordpress - link category rename	23	database-row	0	15	The count value does not match the actual number of links.
		database-field	0	21	
Drupal - lost voting information	38	database-row	86	16	The poll_votes table has duplicate entries.
		database-field	0	18	
Drupal - lost comments	24	database-row	116	0	none
		database-field	0	0	
Gallery2 - removing permissions	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	

generally expect that requests execute atomically, and thus recovery should be performed at a request granularity to minimize inconsistencies in the application after recovery. Table 2 also shows that the program-field dependency policy has the least number of false positives and no false negatives.

Although database-level policies can cause application-level inconsistency, they tend to have fewer false positives than the others. Thus, an administrator can compare the outputs of the database and program policies to derive the correct recovery actions more accurately and rapidly. The request policies do not require replay and can be useful if the program policy replay fails (e.g., incomplete implementation).

Below, we describe one bug for each application in more detail. For each case, we provide an overview of the application, background information for the corresponding bug and explain the results of each dependency policy. These results show that our approach is essential for data recovery, because bugs can be complex and it is hard to know what was corrupted without dependency analysis.

4.2.2 Wordpress: Link Category Rename

Wordpress is a popular blogging application that allows users to create content (e.g., posts, links) and associate it with categories to group and present it in a more organized way.

Scenario: An administrator already has some links associated with a certain category, `cat_1`. To edit a category's name, she has to click on it. A bug [14] allows her to rename `cat_1` to an empty string. She can still associate links with this category by selecting its checkbox. She adds new links associated with this category and others (e.g., {`old_cat_1`, `cat_2`}, {`cat_2`, `cat_3`}) and changes some settings.

Correct recovery actions: Undo the rename operation.

Background: Wordpress maintains links, terms (i.e., categories) and their types (i.e., belonging to posts or links) in separate tables (i.e., `links`, `terms`, `term_taxonomy`). Another table stores the relationships between the content and the terms (i.e., `term_relationships`). After querying this table, the number of links belonging to a certain category

is stored in the `count` column in the `term_taxonomy` table. This field is used for fast access when generating a page.

Results: The request-row policy marks many requests as falsely dependent because of an actively shared table (e.g., `options`). It does not consider whether tainted data a request reads from the above mentioned tables is used to update the database, instead it conservatively taints the request that updates a row in the `options` table. Since all requests query this table, they get tainted leading to many false positives.

The program-row policy reduces the number of false positives, because data flow at the program-level prevents taint from spreading to the `options` table. These false positives are caused by the row-level tainting granularity. When the administrator adds links with other categories than the renamed one (i.e., {`cat_2, cat_3`}), these operations get tainted, because each of these categories were previously used with the renamed category, such as {`old_cat_1, cat_2`} and {`old_cat_1, cat_3`}, causing false positives. The finer-grained program-field policy has fewer false positives and no false negatives. Field-level tainting improves accuracy, recognizing the addition of links associated with {`cat_2, cat_3`} as independent of `old_cat_1`. The six false positives are caused, because these requests added new links associated with `old_cat_1` and updated a link belonging to it.

Wordpress associates a new link with a category in three steps: 1) the relation between the link and the category is inserted into the `term_relationships` table. 2) this table is queried for the number of links associated with the category. 3) this number is used to update the `count` field of the category in the `term_taxonomy` table. The database-row policy only marks the third step as tainted, because it reads the category's row that was tainted previously when another link associated with `old_cat_1` was added. The insert operation in the first step was not marked as tainted, because it did not read any tainted rows. Reverting only the update operation will cause an inconsistency in the application, because the actual number of links belonging to a category in the `term_relationships` table will not match the `count` value in `term_taxonomy` table. In contrast, the database-field policy misses all other related steps (i.e., creating a relationship with that category), because the `count` is blindly overwritten resetting its taint. Exploiting blind writes via field-level tainting to break dependencies is desirable; however, a database-level policy can have false negatives.

Discussion: The addition of the links associated with `old_cat_1` and updating an existing link in this category may be considered as dependent on the initial corrupting request because of the explicit data dependency between the requests (i.e., `old_cat_1`'s id is used to create the relationship). However, this 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 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 of the small number of tainted requests.

4.2.3 Drupal: Lost Voting Information

Drupal allows an administrator to create a poll with multiple choice options via a module. One can control who can vote (e.g., only registered users). A user can only vote once and the application asserts this by keeping track of who has voted.

Scenario: An administrator creates a poll for registered users. After some users have voted, the administrator fixes a typo in the poll contents. A bug [7] causes the information about who has voted to be lost, allowing repeat votes. This creates an inconsistency in the application, because the sum of votes becomes greater than the number of users.

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

Background: Drupal maintains session data in the `sessions` table and retrieves it at the beginning of each request to obtain the associated user's id for permission checks. The table is updated with a timestamp and other related data at the end of each request. The poll content (e.g., the text of the choices, number of votes) is saved in the `poll_choices` table, whereas the `poll_votes` table tracks who has voted.

Results: The request-row policy marked all requests as tainted because of the shared session data. The updated poll is put to the front page. Since every session starts requesting the front page, the taint is spread between different users.

The program-row policy tainted requests that read and use the poll data during voting. When different users vote on the same poll, their sessions get tainted, causing many false positives. To our surprise, the finer-grained program-field policy did not reduce the false positives much, even though the session update was a blind write. Our investigation revealed that this update was using a tainted value, (i.e., the user id) that got tainted when the initial request updated the session data.

The database-row policy marked queries in many requests to be tainted. All false positives were related to the `sessions` and `users` table. On the other hand, the database-field policy marked the queries that updated the number of votes in the `poll_choices` table, but not the queries inserting information about who has voted into the `poll_votes` table. Reverting the effects of only the updates would create an inconsistency in the application, because the `poll_votes` table would have duplicate entries.

Discussion: After examining our logs, we whitelisted `sessions`, `history` and `watchdog` tables. The request-row policy still produced many false positives via the `users` table because of the updates to the `access` timestamp field. In contrast, the field-level policy resets the taint for this field, as the update is a blind write, resulting in no false positives.

The second and third rows in Table 2 show that 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 plays an important

role on determining what kind of dependencies really exist and thus, the correct recovery actions. We can help the administrator by providing detailed results and analysis logs for each policy and how they generate the dependencies.

4.2.4 Gallery2: Remove Permissions Breaks Application

Gallery2 has a fine-grained access control mechanism. An administrator can assign various capabilities (e.g., view) for specific pictures or whole albums to specific users and groups.
Scenario: An administrator temporarily removes other users' permissions to view the entire gallery. She then creates sub-albums under the main album, and adds users and groups. After she logs out, a bug [9] causes the application to show an error message, stopping the application entirely and making the web interface no longer available.
Correct recovery actions: Restore permissions to view the gallery. The administrator considers sub-albums' additions irrelevant to the corruption.
Background: Gallery2 uses a global sequence id for every item (e.g., picture, album) inserted into the database making this id their primary key in their respective tables. It stores the last value in the `SequenceId` table. A global `Entity` table stores each item and its associated data. For entities, such as sub-albums, a `ChildEntity` table stores the relationships. The table `SessionMap` tracks open sessions, associating each session with the corresponding user's id.
Results: The request-row policy marks almost every request as tainted. The session data becomes tainted with the initial request and the taint is spread to other requests when it is read at the beginning of each request. The program-row and the program-field policies also have many false positives. Similar to Drupal, the user id retrieved from the tainted session data is propagated throughout the request and is used to update the session data at the end of the request, spreading the taint.

The database-row policy marks the update queries to the `SessionMap` and `SequenceId` tables as tainted. The session data may be considered temporary and ignored during recovery; however, the sequence update queries are important for correct functionality of the application. Every insertion of an item 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 is already assigned to another item. This will certainly cause undesired behaviour, since the same id is already in use in the `Entity` table.

On the other hand, the database-field policy only marks the sequence id updates, as the old value of the field is read during the update. The taint for the session data is reset with a blind write. The used value cannot be tainted, because this policy does not propagate the taint throughout the request.
Discussion: We examined our logs and whitelisted the `SessionMap` and `SequenceId` tables, significantly reducing the false positives. The rest is caused by the parent-child relationship between the main gallery and the added sub-albums. The id of the main gallery, which was tainted by the initial request, is used to insert new entries to the `ChildEntity` table, resulting in false positives. Choosing the correct recovery actions in this case is also application-specific. One may argue that these requests are really dependent, because viewing sub-albums is prevented if the parent album is not accessible. Thus, sub-albums' additions should be dependent on the first request that removed the permissions from the parent (i.e., main) album. The number of false positives is small and manageable. Also, our replay logs provide enough information on how taint spreads, so that the administrator can decide for the correct recovery actions. The database-level policies not only had false positives, but reverting them caused the inconsistency given in Table 3.

Table 4. Throughput and latency overhead

Monitors Enabled	Throughput (req./sec) and Overhead	Latency (ms) and Overhead
None	31.31(0)	3637 (0)
MySQL	31.50 (-0.61%)	3614 (-0.62%)
PHP	29.55 (5.62%)	3853 (5.95%)
PHP & MySQL	30.06 (3.99%)	3787 (4.12%)

4.3 Performance

We report our system's performance and disk space requirements using the TPC-W industry benchmark. We measure the throughput and logging overhead of our monitors. The server was an Intel Pentium 4 2.80 GHz with dual CPU on Ubuntu Linux 8.04 with Apache 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. Both machines were connected via a 1Gb link. We report averages of at least 15 runs each lasting 30 minutes.

4.3.1 Throughput Overhead

To measure the throughput overhead of our monitors, we ran tests by enabling them separately and both of them together. The results can be found in Table 4. Our instrumentation incurs a maximum of 4% overhead in throughput and latency, when both monitors are on. The overhead is mostly caused by our PHP instrumentation, which can be further optimized.

Our database instrumentation improves performance slightly (compare the first and second rows in Table 4), because our monitor disables the periodic purge of the undo information of committed transactions. For details, see [18].

4.3.2 Disk Space Overhead

The disk overhead arises from disabling the undo log purge, keeping the mapping between transactions and modified rows in the database and the PHP log. The logs take about 4 KB per request (3.08 KB for the PHP log) for TPC-W, totaling 196 MB for a 30 minute run (9.19 GB per day). Compressing the PHP log reduces the log size to 2.23 GB per day. A 250 GB disk can save logs of about 104 days. Given current disk capacities, we believe that this overhead is acceptable for providing a generic recovery system for web applications.

5 Conclusion

A web application bug causing data loss or corruption can affect many users, because these applications store data at the server side. We have described the design of a recovery system for web applications that helps administrators recover from data corruption. Our system tracks and correlates requests across multiple tiers of the application with modest changes to existing software. A significant challenge in data recovery is determining the correct set of dependent requests. Our evaluation compared various dependency schemes, including our proposed tainting-based scheme, and showed how they allow an administrator to successfully diagnose and recover from various corruption scenarios and real bugs. Our prototype implementation with MySQL and PHP shows that generic data recovery functionality can be obtained with little overhead and no modifications to the web applications.

References

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

[2] Amazon shuts after price error. http://news.bbc.co.uk/2/hi/business/2864461.stm.

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

[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.

[6] Drupal Bug Report: Big bug in management comments. http://drupal.org/node/67745.

[7] Drupal Bug Report: Editing a poll clears all old votes. http://drupal.org/node/67895.

[8] Drupal Group: Remove warning modal dialogs and replace them with undo. http://groups.drupal.org/node/21913.

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

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

[11] Wordpress - Blog Tool and Publishing Platform. http://wordpress.org.

[12] Wordpress Codex - Managing Plugins. http://codex.wordpress.org/Managing_Plugins.

[13] Wordpress Codex: IRC Meetup. http://codex.wordpress.org/IRC_Meetups/2007/September/September26RawLog.

[14] Wordpress Ticket: Links category can be set to blank. http://core.trac.wordpress.org/ticket/7336.

[15] Wordpress Ticket: Unprivileged users can perform some actions on pages they aren't allowed to access. http://trac.wordpress.org/ticket/4748.

[16] Wordpress Ticket: Users without capability "create_users" can add new users. http://trac.wordpress.org/ticket/6662.

[17] Wordpress Ticket: Users without unfiltered_html capability can post arbitrary html. http://trac.wordpress.org/ticket/4720.

[18] I. E. Akkus. Data recovery for web applications. Master's thesis, University of Toronto. https://tspace.library.utoronto.ca/handle/1807/18132.

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

[20] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 259–272, 2004.

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

[22] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: Support for Controlling and Analyzing the Execution of Web-Accessible Applications. In *Middleware*, 2005.

[23] T.-C. Chiueh and D. Pilania. Design, implementation, and evaluation of a repairable database management system. In *Proceedings of the Annual Computer Security Applications Conference*, pages 179–188, 2004.

[24] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 163–176, Oct. 2005.

[25] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 223–236, Oct. 2003.

[26] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

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

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

[29] B. Mediratta. Gallery photo album organizer. http://gallery.menalto.com/, 2004.

[30] S. Nanda, L.-C. Lam, and T.-C. Chiueh. Dynamic multiprocess information flow tracking for web application security. In *Proceedings of the ACM/IFIP/USENIX international conference on Middleware*, pages 1–20, 2007.

[31] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[32] Y.-Y. Su, M. Attariyan, and J. 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.

[33] W. Venema. Taint support for PHP. ftp://ftp.porcupine.org/pub/php/index.html.

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

[35] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 245–258, Dec. 2004.

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

[37] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, 2006.

[38] N. Zhu and T.-C. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.