Intrusion Recovery for Database-backed Web Applications

Ramesh Chandra, Taesoo Kim, Meelap Shah,
Neha Narula, and Nickolai Zeldovich

MIT CSAIL

ABSTRACT

WARP is a system that helps users and administrators of web applications recover from intrusions such as SQL injection, cross-site scripting, and clickjacking attacks, while preserving legitimate user changes. WARP repairs from an intrusion by rolling back parts of the database to a version before the attack, and replaying subsequent legitimate actions. WARP allows administrators to *retroactively patch* security vulnerabilities—i.e., apply new security patches to past executions—to recover from intrusions without requiring the administrator to track down or even detect attacks. WARP's *time-travel database* allows fine-grained rollback of database rows, and enables repair to proceed concurrently with normal operation of a web application. Finally, WARP *captures and replays user input at the level of a browser's DOM*, to recover from attacks that involve a user's browser. For a web server running MediaWiki, WARP requires no application source code changes to recover from a range of common web application vulnerabilities with minimal user input at a cost of 24–27% in throughput and 2–3.2 GB/day in storage.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services.*

General Terms: Security.

1 INTRODUCTION

Many web applications have security vulnerabilities that have yet to be discovered. For example, over the past 4 years, an average of 3–4 previously unknown cross-site scripting and SQL injection vulnerabilities were discovered *every single day* [27]. Even if a web application's code contains no vulnerabilities, administrators may misconfigure security policies, making the application vulnerable to attack, or users may inadvertently grant their privileges to malicious code [8]. As a result, even well-maintained applications can and do get compromised [4, 31, 33]. Furthermore, after gaining unauthorized access, an attacker could use web application functionality such as Google Apps Script [7, 9] to install persistent malicious code, and trigger it at a later time, even after the underlying vulnerability has been fixed.

Despite this prevalence of vulnerabilities that allows adversaries to compromise web applications, recovering from a newly discovered vulnerability is a difficult and manual process. Users or administrators must manually inspect the application for signs of an attack that exploited the vulnerability,

SOSP '11, October 23–26, 2011, Cascais, Portugal.

and if an attack is found, they must track down the attacker's actions and repair the damage by hand. Worse yet, this time-consuming process provides no guarantees that every intrusion was found, or that all changes by the attacker were reverted. As web applications take on more functionality of traditional desktop applications, intrusion recovery for web applications will become increasingly important.

This paper presents WARP[1], a system that automates repair from intrusions in web applications. When an administrator learns of a security vulnerability in a web application, he or she can use WARP to check whether that vulnerability was recently exploited, and to recover from any resulting intrusions. Users and administrators can also use WARP to repair from configuration mistakes, such as accidentally giving permissions to the wrong user. WARP works by continuously recording database updates, and logging information about all actions, such as HTTP requests and database queries, along with their input and output dependencies. WARP constructs a global dependency graph from this logged information, and uses it to *retroactively patch* vulnerabilities by rolling back parts of the system to an earlier checkpoint, fixing the vulnerability (e.g., patching a PHP file, or reverting unintended permission changes), and re-executing any past actions that may have been affected by the fix. This both detects any intrusions that exploited the vulnerability and reverts their effects.

To illustrate the extent of challenges facing WARP in recovering from intrusions in a web application, consider the following worst-case attack on a company's Wiki site that is used by both employees and customers, where each user has privileges to edit only certain pages or documents. An attacker logs into the Wiki site and exploits a cross-site scripting (XSS) vulnerability in the Wiki software to inject malicious JavaScript code into one of the publicly accessible Wiki pages. When Alice, a legitimate user, views that page, her browser starts running the attacker's code, which in turn issues HTTP requests to add the attacker to the access control list for every page that Alice can access, and to propagate the attack code to some of those pages. The adversary now uses his new privileges to further modify pages. In the meantime, legitimate users (including Alice) continue to access and edit Wiki pages, including pages modified or infected by the attack.

Although the Retro system previously explored intrusion recovery for command-line workloads on a single machine [14], WARP is the first system to repair from such attacks in web applications. Recovering from intrusions such as the example above requires WARP to address three challenges not answered by Retro, as follows.

First, recovering from an intrusion (e.g., in Retro) typically requires an expert administrator to detect the compromise and to track down the source of the attack, by analyzing database entries and web server logs. Worse yet, this process must be repeated every time a new security problem is discovered, to determine if any attackers might have exploited the vulnerability.

Second, web applications typically handle data on behalf of many users, only a few of which may have been affected by an attack. For a popular web application with many users, reverting all users' changes since the attack or taking the application offline for repair is not an option.

Third, attacks can affect users' browsers, making it difficult to track down the extent of the intrusion purely on the server. In our example attack, when Alice (or any other user) visits an infected Wiki page, the web server cannot tell if a subsequent page edit request from Alice's browser was caused by Alice or by the malicious JavaScript code. Yet an ideal system should revert all effects of the malicious code while preserving any edits that Alice made from the same page in her browser.

To address these challenges, WARP builds on the rollback-and-reexecute approach to repair taken by Retro, but solves a new problem—repair for distributed, web-based applications—using three novel ideas. First, WARP allows administrators to *retroactively apply security patches* without having to manually track down the source of each attack, or even having to decide whether someone already exploited the newfound vulnerability. Retroactive patching works by re-executing past actions using patched application code. If an action re-executes the same way as it did originally, it did not trigger the vulnerability, and requires no further re-execution. Actions that re-execute differently on patched application code could have been intrusions that exploited the original bug, and WARP repairs from this potential attack by recursively re-executing any other actions that were causally affected.

Second, WARP uses a *time-travel database* to determine dependencies between queries, such as finding the set of legitimate database queries whose results were influenced by the queries that an adversary issued. WARP uses these dependencies to roll back just the affected parts of the database

[1] WARP stands for Web Application RePair.

during repair. Precise dependencies are crucial to minimize the amount of rollback and re-execution during repair; otherwise, recovering from a week-old attack that affected just one user would still require re-executing a week's worth of work. Precise dependency analysis and rollback is difficult because database queries operate on entire tables that contain information about all users, instead of individual data objects related to a single user. WARP addresses this problem by partitioning tables on popular lookup keys and using partitions to determine dependencies at a finer granularity than entire database tables. By tracking multiple versions of a row, WARP can also perform repair concurrently with the normal operation of the web application.

Third, to help users recover from attacks that involve client machines, such as cross-site scripting, WARP performs *DOM-level replay of user input*. In our example, WARP's repair process will first roll back any database changes caused by Alice's browser, then open a repaired (and presumably no longer malicious) version of the Wiki page Alice visited, and replay the inputs Alice originally provided to the infected page. Operating at the DOM level allows WARP to replay user input even if the underlying page changed (e.g., the attack's HTML and JavaScript is gone), and can often preserve legitimate changes without any user input. WARP uses a client-side browser extension to record and upload events to the server, and uses a browser clone on the server to re-execute them.

To evaluate our ideas in practice, we built a prototype of WARP, and ported MediaWiki, a popular Wiki application, to run on WARP. We show that an administrator using WARP can fully recover from six different attacks on MediaWiki, either by retroactively applying a security patch (for software vulnerabilities), or by undoing a past action (for administrator's mistakes). WARP requires no application changes, incurs a 24–27% CPU and 2–3.2 GB/day storage cost on a single server, and requires little user input.

In the rest of this paper, we start with an overview of WARP's design and assumptions in §2. We describe the key aspects of WARP's design—retroactive patching, the time-travel database, and browser re-execution—in §3, §4, and §5 respectively. §6 presents our prototype implementation, and §7 explains how all parts of WARP fit together in the context of an example. We evaluate WARP in §8, and compare it to related work in §9. §10 discusses WARP's limitations and future work, and §11 concludes.

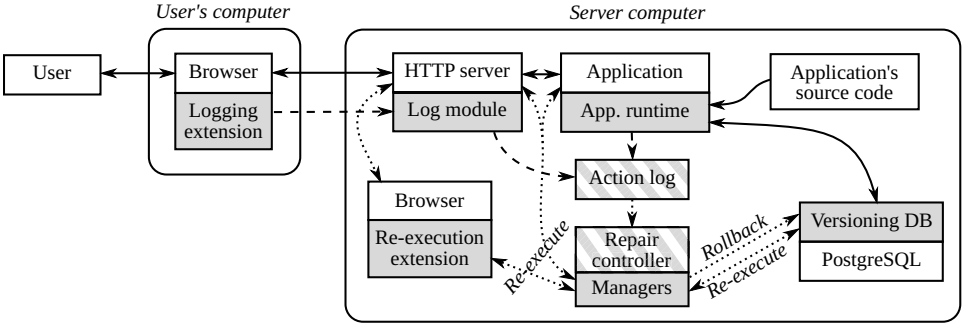

Figure 1: Overview of WARP's design. Components introduced or modified by WARP are shaded; components borrowed from Retro are striped. Solid arrows are the original web application interactions that exist without WARP. Dashed lines indicate interactions added by WARP for logging during normal execution, and dotted lines indicate interactions added by WARP during repair.

2 OVERVIEW

The goal of WARP is to recover the integrity of a web application after it has been compromised by an adversary. More specifically, WARP's goal is to undo all changes made by the attacker to the system, including all indirect effects of the attacker's changes on legitimate actions of other users (e.g., through cross-site scripting vulnerabilities), and to produce a system state as if all the legitimate changes still occurred, but the adversary never compromised the application.

WARP's workflow begins with the administrator deciding that he or she wants to make a retroactive fix to the system, such as applying a security patch or changing a permission in the past. At a high

level, WARP then rolls back the system to a checkpoint before the intended time of the fix, applies the fix, and re-executes actions that happened since that checkpoint, to construct a new system state. This produces a repaired system state that would have been generated if all of the recorded actions happened on the fixed system in the first place. If some of the recorded actions exploited a vulnerability that the fix prevents, those actions will no longer have the same effect in the repaired system state, effectively undoing the attack.

If the application is non-deterministic, there may be many possible repaired states, and WARP only guarantees to provide one of them, which may not necessarily be the one closest to the pre-repair state. In other words, non-deterministic changes unrelated to the attack may appear as a result of repair, and non-determinism may increase the number of actions re-executed during repair, but the repaired state is guaranteed to be free of effects of attack actions. Also, due to changes in system state during repair, some of the original actions may no longer make sense during replay, such as when a user edits a Wiki page created by the attacker and that page no longer exists due to repair. These actions are marked as *conflicts* and WARP asks the user for help in resolving them.

WARP cannot undo disclosures of private data, such as if an adversary steals sensitive information from Wiki pages, or steals a user's password. However, when private data is leaked, WARP can still help track down affected users. Additionally, in the case of stolen credentials, administrators can use WARP to retroactively change the passwords of affected users (at the risk of undoing legitimate changes), or revert just the attacker's actions, if they can identify the attacker's IP address.

The rest of this section first provides a short review of Retro, and then discusses how WARP builds on the ideas from Retro to repair from intrusions in web applications, followed by a summary of the assumptions made by WARP.

2.1 Review of Retro

Repairing from an intrusion in Retro, which operates at the operating system level, involves five steps. First, during normal execution, Retro records a log of all system calls and periodically checkpoints the file system. Second, the administrator must detect the intrusion, and track down the initial attack action (such as a user accidentally running a malware binary). Third, Retro rolls back the files affected by the attack to a checkpoint before the intrusion. Fourth, Retro re-executes legitimate processes that were affected by the rolled-back file (e.g., any process that read the file in the past), but avoids re-executing the attack action. Finally, to undo indirect effects of the attack, Retro finds any other processes whose inputs may have changed as a result of re-execution, rolls back any files they modified, and recursively re-executes them too.

A naïve system that re-executed every action since the attack would face two challenges. First, re-execution is expensive: if the attack occurred a week ago, re-executing everything may take another week. Second, re-execution may produce different results, for reasons that have nothing to do with the attack (e.g., because some process is non-deterministic). A different output produced by one process can lead to a conflict when Retro tries to re-execute subsequent processes, and would require user input to resolve. For example, re-executing sshd can generate a different key for an ssh connection, which makes it impossible to replay that connection's network packets. Thus, while Retro needs some processes to produce different outputs (e.g., to undo the effects of an attack), Retro also needs to minimize re-execution in order to minimize conflicts that require user input, and to improve performance.

To reduce re-execution, Retro checks for *equivalence* of inputs to a process before and after repair, to decide whether to re-execute a process. If the inputs to a process during repair are identical to the inputs originally seen by that process, Retro skips re-execution of that process. Thus, even if some of the files read by a process may have been changed during repair, Retro need not re-execute a process that did not read the changed parts of the file.

Retro's design separates the overall logic of rollback, repair, and recursive propagation (the *repair controller*) from the low-level details of file system rollback and process re-execution (handled by individual *repair managers*). During normal execution, managers record information about checkpoints, actions, and dependencies into a global data structure called an *action history graph*, and periodically garbage-collect old checkpoints and action history graph entries. A node in the action history graph logically represents the history of some part of the system over time, such as all versions of a certain file or directory. The action history graph also contains actions, such as a process executing

for some period of time or issuing a system call. An action has dependencies to and from nodes at a specific time, indicating the versions of a node that either influenced or were influenced by that action. During repair, the repair controller consults the action history graph, and invokes the managers as needed for rollback and re-execution. We refer the reader to Kim et al. [14] for further details.

2.2 Repairing web applications

WARP builds on Retro's repair controller to repair from intrusions in web applications. Figure 1 illustrates WARP's design, and its relation to components borrowed from Retro (in particular, the repair controller, and the structure of the action history graph). WARP's design involves the web browser, HTTP server, application code, and database. Each of these four components corresponds to a *repair manager* in WARP, which records enough information during normal operation to perform rollback and re-execution during repair.

To understand how WARP repairs from an attack, consider the example scenario we presented in §1, where an attacker uses a cross-site scripting attack to inject malicious JavaScript code into a Wiki page. When Alice visits that page, her browser runs the malicious code, and issues HTTP requests to propagate the attack to another page and to give the attacker access to Alice's pages. The attacker then uses his newfound access to corrupt some of Alice's pages. In the meantime, other users continue using the Wiki site: some users visit the page containing the attack code, other users visit and edit pages corrupted by the attack, and yet other users visit unaffected pages.

Some time after the attack takes place, the administrator learns that a cross-site scripting vulnerability was discovered by the application's developers, and a security patch for one of the source files—say, `calendar.php`—is now available. In order to retroactively apply this security patch, WARP first determines which runs of the application code may have been affected by a bug in `calendar.php`. WARP then applies the security patch to `calendar.php`, and considers re-executing all potentially affected runs of the application. In order to re-execute the application, WARP records sufficient information during the original execution[2] about all of the inputs to the application, such as the HTTP request. To minimize the chance that the application re-executes differently for reasons *other* than the security patch, WARP records and replays the original return values from non-deterministic function calls. §3 discusses how WARP implements retroactive patching in more detail.

Now consider what happens when WARP re-executes the application code for the attacker's initial request. Instead of adding the attacker's JavaScript code to the Wiki page as it did during the original execution, the newly patched application code will behave differently (e.g., pass the attacker's JavaScript code through a sanitization function), and then issue an SQL query to store the resulting page in the database. This SQL query must logically replace the application's original query that stored an infected page, so WARP first rolls back the database to its state before the attack took place.

After the database has been rolled back, and the new query has executed, WARP must determine what other parts of the system were affected by this changed query. To do this, during original execution WARP records all SQL queries, along with their results. During repair, WARP re-executes any queries it determines may have been affected by the changed query. If a re-executed query produces results different from the original execution, WARP re-executes the corresponding application run as well, such as Alice's subsequent page visit to the infected page. §4 describes the design of WARP's time-travel database in more detail, including how it determines query dependencies, how it re-executes queries in the past, and how it minimizes rollback.

When the application run for Alice's visit to the infected page is re-executed, it generates a different HTTP response for Alice's browser (with the attack now gone). WARP must now determine how Alice's browser would behave given this new page. Simply undoing all subsequent HTTP requests from Alice's browser would needlessly undo all of her legitimate work, and asking Alice to manually check each HTTP request that her browser made is not practical either. To help Alice recover from such attacks, WARP provides a browser extension that records all events for each open page in her browser (such as HTTP requests and user input) and uploads this information to the server. If WARP determines that her browser may have been affected by an attack, it starts a clone of her browser on the server, and re-executes her original input on the repaired page, without having to involve her. Since Alice's re-executed browser will no longer issue the HTTP requests from the XSS attack, WARP will

[2]We use the terms "original execution" and "normal execution" interchangeably.

recursively undo the effects of those requests as well. §5 explains how WARP's browser extension works in more detail.

If a user's actions depend on changes by the attacker, WARP may be unable to replay the user's original inputs in the browser clone. For example, if the attacker created a new Wiki page, and a curious user subsequently edited that page, WARP will not be able to re-execute the user's actions once the attack is undone. In this case, WARP signals a conflict and asks the user (or administrator) to resolve it. WARP cannot rely on users being always online, so WARP queues the conflict, and proceeds with repair.

When the user next logs in, WARP redirects the user to a conflict resolution page. To resolve a conflict, the user is presented with the original page they visited, the newly repaired version of that page, and the original action that the server is unable to replay on the new page, and is asked to specify what actions they would like to perform instead. For example, the user can ask WARP to cancel that page visit altogether. Users or administrators can also use the same mechanism to undo their own actions from the past, such as if an administrator accidentally gave administrative privileges to a user. §5 further discusses WARP's handling of conflicts and user-initiated undo.

2.3 Assumptions

To recover from intrusions, WARP makes two key assumptions. First, WARP assumes that the adversary does not exploit any vulnerabilities in the HTTP server, database, or the application's language runtime, does not cause the application code to execute arbitrary code (e.g., spawning a Unix shell), and does not corrupt WARP's log. Most web application vulnerabilities fall into this category [10], and §8 shows how WARP can repair from common attacks such as cross-site scripting, SQL injection, cross-site request forgery, and clickjacking.

Second, to recover from intrusions that involve a user's browser, our prototype requires the user to install a browser extension that uploads dependency information to WARP-enabled web servers. In principle, the same functionality could be performed purely in JavaScript (see §10), but for simplicity, our prototype uses a separate extension. WARP's server trusts each browser's log information only as much as it trusts the browser's HTTP requests. This ensures that a malicious user cannot gain additional privileges by uploading a log containing user input that tries to issue different HTTP requests.

If one user does not have our prototype's extension installed, but gets compromised by a cross-site scripting attack, WARP will not be able to precisely undo the effects of malicious JavaScript code in that user's browser. As a result, server-side state accessible to that user (e.g., that user's Wiki pages or documents) may remain corrupted. However, WARP will still inform the user that his or her browser might have received a compromised reply from the server in the past. At that point, the user can manually inspect the set of changes made to his data from that point onward, and cancel his or her previous HTTP requests, if unwanted changes are detected.

3 RETROACTIVE PATCHING

To implement retroactive patching, WARP's application repair manager must be able to determine which runs of an application may have been affected by a given security patch, and to re-execute them during repair. To enable this, WARP's application repair manager interposes on the application's language runtime (PHP in our current prototype) to record any dependencies to and from the application, including application code loaded at runtime, queries issued to the database, and HTTP requests and responses sent to or from the HTTP server.

3.1 Normal execution

During normal execution, the application repair manager records three types of dependencies for the executing application code (along with the dependency's data, used later for re-execution). First, the repair manager records an input dependency to the HTTP request and an output dependency to the HTTP response for this run of the application code (along with all headers and data). Second, for each read or write SQL query issued by the application, the repair manager records, respectively, input or output dependencies to the database. Third, the repair manager records input dependencies on the source code files used by the application to handle its specific HTTP request. This includes the initial

PHP file invoked by the HTTP request, as well as any additional PHP source files loaded at runtime through `require` or `include` statements.

In addition to recording external dependencies, WARP's application manager also records certain internal functions invoked by the application code, to reduce non-determinism during re-execution. This includes calls to functions that return the current date or time, functions that return randomness (such as `mt_rand` in PHP), and functions that generate unique identifiers for HTTP sessions (such as `session_start` in PHP). For each of these functions, the application manager records the arguments and return value. This information is used to avoid re-executing these non-deterministic functions during repair, as we will describe shortly.

3.2 Initiating repair

To initiate repair through retroactive patching, the administrator needs to provide the filename of the buggy source code file, a patch to that file which removes the vulnerability, and a time at which this patch should be applied (by default, the oldest time available in WARP's log). In response, the application repair manager adds a new action to WARP's action history graph, whose re-execution would apply the patch to the relevant file at the specified (past) time. The application repair manager then requests that WARP's repair controller re-execute the newly synthesized action. WARP will first re-execute this action (i.e., apply the patch to the file in question), and then use dependencies recorded by the application repair manager to find and re-execute all runs of the application that loaded the patched source code file.

3.3 Re-execution

During re-execution, the application repair manager invokes the application code in much the same way as during normal execution, with two differences. First, all inputs and outputs to and from the application are handled by the repair controller. This allows the repair controller to determine when re-execution is necessary, such as when a different SQL query is issued during repair, and to avoid re-executing actions that are not affected or changed.

Second, the application repair manager tries to match up calls to non-deterministic functions during re-execution with their counterparts during the original run. In particular, when a non-deterministic function is invoked during re-execution, the application repair manager searches for a call to the same function, from the same caller location. If a match is found, the application repair manager uses the original return value in lieu of invoking the function. The repair manager matches non-deterministic function calls from the same call site in-order (i.e., two non-deterministic function calls that happened in some order during re-execution will always be matched up to function calls in that same order during the original run).

One important aspect of this heuristic is that it is strictly an optimization. Even if the heuristic fails to match up any of the non-deterministic function calls, the repair process will still be *correct*, at the cost of increased re-execution (e.g., if the application code generates a different HTTP cookie during re-execution, WARP will be forced to re-execute all page visits that used that cookie).

4 TIME-TRAVEL DATABASE

The job of WARP's time-travel database is to checkpoint and roll back the application's persistent data, and to re-execute past SQL queries during repair. Its design is motivated by two requirements: first, the need to minimize the number of SQL queries that have to be re-executed during repair, and second, the need to repair a web application concurrently with normal operation. This section discusses how WARP addresses these requirements.

4.1 Reducing re-execution

Minimizing the re-execution of SQL queries during repair is complicated by the fact that clients issue queries over entire tables, and tables often contain data for many independent users or objects of the same type.

There are two reasons why WARP may need to re-execute an SQL query. First, an SQL query that modifies the database (e.g., an `INSERT`, `UPDATE`, or `DELETE` statement) needs to be re-executed in order to re-apply legitimate changes to a database after rollback. Second, an SQL query that reads the

database (e.g., a `SELECT` statement, or any statement with a `WHERE` clause) needs to be re-executed if the data read by that statement may have changed as a result of repair.

To minimize re-execution of write SQL queries, the database manager performs fine-grained rollback, at the level of individual rows in a table. This ensures that, if one row is rolled back, it may not be necessary to re-execute updates to other rows in the same table. One complication lies in the fact that SQL has no inherent way of naming unique rows in a database. To address this limitation, WARP introduces the notion of a *row ID*, which is a unique name for a row in a table. Many web applications already use synthetic primary keys which can serve as row IDs; in this case, WARP uses that primary key as a row ID in that table. If a table does not already have a suitable row ID column, WARP's database manager transparently adds an extra `row_id` column for this purpose.

To minimize re-execution of SQL queries that read the database, the database manager logically splits the table into *partitions*, based on the values of one or more of the table's columns. During repair, the database manager keeps track of the set of partitions that have been modified (as a result of either rollback or re-execution), and avoids re-executing SQL queries that read from only unmodified partitions. To determine the partitions read by an SQL query, the database manager inspects the query's `WHERE` clause. If the database manager cannot determine what partitions a query might read based on the `WHERE` clause, it conservatively assumes that the query reads all partitions.

In our current prototype, the programmer or administrator must manually specify the row ID column for each table (if they want to avoid the overhead of an extra `row_id` column created by WARP), and the partitioning columns for each table (if they want to benefit from the partitioning optimization). A partitioning column need not be the same column as the row ID. For example, a Wiki application may store Wiki pages in a table with four columns: a unique page ID, the page title, the user ID of the last user who edited the page, and the contents of that Wiki page. Because the title, the last editor's user ID, and the content of a page can change, the programmer would specify the immutable page ID as the row ID column. However, the application's SQL queries may access pages either by their title or by the last editor's user ID, so the programmer would specify them as the partitioning columns.

4.2 Re-executing multi-row queries

SQL queries can access multiple rows in a table at once, if the query's `WHERE` clause does not guarantee a unique row. Re-executing such queries—where WARP cannot guarantee by looking at the `WHERE` clause that only a single row is involved—poses two challenges. First, in the case of a query that may read multiple rows, WARP must ensure that all of those rows are in the correct state prior to re-executing that query. For instance, if some of those rows have been rolled back to an earlier version due to repair, but other rows have not been rolled back since they were not affected, naïvely re-executing the multi-row query can produce incorrect results, mixing data from old and new rows. Second, in the case of a query that may modify multiple rows, WARP must roll back all of those rows prior to re-executing that query, and subsequently re-execute any queries that read those rows.

To re-execute multi-row read queries, WARP performs *continuous versioning* of the database, by keeping track of every value that ever existed for each row. When re-executing a query that accesses some rows that have been rolled back, and other rows that have not been touched by repair, WARP allows the re-executed query to access the old value of the untouched rows from precisely the time that query originally ran. Thus, continuous versioning allows WARP's database manager to avoid rolling back and reconstructing rows for the sole purpose of re-executing a read query on their old value.

To re-execute multi-row write queries, WARP performs *two-phase re-execution* by splitting the query into two parts: the `WHERE` clause, and the actual write query. During normal execution, WARP records the set of row IDs of all rows affected by a write query. During re-execution, WARP first executes a `SELECT` statement to obtain the set of row IDs matching the new `WHERE` clause. These row IDs correspond to the rows that would be modified by this new write query on re-execution. WARP uses continuous versioning to precisely roll back both the original and new row IDs to a time just before the write query originally executed. It then re-executes the write query on this rolled-back database.

To implement continuous versioning, WARP augments every table with two additional columns, `start_time` and `end_time`, which indicate the time interval during which that row value was valid. Each row R in the original table becomes a series of rows in the continuously versioned table, where

the end_time value of one version of R is the start_time value of the next version of R. The column end_time can have the special value ∞, indicating that row version is the current value of R. During normal execution, if an SQL query modifies a set of rows, WARP sets end_time for the modified rows to the current time, with the rest of the columns retaining their old values, and inserts a new set of rows with start_time set to the current time, end_time set to ∞, and the rest of the columns containing the new versions of those rows. When a row is deleted, WARP simply sets end_time to the current time. Read queries during normal execution always access rows with end_time $= \infty$. Rolling back a row to time T involves deleting versions of the row with start_time $\geq T$ and setting end_time $\leftarrow \infty$ for the version with the largest remaining end_time.

Since WARP's continuous versioning database grows in size as the application makes modifications, the database manager periodically deletes old versions of rows. Since repair requires that both the old versions of database rows and the action history graph be available for rollback and re-execution, the database manager deletes old rows in sync with WARP's garbage-collection of the action history graph.

4.3 Concurrent repair and normal operation

Since web applications are often serving many users, it's undesirable to take the application offline while recovering from an intrusion. To address this problem, WARP's database manager introduces the notion of *repair generations*, identified by an integer counter, which are used to denote the state of the database after a given number of repairs. Normal execution happens in the *current* repair generation. When repair is initiated, the database manager creates the *next* repair generation (by incrementing the current repair generation counter by one), which creates a fork of the current database contents. All database operations during repair are applied to the next generation. If, during repair, users make changes to parts of the current generation that are being repaired, WARP will re-apply the users' changes to the next generation through re-execution. Changes to parts of the database not under repair are copied verbatim into the next generation. Once repair is near completion, the web server is briefly suspended, any final requests are re-applied to the next generation, the current generation is set to the next generation, and the web server is resumed.

To implement repair generations, WARP augments all tables with two additional columns, start_gen and end_gen, which indicate the generations in which a row is valid. Much as with continuous versioning, end_gen $= \infty$ indicates that the row has not been superseded in any later generation. During normal execution, queries execute over rows that match start_gen $\leq$ *current* and end_gen $\geq$ *current*. During repair, if a row with start_gen $<$ *next* and end_gen $\geq$ *next* is about to be updated or deleted (due to either re-execution or rollback), the existing row's end_gen is set to *current*, and, in case of updates, the update is executed on a copy of the row with start_gen $=$ *next*.

4.4 Rewriting SQL queries

WARP intercepts all SQL queries made by the application, and transparently rewrites them to implement database versioning and generations. For each query, WARP determines the time and generation in which the query should execute. For queries issued as part of normal execution, WARP uses the current time and generation. For queries issued as part of repair, WARP's repair controller explicitly specifies the time for the re-executed query, and the query always executes in the *next* generation.

To execute a SELECT query at time T in generation G, WARP restricts the query to run over currently valid rows by augmenting its WHERE clause with AND start_time $\leq T \leq$ end_time AND start_gen $\leq G \leq$ end_gen.

During normal execution, on an UPDATE or DELETE query at time T (the current time), WARP implements versioning by making a copy of the rows being modified. To do this, WARP sets the end_time of rows being modified in the *current* generation to T, and inserts copies of the rows with start_time $\leftarrow T$, end_time $\leftarrow \infty$, start_gen $\leftarrow G$, and end_gen $\leftarrow \infty$, where $G =$ *current*. WARP also restricts the WHERE clause of such queries to run over currently valid rows, as with SELECT queries above. On an INSERT query, WARP sets start_time, end_time, start_gen, and end_gen columns of the inserted row as for UPDATE and DELETE queries above.

To execute an UPDATE or DELETE query during repair at time T, WARP must first preserve any rows being modified that are also accessible from the *current* generation, so that they continue to be accessible to concurrently executing queries in the *current* generation. To do so, WARP creates a copy

of all matching rows, with `end_gen` set to $current$, sets the `start_gen` of the rows to be modified to $next$, and then executes the `UPDATE` or `DELETE` query as above, except in generation $G = next$. Executing an `INSERT` query during repair does not require preserving any existing rows; in this case, WARP simply performs the same query rewriting as for normal execution, with $G = next$.

5 BROWSER RE-EXECUTION

To help users recover from attacks that took place in their browsers, WARP uses two ideas. First, when WARP determines that a past HTTP response was incorrect, it *re-executes the changed web page in a cloned browser* on the server, in order to determine how that page would behave as a result of the change. For example, if a new HTTP response no longer contains an adversary's JavaScript code (e.g., because the cross-site scripting vulnerability was retroactively patched), re-executing the page in a cloned browser will not generate the HTTP requests that the attacker's JavaScript code may have originally initiated, and will thus allow WARP to undo those requests.

Second, WARP performs *DOM-level replay of user input* when re-executing pages in a browser. By recording and re-executing user input at the level of the browser's DOM, WARP can better capture the user's intent as to what page elements the user was trying to interact with. A naïve approach that recorded pixel-level mouse events and key strokes may fail to replay correctly when applied to a page whose HTML code has changed slightly. On the other hand, DOM elements are more likely to be unaffected by small changes to an HTML page, allowing WARP to automatically re-apply the user's original inputs to a modified page during repair.

5.1 Tracking page dependencies

In order to determine what should be re-executed in the browser given some changes on the server, WARP needs to be able to correlate activity on the server with activity in users' browsers.

First, to correlate requests coming from the same web browser, WARP's browser extension assigns each client a unique client ID value. The client ID also helps WARP keep track of log information uploaded to the server by different clients. The client ID is a long random value to ensure that an adversary cannot guess the client ID of a legitimate user and upload logs on behalf of that user.

Second, WARP also needs to correlate different HTTP requests coming from the same page in a browser. To do this, WARP introduces the notion of a *page visit*, corresponding to the period of time that a single web page is open in a browser frame (e.g., a tab, or a sub-frame in a window). If the browser loads a new page in the same frame, WARP considers this to be a new visit (regardless of whether the frame navigated to a different URL or to the same URL), since the frame's page starts executing in the browser anew. In particular, WARP's browser extension assigns each page visit a visit ID, unique within a client. Each page visit can also have a dependency on a previous page visit. For example, if the user clicks on a link as part of page visit #1, the browser extension creates page visit #2, which depends on page visit #1. This allows WARP to check whether page visit #2 needs to re-execute if page visit #1 changes. If the user clicks on more links, and later hits the back button to return to the page from visit #2, this creates a fresh page visit #N (for the same page URL as visit #2), which also depends on visit #1.

Finally, WARP needs to correlate HTTP requests issued by the web browser with HTTP requests received by the HTTP server, for tracking dependencies. To do this, the WARP browser extension assigns each HTTP request a request ID, unique within a page visit, and sends the client ID, visit ID, and request ID along with every HTTP request to the server via HTTP headers.

On the server side, the HTTP server's manager records dependencies between HTTP requests and responses (identified by a $\langle client\_id, visit\_id, request\_id \rangle$ tuple) and runs of application code (identified by a $\langle pid, count \rangle$ tuple, where pid is the PID of the long-lived PHP runtime process, and $count$ is a unique counter identifying a specific run of the application).

5.2 Recording events

During normal execution, the browser extension performs two tasks. First, it annotates all HTTP requests, as described above, with HTTP headers to help the server correlate client-side actions with server-side actions. Second, it records all JavaScript events that occur during each page visit (including timer events, user input events, and `postMessage` events). For each event, the extension records event

parameters, including time and event type, and the XPath of the event's target DOM element, which helps perform DOM-level replay during repair.

The extension uploads its log of JavaScript events for each page visit to the server, using a separate protocol (tagged with the client ID and visit ID). On the server side, WARP's HTTP server records the submitted information from the client into a separate per-client log, which is subject to its own storage quota and garbage-collection policy. This ensures that a single client cannot monopolize log space on the server, and more importantly, cannot cause a server to garbage-collect recent log entries from other users needed for repair.

Although the current WARP prototype implements client-side logging using an extension, the extension does not circumvent any of the browser's privacy policies. All of the information recorded by WARP's browser extension can be captured at the JavaScript level by event handlers, and in future work, we hope to implement an extension-less version of WARP's browser logging by interposing on all events using JavaScript rewriting.

5.3 Server-side re-execution

When WARP determines that an HTTP response changed during repair, the browser repair manager spawns a browser on the server to re-execute the client's uploaded browser log for the affected page visit. This re-execution browser loads the client's HTTP cookies, loads the same URL as during original execution, and replays the client's original DOM-level events. The user's cookies are loaded either from the HTTP server's log, if re-executing the first page for a client, or from the last browser page re-executed for that client. The re-executed browser runs in a sandbox, and only has access to the client's HTTP cookie, ensuring that it gets no additional privileges despite running on the server. To handle HTTP requests from the re-executing browser, the HTTP server manager starts a separate copy of the HTTP server, which passes any HTTP requests to the repair controller, as opposed to executing them directly. This allows the repair controller to prune re-execution for identical requests or responses.

During repair, WARP uses a *re-execution extension* in the server-side browser to replay the events originally recorded by the user's browser. For each event, the re-execution extension tries to locate the appropriate DOM element using its XPath. For keyboard input events into text fields, the re-execution extension performs a three-way text merge between the original value of the text field, the new value of the text field during repair, and the user's original keyboard input. For example, this allows the re-execution extension to replay the user's changes to a text area when editing a Wiki page, even if the Wiki page in the text area is somewhat different during repair.

If, after repair, a user's HTTP cookie in the cloned browser differs from the user's cookie in his or her real browser (based on the original timeline), WARP queues that client's cookie for invalidation, and the next time the same client connects to the web server (based on the client ID), the client's cookie will be deleted. WARP assumes that the browser has no persistent client-side state aside from the cookie. Repair of other client-side state could be similarly handled at the expense of additional logging and synchronization.

5.4 Conflicts

During repair, the server-side browser extension may fail to re-execute the user's original inputs, if the user's actions somehow depended on the reverted actions of the attacker. For example, in the case of a Wiki page, the user may have inadvertently edited a part of the Wiki page that the attacker modified. In this situation, WARP's browser repair manager signals a conflict, stops re-execution of that user's browser, and requires the user (or an administrator, in lieu of the user) to resolve the conflict.

Since users are not always online, WARP queues the conflict for later resolution, and proceeds with repair, assuming, for now, that subsequent requests from that user's browser do not change. When the user next logs into the web application (based on the client ID), the application redirects the user to a conflict resolution page, which tells the user about the page on which the conflict arose, and the user's input which could not be replayed. The user must then indicate how the conflict should be resolved. For example, the user can indicate that they would like to cancel the conflicted page visit altogether (i.e., undo all of its HTTP requests), and apply the legitimate changes (if any) to the current state of the system by hand.

Component	Lines of code
Firefox extension	2,000 lines of JavaScript / HTML
Apache logging module	900 lines of C
PHP runtime / SQL rewriter	1,400 lines of C and PHP
PHP re-execution support	200 lines of Python
Repair managers:	4,300 lines of Python, total
Retro's repair controller	400 lines of Python
PHP manager	800 lines of Python
Apache manager	300 lines of Python
Database manager	1,400 lines of Python and PHP
Firefox manager	400 lines of Python
Retroactive patching manager	200 lines of Python
Others	800 lines of Python

Table 1: Lines of code for different components of the WARP prototype, excluding blank lines and comments.

While WARP's re-execution extension flags conflicts that arise during replay of input *from* the user, some applications may have important information that must be correctly displayed *to* the user. For example, if an online banking application displayed $1,000 as the user's account balance during the original execution, but during repair it is discovered that the user's balance should have been $2,000, WARP will not raise a re-execution conflict. An application programmer, however, can provide a *UI conflict function*, which, given the old and new versions of a web page, can signal a conflict even if all of the user input events replay correctly. For the example applications we evaluated with WARP, we did not find the need to implement such conflict functions.

5.5 User-initiated repair

In some situations, users or administrators may want to undo their own past actions. For example, an administrator may have accidentally granted administrative privileges to a user, and later may want to revert any actions that were allowed due to this mis-configuration. To recover from this mistake, the administrator can use WARP's browser extension to specify a URL of the page on which the mistake occurred, find the specific page visit to that URL which led to the mistake, and request that the page visit be canceled. Our prototype does not allow replacing one past action with another, although this is mostly a UI limitation.

Allowing users to undo their own actions runs the risk of creating more conflicts, if other users' actions depended on the action in question. To prevent cascading conflicts, WARP prohibits a regular user (as opposed to an administrator) from initiating repair that causes conflicts for other users. WARP's repair generation mechanism allows WARP to try repairing the server-side state upon user-initiated repair, and to abort the repair if any conflicts arise. The only exception to this rule is if the user's repair is a result of a conflict being reported to that user on that page, in which case the user is allowed to cancel all actions, even if it propagates a conflict to another user.

6 IMPLEMENTATION

We have implemented a prototype of WARP which builds on Retro. Our prototype works with the Firefox browser on the client, and Apache, PostgreSQL, and PHP on the server. Table 1 shows the lines of code for the different components of our prototype.

Our Firefox extension intercepts all HTTP requests during normal execution and adds WARP's client ID, visit ID, and request ID headers to them. It also intercepts all browser frame creations, and adds an event listener to the frame's window object. This event listener gets called on every event in the frame, and allows us to record the event. During repair, the re-execution extension tries to match up HTTP requests with requests recorded during normal execution, and adds the matching request ID header when a match is found. Our current conflict resolution UI only allows the user to cancel the conflicting page visit; other conflict resolutions must be performed by hand. We plan to build a more comprehensive UI, but canceling has been sufficient for now.

In our prototype, the user's client-side browser and the server's re-execution browser use the same version of Firefox. While this has simplified the development of our extension, we expect that DOM-level events are sufficiently standardized in modern browsers that it would be possible to replay

events across different browsers, such as recent versions of Firefox and Chrome. We have not verified this to date, however.

Our time-travel database and repair generations are implemented on top of PostgreSQL using SQL query rewriting. After the application's database tables are installed, WARP extends the schema of all the tables to add its own columns, including `row_id` if no existing column was specified as the row ID by the programmer. All database queries are rewritten to update these columns appropriately when the rows are modified. The approach of using query rewriting was chosen to avoid modifying the internals of the Postgres server, although an implementation inside of Postgres would likely have been more efficient.

To allow multiple versions of a row from different times or generations to exist in the same table, WARP modifies database uniqueness constraints and primary keys specified by the application to include the `end_ts` and `end_gen` columns. While this allows multiple versions of the same row over time to co-exist in the same table, WARP must now detect dependencies between queries through uniqueness violations. In particular, WARP checks whether the success (or failure) of each `INSERT` query would change as a result of other rows inserted or deleted during repair, and rolls back that row if so. WARP needs to consider `INSERT` statements only for partitions under repair. Our time-travel database implementation does not support foreign keys, so it disables them. We plan to implement foreign key constraints in the future in a database trigger. Our design is compatible with multi-statement transactions; however, our current implementation does not support them, and we did not need them for our current applications.

WARP extends Apache's PHP module to log HTTP requests that invoke PHP scripts. WARP intercepts a PHP script's calls to database functions, `mt_rand`, date and time functions, and `session_start`, by rewriting all scripts to call a wrapper function that invokes the wrapped function and logs the arguments and results.

7 PUTTING IT ALL TOGETHER

We now illustrate how different components of WARP work together in the context of a simple Wiki application. In this case, no attack takes place, but most of the steps taken by WARP remain the same as in a case with an attack.

Consider a user who, during normal execution, clicks on a link to edit a Wiki page. The user's browser issues an HTTP request to `edit.php`. WARP's browser extension intercepts this request, adds client ID, visit ID, and request ID HTTP headers to it, and records the request in its log (§5.1). The web server receives this request and dispatches it to WARP's PHP module. The PHP module assigns this request a unique server-side request ID, records the HTTP request information along with the server-side request ID, and forwards the request to the PHP runtime.

As WARP's PHP runtime executes `edit.php`, it intercepts three types of operations. First, for each non-deterministic function call, it records the arguments and the return value (§3.1). Second, for each operation that loads an additional PHP source file, it records the file name (§3.1). Third, for each database query, it records the query, rewrites the query to implement WARP's time-travel database, and records the result set and the row IDs of all rows modified by the query (§4).

Once `edit.php` completes execution, the response is recorded by the PHP module and returned to the browser. When the browser loads the page, WARP's browser extension attaches handlers to intercept user input, and records all intercepted actions in its log (§5.2). The WARP browser extension periodically uploads its log to the server.

When a patch fixing a vulnerability in `edit.php` becomes available, the administrator instructs WARP to perform retroactive patching. The WARP repair controller uses the action history graph to locate all PHP executions that loaded `edit.php` and queues them for re-execution; the user edit action described above would be among this set.

To re-execute this page in repair mode, the repair controller launches a browser on the server, identical to the user's browser, and instructs it to replay the user session. The browser re-issues the same requests, and the WARP browser extension assigns the same IDs to the request as during normal execution (§5.3). The WARP PHP module forwards this request to the repair controller, which launches WARP's PHP runtime to re-execute it.

During repair, the PHP runtime intercepts two types of operations. For non-deterministic function calls, it checks whether the same function was called during the original execution, and if so, re-uses the original return value (§3.3). For database queries, it forwards the query to the repair controller for re-execution.

To re-execute a database query, the repair controller determines the rows and partitions that the query depends on, rolls them back to the right version (for a write operation), rewrites the query to support time-travel and generations, executes the resulting query, and returns the result to the PHP runtime (§4).

After a query re-executes, the repair controller uses the action history graph to find other database queries that depended on the partitions affected by the re-executed query (assuming it was a write). For each such query, the repair controller checks whether their return values would now be different. If so, it queues the page visits that issued those queries for re-execution.

After `edit.php` completes re-execution, the HTTP response is returned to the repair controller, which forwards it to the re-executing browser via the PHP module. Once the response is loaded in the browser, the WARP browser extension replays the original user inputs on that page (§5.3). If conflicts arise, WARP flags them for manual repair (§5.4).

WARP's repair controller continues repairing pages in this manner until all affected pages are re-executed. Even though no attack took place in this example, this re-execution algorithm would repair from any attack that exploited the vulnerability in `edit.php`.

8 EVALUATION

In evaluating WARP, we answer several questions. §8.1 shows what it takes to port an existing web application to WARP. §8.2 shows what kinds of attacks WARP can repair from, what attacks can be detected and fixed with retroactive patching, how much re-execution may be required, and how often users need to resolve conflicts. §8.3 shows the effectiveness of WARP's browser re-execution in reducing user conflicts. §8.4 compares WARP with the state-of-the-art work in data recovery for web applications [1]. Finally, §8.5 measures WARP's runtime cost.

We ported a popular Wiki application, MediaWiki [21], to use WARP, and used several previously discovered vulnerabilities to evaluate how well WARP can recover from intrusions that exploit those bugs. The results show that WARP can recover from six common attack types, that retroactive patching detects and repairs all tested software bugs, and that WARP's techniques reduce re-execution and user conflicts. WARP's overheads are 24–27% in throughput and 2–3.2 GB/day of storage.

8.1 Application changes

We did not make any changes to MediaWiki source code to port it to WARP. To choose row IDs for each MediaWiki table, we picked a primary or unique key column whose value MediaWiki assigns once during creation of a row and never overwrites. If there is no such column in a table, WARP adds a new `row_id` column to the table, transparent to the application. We chose partition columns for each table by analyzing the typical queries made by MediaWiki and picking the columns that are used in the `WHERE` clauses of a large number of queries on that table. In all, this required a total of 89 lines of annotation for MediaWiki's 42 tables.

8.2 Recovery from attacks

To evaluate how well WARP can recover from intrusions, we constructed six worst-case attack scenarios based on five recent vulnerabilities in MediaWiki and one configuration mistake by the administrator, shown in Table 2. After each attack, users browse the Wiki site, both reading and editing Wiki pages. Our scenarios purposely create significant interaction between the attacker's changes and legitimate users, to stress WARP's recovery aspects. If WARP can disentangle these challenging attacks, it can also handle any simpler attack.

In the *stored XSS attack*, the attacker injects malicious JavaScript code into a MediaWiki page. When a victim visits that Wiki page, the attacker's JavaScript code appends text to a second Wiki page that the victim has access to, but the attacker does not. The *SQL injection* and *reflected XSS attacks* are similar in design. Successful recovery from these three attacks requires deleting the attacker's JavaScript code; detecting what users were affected by that code; undoing the effects of the JavaScript code in their browsers (i.e., undoing the edits to the second page); verifying that the appended text

did not cause browsers of users that visited the second page to misbehave; and preserving all users' legitimate actions.

The *CSRF attack* is a login CSRF attack, where the goal of the attacker is to trick the victim into making her edits on the Wiki under the attacker's account. When the victim visits the attacker's site, the attack exploits the CSRF vulnerability to log the victim out of the Wiki site and log her back in under the attacker's account. The victim then interacts with the Wiki site, believing she is logged in as herself, and edits various pages. A successful repair in this scenario would undo all of the victim's edits under the attacker's account, and re-apply them under the victim's own account.

In the *clickjacking attack*, the attacker's site loads the Wiki site in an invisible frame and tricks the victim into thinking she is interacting with the attacker's site, while in fact she is unintentionally interacting with the Wiki site, logged in as herself. Successful repair in this case would undo all modifications unwittingly made by the user through the clickjacked frame.

We used retroactive patching to recover from all the above attacks, with patches implementing the fixes shown in Table 2.

Finally, we considered a scenario where the administrator of the Wiki site mistakenly grants a user access to Wiki pages she should not have been given access to. At a later point of time, the administrator detects the misconfiguration, and initiates undo of his action using WARP. Meanwhile, the user has used her elevated privileges to edit pages that she should not have been able to edit in the first place. Successful recovery, in this case, would undo all the modifications by the unprivileged user.

For each of these scenarios we ran a workload with 100 users. For all scenarios except the ACL error scenario, we have one attacker, three victims that were subject to attack, and 96 unaffected users. For the ACL error scenario, we have one administrator, one unprivileged user that takes advantage of the administrator's mistake, and 98 other users. During the workloads, all users login, read, and edit Wiki pages. In addition, in all scenarios except the ACL error, the victims visit the attacker's web site, which launches the attack from their browser.

Table 3 shows the results of repair for each of these scenarios. First, WARP can successfully repair all of these attacks. Second, retroactive patching detects and repairs from intrusions due to all five software vulnerabilities; the administrator does not need to detect or track down the initial attacks. Finally, WARP has few user-visible conflicts. Conflicts arise either because a user was tricked by the attacker into performing some browser action, or because the user should not have been able to perform the action in the first place. The conflicts in the clickjacking scenario are of the first type; we expect users would cancel their page visit on conflict, since they did not mean to interact with the MediaWiki page on the attack site. The conflict in the ACL error scenario is of the second type, since the user no longer has access to edit the page; in this case, the user's edit has already been reverted, and the user can resolve the conflict by, perhaps, editing a different page.

8.3 Browser re-execution effectiveness

We evaluated the effectiveness of browser re-execution in WARP by considering three types of attack code, for an XSS attack. The first is a benign, *read-only* attack where the attacker's JavaScript code runs in the user's browser but does not modify any Wiki pages. The second is an *append-only* attack, where the malicious code appends text to the victim's Wiki page. Finally, the *overwrite* attack completely corrupts the victim's Wiki page.

We ran these attacks under three configurations of the client browser: First, without WARP's browser extension; second, with WARP's browser extension but without WARP's text merging for user input; and third, with WARP's complete browser extension. Our experiment had one attacker and eight victims. Each user logged in, visited the attack page to trigger one of the three above attacks, edited Wiki pages, and logged out.

Table 4 shows the results when WARP is invoked to retroactively patch the XSS vulnerability. Without WARP's browser extension, WARP cannot verify whether the attacker's JavaScript code was benign or not, and raises a conflict for every victim of the XSS attack. With the browser extension but without text-merging, WARP can verify that the *read-only* attack was benign, and raises no conflict, but cannot re-execute the user's page edits if the attacker did modify the page slightly, raising a conflict in that scenario. Finally, WARP's full browser extension is able to re-apply the user's page edits despite the attacker's appended text, and raises no conflict in that situation. When the attacker completely

corrupts the page, applying user's original changes in the absence of the attack is meaningless, and a conflict is always raised.

8.4 Recovery comparison with prior work

Here we compare WARP with state-of-the-art work in data recovery for web applications by Akkuş and Goel [1]. Their system uses taint tracking in web applications to recover from data corruption bugs. In their system, the administrator identifies the request that triggered the bug, and their system uses several dependency analysis policies to do offline taint analysis and compute dependencies between the request and database elements. The administrator uses these dependencies to manually undo the corruption. Each specific policy can output too many dependencies (false positives), leading to lost data, or too few (false negatives), leading to incomplete recovery.

Akkuş and Goel used five corruption bugs from popular web applications to evaluate their system. To compare WARP with their system, we evaluated WARP with four of these bugs—two each in Drupal and Gallery2. The remaining bug is in Wordpress, which does not support our Postgres database. Porting the buggy versions of Drupal and Gallery2 to use WARP did not require any changes to source code. We replicated each of the four bugs under WARP. Once we verified that the bugs were triggered, we retroactively patched the bug. Repair did not require any user input, and after repair, the applications functioned correctly without any corrupted data.

Table 5 summarizes this evaluation. WARP has three key advantages over Akkuş and Goel's system. First, unlike their system, WARP never incurs false negatives and always leaves the application in an uncorrupted state. Second, WARP only requires the administrator to provide the patch that fixes the bug, whereas Akkuş and Goel require the administrator to manually guide the dependency analysis by identifying requests causing corruption, and by whitelisting database tables. Third, unlike WARP, their system cannot recover from *attacks* on web applications, and cannot recover from problems that occur in the browser.

8.5 Performance evaluation

In this subsection, we evaluate WARP's performance under different scenarios. In these experiments, we ran the server on a 3.07 GHz Intel Core i7 950 machine with 12 GB of RAM. WARP's repair algorithm is currently sequential. Running it on a machine with multiple cores makes it difficult to reason about the CPU usage of various components of WARP; so we ran the server with only one core turned on and with hyperthreading turned off. However, during normal execution, WARP can take full advantage of multiple processor cores when available.

Logging overhead. We first evaluate the overhead of using WARP by measuring the performance of MediaWiki with and without WARP for two workloads: reading Wiki pages, and editing Wiki pages. The clients were 8 Firefox browsers running on a machine different from the server, sending requests as fast as possible; the server experienced 100% CPU load. The client and server machines were connected with a 1 Gbps network.

Table 6 shows the throughput of MediaWiki with and without WARP, and the size of WARP's logs. For the reading and editing workloads, respectively, WARP incurs throughput overheads of 24% and 27%, and storage costs of 3.71 KB and 7.34 KB per page visit (or 2 GB/day and 3.2 GB/day under continuous 100% load). Many web applications already store similar log information; a 1 TB drive could store about a year's worth of logs at this rate, allowing repair from attacks within that time period. We believe that this overhead would be acceptable to many applications, such as a company's Wiki or a conference reviewing web site.

To evaluate the overhead of WARP's browser extension, we measured the load times of a Wiki page in the browser with and without the WARP extension. This experiment was performed with an unloaded MediaWiki server. The load times were 0.21 secs and 0.20 secs with and without the WARP extension respectively, showing that the WARP browser extension imposes negligible overhead.

Finally, WARP indexes its logs to support incremental loading of its dependency graph during repair. In our current prototype, for convenience, indexing is implemented as a separate step after normal execution. This indexing step takes 24–28 ms per page visit for the workloads we tested. If done during normal execution, this would add less than an additional 12% overhead.

Repair performance. We evaluate WARP's repair performance by considering four scenarios. First, we consider a scenario where a retroactive patch affects a small, isolated part of the action history graph. This scenario evaluates WARP's ability to efficiently load and redo only the affected actions. To evaluate this scenario, we used the XSS, SQL injection, and ACL error workloads from §8.2 with 100 users, and victim page visits at the end of the workload. The results are shown in the first four rows of Table 7. The re-executed actions columns show that WARP re-executes only a small fraction of the total number of actions in the workload, and a comparison of the original execution time and total repair time columns shows that repair in these scenarios takes an order of magnitude less time than the original execution time.

Second, we evaluate a scenario where the patch affects a small part of the action history graph as before, but the affected actions in turn may affect several other actions. To test this scenario, we used the reflected XSS workload with 100 users, but with victims at the beginning of the workload, rather than at the end. Re-execution of the victims' page visits in this case causes the database state to change, which affects non-victims' page visits. This scenario tests WARP's ability to track database dependencies and selectively re-execute database queries without having to re-execute non-victim page visits. The results for this scenario are shown in the fifth row of Table 7.

A comparison of the results for both the reflected XSS attack scenarios shows that WARP re-executes the same number of page visits in both cases, but the number of database queries is significantly greater when victims are at the beginning. These extra database queries are queries from non-victim page visits which depend on the database partitions that changed as a result of re-executing victim pages. These queries are of two types: `SELECT` queries that need to be re-executed to check whether their result has changed, and `UPDATE` queries that need to be re-executed to update the rolled-back database rows belonging to the affected database partitions. From the repair time breakdown columns, we see that the graph loading for these database query actions and their re-execution are the main contributors to the longer repair time for this scenario, as compared to when victims were at the end of the workload. Furthermore, we see that the total repair time is about one-third of the time for original execution, and so WARP's repair is significantly better than re-executing the entire workload.

Third, we consider a scenario where a patch requires all actions in the history to be re-executed. We use the CSRF and clickjacking attacks as examples of this scenario. The results are shown in the last two rows of Table 7. WARP takes an order of magnitude more time to re-execute all the actions in the graph than the original execution time. Our unoptimized repair controller prototype is currently implemented in Python, and the step-by-step re-execution of the repaired actions is a significant contributor to this overhead. We believe implementing WARP in a more efficient language, such as C++, would significantly reduce this overhead.

Finally, we evaluate how WARP scales to larger workloads. We measure WARP's repair performance for XSS, SQL injection, and ACL error workloads, as in the first scenario, but with 5,000 users instead of 100. The results for this experiment are shown in Table 8. The number of actions affected by the attack remain the same, and only those actions are re-executed as part of the repair. This indicates WARP successfully avoids re-execution of requests that were not affected by the attack. Differences in the number of re-executed actions (e.g., in the stored XSS attack) are due to non-determinism introduced by MediaWiki object caching. We used a stock MediaWiki installation for our experiments, in which MediaWiki caches results from past requests in an `objectcache` database table. During repair, MediaWiki may invalidate some of the cache entries, resulting in more re-execution.

The repair time for the 5,000-user workload is only $3\times$ the repair time for 100 users, for all scenarios except SQL injection, despite the $50\times$ increase in the overall workload. This suggests that WARP's repair time does not increase linearly with the size of the workload, and is mostly determined by the number of actions that must be re-executed during repair. The SQL injection attack had a $10\times$ increase in repair time because the number of database rows affected by the attack increases linearly with the number of users. The attack injects the SQL query `UPDATE pagecontent SET old_text = old_text || 'attack'`, which modifies every page. Recovering from this attack requires rolling back all the users' pages, and the time to do that increases linearly with the total number of users.

Concurrent repair overhead. When repair is ongoing, WARP allows the web application to continue normal operation using repair generations. To evaluate repair generations, we measured the

performance of MediaWiki for the read and edit workloads from §8.5 while repair is underway for the CSRF attack.

The results are shown in the "During repair" column of Table 6. They demonstrate that WARP allows MediaWiki to be online and functioning normally while repair is ongoing, albeit at a lower performance—with 24% to 30% lower number of page visits per second than if there were no repair in progress. The drop in performance is due to both repair and normal execution sharing the same machine resources. This can be alleviated if dedicated resources (e.g., a dedicated processor core) were available for repair.

9 RELATED WORK

The two closest pieces of work related to WARP are the Retro intrusion recovery system [14] and the web application data recovery system by Akkuş and Goel [1].

While WARP builds on ideas from Retro, Retro focuses on shell-oriented Unix applications on a single machine. WARP extends Retro with three key ideas to handle web applications. First, Retro requires an intrusion detection system to detect attacks, and an expert administrator to track down the root cause of every intrusion; WARP's retroactive patching allows an administrator to simply supply a security patch for the application's code. Second, Retro's file- and process-level rollback and dependency tracking cannot perform fine-grained rollback and dependency analysis for individual SQL queries that operate on the same table, and cannot perform online repair, and WARP's time-travel database can.[3] Third, repairing any network I/O in Retro requires user input; in a web application, this would require every user to resolve conflicts at the TCP level. WARP's browser re-execution eliminates the need to resolve most conflicts, and presents a meaningful UI for true conflicts that require user input.

Akkuş and Goel's data recovery system uses taint tracking to analyze dependencies between HTTP requests and database elements, and thereby recover from data corruption errors in web applications. However, it can only recover from accidental mistakes, as opposed to malicious attacks (in part due to relying on white-listing to reduce false positives), and requires administrator guidance to reduce false positives and false negatives. WARP can fully recover from data corruptions due to bugs as well as attacks, with no manual intervention (except when there are conflicts during repair). §8.4 compared WARP to Akkuş and Goel's system in more detail.

Provenance-aware storage systems [24, 26] record dependency information similar to WARP, and can be used by an administrator to track down the effects of an intrusion or misconfiguration. Margo and Seltzer's browser provenance system [20] shows how provenance information can be extended to web browsers. WARP similarly tracks provenance information across web servers and browsers, and aggregates this information at the server, but WARP also records sufficient information to re-execute browser events and user input in a new context during repair. However, our WARP prototype does not help users understand the provenance of their own data.

Ibis [28] and PASSv2 [25] show how to incorporate provenance information across multiple layers in a system. While WARP only tracks dependencies at a fixed level (SQL queries, HTTP requests, and browser DOM events), we hope to adopt ideas from these systems in the future, to recover from intrusions that span many layers (e.g., the database server or the language runtime).

WARP's idea of retroactive patching provides a novel approach to intrusion detection, which can be used on its own to detect whether recently patched vulnerabilities have been exploited before the patch was applied. Work on vulnerability-specific predicates [13] is similar in its use of re-execution (at the virtual machine level), but requires writing specialized predicates for each vulnerability, whereas WARP only requires the patch itself.

Much of the work on intrusion detection and analysis [5, 11, 15, 16, 18, 32] is complementary to WARP, and can be applied in parallel. When an intrusion is detected and found using an existing intrusion detection tool, the administrator can use WARP to recover from the effects of that intrusion in a web application.

Polygraph [19] recovers from compromises in a weakly consistent replication system. Unlike WARP, Polygraph does not attempt to preserve legitimate changes to affected files, and does not

[3] One of Retro's scenarios involved database repair, but it worked by rolling back the entire database file, and re-executing every SQL query.

attempt to automate detection of compromises. Polygraph works well for applications that do not operate on multiple files at once. In contrast, WARP deals with web applications, which frequently access shared data in a single SQL database.

Tracking down and reverting malicious actions has been explored in the context of databases [2, 17]. WARP cannot rely purely on database transaction dependencies, because web applications tend to perform significant amounts of data processing in the application code and in web browsers, and WARP tracks dependencies across all those components. WARP's time-travel database is in some ways reminiscent of a temporal database [29, 30]. However, unlike a temporal database, WARP has no need for more complex temporal queries; supports two time-like dimensions (wall-clock time and repair generations); and allows partitioning rows for dependency analysis.

Many database systems exploit partitioning for performance; WARP uses partitioning for dependency analysis. The problem of choosing a suitable partitioning has been addressed in the context of minimizing distributed transactions on multiple machines [3], and in the context of index selection [6, 12]. These techniques might be helpful in choosing a partitioning for tables in WARP.

Mugshot [22] performs deterministic recording and replay of JavaScript events, but cannot replay events on a changed web page. WARP must replay user input on a changed page in order to re-apply legitimate user changes after effects of the attack have been removed from a page. WARP's DOM-level replay matches event targets between record and replay even if other parts of the page differ.

10 DISCUSSION AND LIMITATIONS

While our prototype depends on a browser extension to record client-side events and user input, we believe it would be possible to do so in pure JavaScript as well. In future work, we plan to explore this possibility, perhaps leveraging Caja [23] to wrap existing JavaScript code and record all browser events and user input; the browser's same-origin policy already allows JavaScript code to perform all of the necessary logging. We also plan to verify that DOM-level events recorded in one browser can be re-executed in a different standards-compliant browser. In the meantime, we note that operators of complex web applications often already have an infrastructure of virtual machines and mobile phone emulators for testing across browser platforms, and a similar infrastructure could be used for WARP's repair.

The client-side logs, uploaded by WARP's extension to the server, can contain sensitive information. For example, if a user enters a password on one of the pages of a web application, the user's key strokes will be recorded in this log, in case that page visit needs to be re-executed at a later time. Although this information is accessible to web applications even without WARP, applications might not record or store this information on their own, and WARP must safeguard this additional stored information from unintended disclosure.

In future work, we plan to explore ways in which WARP-aware applications can avoid logging known-sensitive data, such as passwords, by modifying replay to assume that a valid (or invalid) password was supplied, without having to re-enter the actual password. The logs can also be encrypted so that the administrator must provide the corresponding decryption key to initiate repair. An alternative design—storing the logs locally on each client machine and relying on client machines to participate in the repair process—would prevent a single point of compromise for all logs, but would make complete repair a lengthy process, since each client machine will have to come online to replay its log.

WARP's current design cannot re-execute mashup web applications (i.e., those involving multiple web servers), since the event logs for each web application's frame would be uploaded to a different web server. We plan to explore re-execution of such multi-origin web applications, as long as all of the web servers involved in the mashup support WARP. The approach we imagine taking is to have the client sign each event that spans multiple origins (such as a `postMessage` between frames) with a private key corresponding to the source origin. This would allow WARP re-executing at the source origin's server to convince WARP on the other frame's origin server that it should be allowed to initiate re-execution for that user.

Retroactive patching by itself cannot be used to recover from attacks that resulted from leaked credentials. For example, an attacker can use an existing XSS vulnerability in an application to steal a user's credentials and use them to impersonate the user and perform unauthorized actions. Retroactive patching of the XSS vulnerability cannot distinguish the actions of the attacker's browser

from legitimate actions of the user's browser, as both used the same credentials. However, if the user is willing to identify the legitimate browsers, WARP can undo the actions performed by the attacker's browser.

We plan to explore tracking dependencies at multiple levels of abstraction, borrowing ideas from prior work [14, 25, 28]. This may allow WARP to recover from compromises in lower layers of abstraction, such as a database server or the application's language runtime. We also hope to extend WARP's undo mechanism higher into the application, to integrate with application-level undo features, such as MediaWiki's revert mechanism.

In our current prototype, we instrument the web application server to log HTTP requests and database queries. This requires that the application server be fully trusted to not tamper with WARP logging, and requires modification of the application server software, which may not always be possible. It also does not support replicated web application servers, as the logs for a replica contain the local times at that replica, which are not directly comparable to local times at other replicas. In future work, we plan to explore an alternative design with WARP proxies in front of the application's HTTP load balancer and the database, and perform logging in those proxies. This design addresses the above limitations, but can lead to more re-execution during repair, as it does not capture the exact database queries made for each HTTP request.

We also plan to explore techniques to further reduce the number of application runs re-executed due to retroactive patching, by determining which runs actually invoked the patched function, instead of the runs that just loaded the patched file.

Our current prototype assumes that the application code does not change, other than through retroactive patching. While this assumption is unrealistic, fixing it is straightforward. WARP's application repair manager would need to record each time the application's source code changed. Then, during repair, the application manager would roll back these source code changes (when rolling back to a time before these changes were applied), and would re-apply these patches as the repaired timeline progressed (in the process merging these original changes with any newly supplied retroactive patches).

11 SUMMARY

This paper presented WARP, an intrusion recovery system for web applications. WARP introduced three key ideas to make intrusion recovery practical. *Retroactive patching* allows administrators to recover from past intrusions by simply supplying a new security patch, without having to even know if an attack occurred. The *time-travel database* allows WARP to perform precise repair of just the affected parts of the system. Finally, *DOM-level replay of user input* allows WARP to preserve legitimate changes with no user input in many cases. A prototype of WARP can recover from attacks, misconfigurations, and data loss bugs in three applications, without requiring any code changes, and with modest runtime overhead.

ACKNOWLEDGMENTS

We thank Victor Costan, Frans Kaashoek, Robert Morris, Jad Naous, Hubert Pham, Eugene Wu, the anonymous reviewers, and our shepherd, Yuanyuan Zhou, for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, by NSF award CNS-1053143, by Quanta, and by Google. Taesoo Kim is partially supported by the Samsung Scholarship Foundation. The opinions in this paper do not necessarily represent DARPA or official US policy.

REFERENCES

[1] İ. E. Akkuş and A. Goel. Data recovery for web applications. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, Jun–Jul 2010.

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

[3] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[4] Damon Cortesi. Twitter StalkDaily worm postmortem. `http://dcortesi.com/2009/04/11/twitter-stalkdaily-worm-postmortem/`.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, Dec 2002.

[6] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.

[7] C. Goldfeder. Gmail snooze with apps script. `http://googleappsdeveloper.blogspot.com/2011/07/gmail-snooze-with-apps-script.html`.

[8] D. Goodin. Surfing Google may be harmful to your security. *The Register*, Aug 2008. `http://www.theregister.co.uk/2008/08/09/google_gadget_threats/`.

[9] Google, Inc. Google apps script. `http://code.google.com/googleapps/appsscript/`.

[10] S. Gordeychik. Web application security statistics. `http://www.webappsec.org/projects/statistics/`.

[11] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[12] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. Softw. Eng.*, 9(2):135–143, 1983.

[13] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 91–104, Brighton, UK, Oct 2005.

[14] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 89–104, Vancouver, Canada, Oct 2010.

[15] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, Feb 2005.

[16] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from Unix process execution traces for intrusion detection. In *Proceedings of the AAAI Workshop on AI Approaches in Fraud Detection and Risk Management*, pages 50–56, Jul 1997.

[17] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovering from malicious transactions. *Journal of Distributed and Parallel Databases*, 8:7–40, 2000.

[18] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, Jun 2008.

[19] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, Mar 2009.

[20] D. W. Margo and M. Seltzer. The case for browser provenance. In *Proceedings of the 1st Workshop on the Theory and Practice of Provenance*, San Francisco, CA, Feb 2009.

[21] MediaWiki. MediaWiki. `http://www.mediawiki.org`.

[22] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, San Jose, CA, Apr 2010.

[23] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008. `http://code.google.com/p/google-caja/downloads/list`.

[24] K.-K. Muniswamy-Reddy, D. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, May–Jun 2006.

[25] K.-K. Muniswamy-Reddy, U. Braun, D. Holland, P. Macko, D. Maclean, D. W. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, Jun 2009.

[26] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the cloud. In *Proceedings of the 8th Conference on File and Storage Technologies*, San Jose, CA, Feb 2010.

[27] National Vulnerability Database. CVE statistics. `http://web.nvd.nist.gov/view/vuln/statistics`, Feb 2011.

[28] C. Olston and A. D. Sarma. Ibis: A provenance manager for multi-layer systems. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, Pacific Grove, CA, Jan 2011.

[29] Oracle Corporation. Oracle flashback technology. `http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html`.

[30] R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, Sep 1986.

[31] J. Tyson. Recent Facebook XSS attacks show increasing sophistication. `http://theharmonyguy.com/2011/04/21/recent-facebook-xss-attacks/`, Apr 2011.

[32] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 20th IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[33] K. Wickre. About that fake post. `http://googleblog.blogspot.com/2006/10/about-that-fake-post.html`.

Attack type	CVE	Description	Fix
Reflected XSS	2009-0737	The user options (`wgDB*`) in the live web-based installer (`config/index.php`) are not HTML-escaped.	Sanitize all user options with `htmlspecialchars()` (r46889).
Stored XSS	2009-4589	The name of contribution link (`Special:Block?ip`) is not HTML-escaped.	Sanitize the `ip` parameter with `htmlspecialchars()` (r52521).
CSRF	2010-1150	HTML/API login interfaces do not properly handle an unintended login attempt (login CSRF).	Include a random challenge token in a hidden form field for every login attempt (r64677).
Clickjacking	2011-0003	A malicious website can embed MediaWiki within an `iframe`.	Add `X-Frame-Options:DENY` to HTTP headers (r79566).
SQL injection	2004-2186	The language identifier, `thelang`, is not properly sanitized in `SpecialMaintenance.php`.	Sanitize the `thelang` parameter with `wfStrencode()`.
ACL error	—	Administrator accidentally grants admin privileges to a user.	Revoke the user's admin privileges.

Table 2: Security vulnerabilities and corresponding fixes for MediaWiki. Where available, we indicate the revision number of each fix in MediaWiki's subversion repository, in parentheses.

Attack scenario	Initial repair	Repaired?	# users with conflicts
Reflected XSS	Retroactive patching	✓	0
Stored XSS	Retroactive patching	✓	0
CSRF	Retroactive patching	✓	0
Clickjacking	Retroactive patching	✓	3
SQL injection	Retroactive patching	✓	0
ACL error	Admin-initiated	✓	1

Table 3: WARP repairs the attack scenarios listed in Table 2. The initial repair column indicates the method used to initiate repair.

Attack action	Number of users with conflict		
	No extension	No text merge	WARP
read-only	8	0	0
append-only	8	8	0
overwrite	8	8	8

Table 4: Effectiveness of WARP UI repair. Each entry indicates whether a user-visible conflict was observed during repair. This experiment involved eight victim users and one attacker.

Bug causing corruption	Akkuş and Goel [1]		WARP	
	FP	User input	FP	User input
Drupal – lost voting info	89 / 0	Yes	0	No
Drupal – lost comments	95 / 0	Yes	0	No
Gallery2 – removing perms	82 / 10	Yes	0	No
Gallery2 – resizing images	119 / 0	Yes	0	No

Table 5: Comparison of WARP with Akkuş and Goel's system [1]. FP reports false positives. Akkuş and Goel can also incur false negatives, unlike WARP. False positives are reported for the *best* dependency policy in [1] that has no false negatives for these bugs, although there is no single best policy for that system. The numbers shown before and after the slash are without and with table-level white-listing, respectively.

Workload	Page visits / second			Data stored per page visit		
	No WARP	WARP	During repair	Browser	App.	DB
Reading	8.46	6.43	4.50	0.22 KB	1.49 KB	2.00 KB
Editing	7.19	5.26	4.00	0.21 KB	1.67 KB	5.46 KB

Table 6: Overheads for users browsing and editing Wiki pages in MediaWiki. The first numbers are page visits per second without WARP, with WARP installed, and with WARP while repair is concurrently underway. A single page visit in MediaWiki can involve multiple HTTP and SQL queries. Data stored per page visit includes all dependency information (compressed) and database checkpoints.

Attack scenario	Number of re-executed actions			Original	Repair time breakdown							
	Page visits	App. runs	SQL queries	exec. time	*Total*	Init	Graph	Firefox	DB	App.	Ctrl	Idle
Reflected XSS	14 / 1,011	13 / 1,223	258 / 24,746	180.04	17.87	2.44	0.13	1.21	1.24	2.45	8.99	1.41
Stored XSS	14 / 1,007	15 / 1,219	293 / 24,740	179.22	16.74	2.64	0.12	1.12	0.98	2.45	8.23	1.20
SQL injection	22 / 1,005	23 / 1,214	524 / 24,541	177.82	29.70	2.41	0.16	1.65	0.05	4.16	17.25	4.01
ACL error	13 / 1,000	13 / 1,216	185 / 24,326	176.52	10.75	0.54	0.49	1.04	0.03	2.25	6.04	0.35
Reflected XSS (victims at start)	14 / 1,011	14 / 1,223	1,800 / 24,741	178.21	66.67	2.50	14.46	1.27	26.13	2.23	14.12	5.97
CSRF	1,005 /1,005	1,007 / 1,217	19,799 / 24,578	174.97	1,644.53	159.99	0.46	52.01	0.70	174.04	1,222.05	35.27
Clickjacking	1,011 /1,011	995 / 1,216	23,227 / 24,641	174.31	1,751.74	162.49	0.45	52.19	0.75	171.18	1,320.89	43.78

Table 7: Performance of WARP in repairing attack scenarios described in Table 2 for a workload with 100 users. The "re-executed actions" columns show the number of re-executed actions out of the total number of actions in the workload. The execution times are in seconds. The "original execution time" column shows the CPU time taken by the web application server, including time taken by database queries. The "repair time breakdown" columns show, respectively, the total wall clock repair time, the time to initialize repair (including time to search for attack actions), the time spent loading nodes into the action history graph, the CPU time taken by the re-execution Firefox browser, the time taken by re-executed database queries that are not part of a page re-execution, time taken to re-execute page visits including time to execute database queries issued during page re-execution, time taken by WARP's repair controller, and time for which the CPU is idle during repair.

Attack scenario	Number of re-executed actions			Original exec. time	Repair time breakdown							
	Page visits	App. runs	SQL queries		*Total*	Init	Graph	Firefox	DB	App.	Ctrl	Idle
Reflected XSS	14 / 50,011	14 / 60,023	281 / 1,222,656	8,861.55	48.28	11.34	10.89	1.33	0.52	2.23	21.30	0.67
Stored XSS	32 / 50,007	33 / 60,019	733 / 1,222,652	8,841.67	56.50	11.49	11.10	2.10	0.04	5.58	23.98	2.22
SQL injection	26 / 50,005	27 / 60,014	578 / 1,222,495	8,875.06	273.40	14.57	15.98	7.37	0.09	4.85	118.18	112.36
ACL error	11 / 50,000	11 / 60,016	133 / 1,222,308	8,879.55	41.81	9.20	10.25	1.07	0.08	1.74	19.10	0.37

Table 8: Performance of WARP in attack scenarios for workloads of 5,000 users. See Table 7 for a description of the columns.